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
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
https://haskell.foundation/podcast/
The Haskell Unfolder

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

Next episode on Wednesday, 14 June, on a topic related to this talk!
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
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.
Why do we evaluate anything at all?
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).
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

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.

```haskell
null :: [a] -> Bool
null [] = True
null (_ : _) = False
```
Looking at definitions

Let’s start with our own definitions.

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

```haskell
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`. 
Let's assume \( n = 2 \):

\[
null (\text{enumFromTo } 0 \ 2)
\]
Equational reasoning

Let’s assume \( n = 2 \):

\[
\text{null (enumFromTo 0 2)}
\]
Equational reasoning

Let’s assume $n = 2$:

$$\begin{align*}
null \ (\text{enumFromTo} \ 0 \ 2) \\
= \ null \ (\text{if} \ 0 > 2 \ \text{then} \ [] \ \text{else} \ 0 : \ \text{enumFromTo} \ (0 + 1) \ 2)
\end{align*}$$
Equational reasoning

Let’s assume $n = 2$:

$$\text{null (enumFromTo 0 2)}$$

$$= \text{null (if } 0 > 2 \text{ then } [] \text{ else } 0 : \text{enumFromTo (0 + 1) 2)}$$

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

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

Let’s assume $n = 2$:

$$\text{null (enumFromTo 0 2)}$$
$$= \text{null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)}$$
$$= \text{null (if False then [] else 0 : enumFromTo (0 + 1) 2)}$$

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

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

Let’s assume $n = 2$:

$$
\text{null (enumFromTo 0 2)} = \text{null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)} = \text{null (if False then [] else 0 : enumFromTo (0 + 1) 2)}
$$
Equational reasoning

Let’s assume \( n = 2 \):

\[
\begin{align*}
\text{null} \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{null} \ (\text{if} \ 0 > 2 \ \text{then} \ [] \ \text{else} \ 0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{null} \ (\text{if} \ \text{False} \ \text{then} \ [] \ \text{else} \ 0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{null} \ (0 : \text{enumFromTo} \ (0 + 1) \ 2)
\end{align*}
\]
Equational reasoning

Let’s assume $n = 2$:

\[
\text{null (enumFromTo 0 2)} \\
\text{= null (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2)} \\
\text{= null (if False then [] else 0 : enumFromTo (0 + 1) 2)} \\
\text{= null (0 : enumFromTo (0 + 1) 2)}
\]
Equational reasoning

Let’s assume \( n = 2 \):

\[
\begin{align*}
null \ (\text{enumFromTo} \ 0 \ 2) &= null \ (if \ 0 > 2 \ then \ [] \ else \ 0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
&= null \ (if \ False \ then \ [] \ else \ 0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
&= null \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
&= False
\end{align*}
\]
Let’s assume \( n = 2 \) :

\[
\text{null (enumFromTo 0 2)} \\
= \text{null (if } 0 > 2 \text{ then } [] \text{ else } 0 : \text{enumFromTo (0 + 1) 2)} \\
= \text{null (if False then } [] \text{ else } 0 : \text{enumFromTo (0 + 1) 2)} \\
= \text{null (0 : enumFromTo (0 + 1) 2)} \\
= \text{False}
\]

Reduction sequence does not depend on \( n \), only on \( 0 > n \) being \( \text{False} \).
Equational reasoning

Let’s assume $n = 2$:

$$
\begin{align*}
\text{null } (\text{enumFromTo } 0 \ 2) \\
= \text{null } (\text{if } 0 > 2 \ \text{then } [] \ \text{else } 0 : \text{enumFromTo } (0 + 1) \ 2) \\
= \text{null } (\text{if } False \ \text{then } [] \ \text{else } 0 : \text{enumFromTo } (0 + 1) \ 2) \\
= \text{null } (0 : \text{enumFromTo } (0 + 1) \ 2) \\
= \text{False}
\end{align*}
$$

Reduction sequence does not depend on $n$, only on $0 > n$ being $\text{False}$.

Answer to our original question is constant space (and time).
null (0 : enumFromTo (0 + 1) 2)
null (0 : enumFromTo (0 + 1) 2)
Redexes

We generally have more than one redex (reducible expression). One aspect of lazy evaluation is that we are generally choosing the outermost redex.
We generally have more than one redex (reducible expression). One aspect of lazy evaluation is that we are generally choosing the outermost redex.
null (\(0 : \text{enumFromTo}(0 + 1)\ 2\))

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.
Why does anything happen at all?
Why does anything happen at all?

- We want to **print** the resulting **Bool**.
Why does anything happen at all?

- We want to **print** the resulting `Bool`.
- In order to print it, we have to know it.
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 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.
As can be observed by the reduction

\[
\text{null (0 : enumFromTo (0 + 1) 2)}
\]

\[
= \text{False}
\]

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

As can be observed by the reduction

\[
\text{null} (0 : \text{enumFromTo} (0 + 1) 2) \\
= \text{False}
\]

revealing the top-level constructor is sufficient to reduce \text{null}.

An expression is in \textbf{weak head-normal form (WHNF)} if it is a constructor application (or a lambda).
As can be observed by the reduction

\[
\text{null (0 : enumFromTo (0 + 1) 2)} = \text{False}
\]

revealing the top-level constructor is sufficient to reduce \text{null}.

An expression is in \textit{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 \( x \))
Aside: strict functions

A function $f$ is called **strict** if and only if $f \bot = \bot$.

(Here, $\bot$ 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 $f \perp = \perp$.

(Here, $\perp$ 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 $f \bot = \bot$.

(Here, $\bot$ 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”.
Is \texttt{null} strict?
Examples

Is `null` strict?

Yes!

```
GHCi> null undefined
*** Exception: Prelude.undefined
```
What is an example of a non-strict function?
Examples

What is an example of a non-strict function?

constZero :: a -> Int
constZero _ = 0
What is an example of a non-strict function?

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

```haskell
GHCi> constZero undefined
0
```
Identity

\[
\text{id} :: a \rightarrow a \\
\text{id} \ x = x
\]

Is \text{id} strict?
Identity

\[
\text{id} \quad :: \quad a \rightarrow a \\
\text{id} \; x = x
\]

Is \text{id} strict?

Yes!

\[
\text{GHCi} > \; \text{id} \; \text{undefined} \\
*** \text{Exception: Prelude.undefined}
\]
**Identity**

```
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
cnstError _ = undefined

This function is also strict.
Example 2: null via equality
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)
nullViaEq (enumFromTo 0 2)
nullViaEq (enumFromTo 0 2) = enumFromTo 0 2 == []

Reduction steps change, but still independent of \( n \).
Still constant space (and time).
Equational reasoning

\[
\text{nullViaEq } (\text{enumFromTo } 0 \ 2) \\
= \text{enumFromTo } 0 \ 2 \ == \ []
\]
nullViaEq (enumFromTo 0 2)
= enumFromTo 0 2 == []
= (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
nullViaEq (enumFromTo 0 2)
= enumFromTo 0 2 == []
= (if 0 > 2 then [] else 0 : enumFromTo (0 + 1) 2) == []
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) == []
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) == []
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) == []
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
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

```plaintext

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)
```
This is where sharing comes into play:

```ml
let x = enumFromTo 0 2 in x == x
```

Linear time, but constant space.
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:

\[
\text{selfEqual (enumFromTo 0 2)} \\
= \text{let } x = \text{enumFromTo 0 2 in } x == x \\
= \text{let } x = 0 : \text{enumFromTo (0 + 1) 2 in } x == x
\]
Equational reasoning

This is where sharing comes into play:

```plaintext
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
```

Linear time, but constant space.
Equational reasoning

This is where sharing comes into play:

```plaintext
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
```

Linear time, but constant space.
Equational reasoning

This is where sharing comes into play:

\[
\text{selfEqual (enumFromTo 0 2)} \\
= \text{let } x = \text{enumFromTo 0 2 in } x == x \\
= \text{let } x = 0 : \text{enumFromTo (0 + 1) 2 in } x == x \\
= \text{let } x = 0 : x'; x' = \text{enumFromTo (0 + 1) 2 in } x == x \\
= \text{let } x = 0 : x'; x' = \text{enumFromTo (0 + 1) 2 in } 0 == 0 && x' == x'
\]
Equational reasoning

This is where sharing comes into play:

```ml
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'
```

Linear time, but constant space.
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'
```

Linear time, but constant space.
Equational reasoning

This is where sharing comes into play:

```haskell
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'
```

Linear time, but constant space.
Equational reasoning

This is where sharing comes into play:

```plaintext
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'
```

Linear time, but constant space.
Equational reasoning

This is where sharing comes into play:

\[
\text{selfEqual (enumFromTo 0 2)}
\]

\[
= \text{let } x = \text{enumFromTo 0 2 in } x == x \\
= \text{let } x = 0 : \text{enumFromTo (0 + 1) 2 in } x == x \\
= \text{let } x = 0 : x'; x' = \text{enumFromTo (0 + 1) 2 in } x == x \\
= \text{let } x = 0 : x'; x' = \text{enumFromTo (0 + 1) 2 in } 0 == 0 && x' == x' \\
= \text{let } x' = \text{enumFromTo (0 + 1) 2 in } 0 == 0 && x' == x' \\
= \text{let } x' = \text{enumFromTo (0 + 1) 2 in } \text{True && x' == x'} \\
= \text{let } x' = \text{enumFromTo (0 + 1) 2 in } x' == x'
\]
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.
A somewhat special case is sharing introduced at the top-level.

\[
\begin{align*}
\text{fib} & : \text{Int} \rightarrow \text{Int} \\
\text{fib} & \ 0 = 0 \\
\text{fib} & \ 1 = 1 \\
\text{fib} \ n & = \text{fib} \ (n - 1) + \text{fib} \ (n - 2) \\
\text{expensive} & : \text{Int} \\
\text{expensive} & = \text{fib} \ 32
\end{align*}
\]
A somewhat special case is sharing introduced at the top-level.

```haskell
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)**.
A somewhat special case is sharing introduced at the top-level.

\[
\begin{align*}
\text{fib} &: \text{Int} \rightarrow \text{Int} \\
\text{fib} \ 0 &= 0 \\
\text{fib} \ 1 &= 1 \\
\text{fib} \ n &= \text{fib} \ (n - 1) + \text{fib} \ (n - 2)
\end{align*}
\]

\[\text{expensive} :: \text{Int} \]

\[\text{expensive} = \text{fib} \ 32\]

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
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`

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

```haskell
null (map (\x -> x <= 10) (enumFromTo 0 2))
```
null (map (<= 10) (enumFromTo 0 2))
null (map (<= 10) (enumFromTo 0 2))
= null (map (<= 10) (0 : enumFromTo (0 + 1) 2))
null (map (<= 10) (enumFromTo 0 2))

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

\[
\begin{align*}
\text{null} & \left( \text{map} \ (\leq 10) \ (\text{enumFromTo} \ 0 \ 2) \right) \\
& = \text{null} \ (\text{map} \ (\leq 10) \ (0 : \text{enumFromTo} \ (0 + 1) \ 2)) \\
& = \text{null} \ ((0 \leq 10) : \text{map} \ (\leq 10) \ \text{enumFromTo} \ (0 + 1) \ 2) \\
& = \text{False}
\end{align*}
\]

Still **constant space** (and time).
Adding a different function

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

`reverse :: [a] -> [a]`

`reverse = reverseAcc []`

`reverseAcc :: [a] -> [a] -> [a]`

`reverseAcc acc [] = acc`

`reverseAcc acc (x : xs) = reverseAcc (x : acc) xs`
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 (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
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 : []) [])
= False

This operates in **linear** space (and time).
What is the key difference between \texttt{map} and \texttt{reverse}?

The function \texttt{map} is incremental, while \texttt{reverse} is not. More precisely:

- For \texttt{map}, we only need to evaluate the input list as far as we want to evaluate the output list.
- For \texttt{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.
Comparing `map` and `reverse`

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

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

Incrementality is not precisely defined, but I am calling functions incremental that can produce (parts of) their output without evaluating all of their input.
What is the key difference between \texttt{map} and \texttt{reverse}?

The function \texttt{map} is \textit{incremental}, while \texttt{reverse} is not.

More precisely:

- for \texttt{map}, we only need to evaluate the input list as far as we want to evaluate the output list.
- for \texttt{reverse}, even for just evaluating the result list to WHNF, we have to evaluate the entire spine of the input list.
What is the key difference between \texttt{map} and \texttt{reverse}?

The function \texttt{map} is \textit{incremental}, while \texttt{reverse} is not.

More precisely:

- for \texttt{map}, we only need to evaluate the input list as far as we want to evaluate the output list.
- for \texttt{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.
Which of the following functions are (or should be) incremental?

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

- map \( f \)
- reverse
- filter \( p \)
Which of the following functions are (or should be) incremental?

- map f
- reverse
- filter p
- length
Which of the following functions are (or should be) incremental?

- map \( f \)
- reverse
- filter \( p \)
- length
- sum
Which of the following functions are (or should be) incremental?

- map \( f \)
- reverse
- filter \( p \)
- length
- sum
- and
Which of the following functions are (or should be) incremental?

- map \( f \)
- reverse
- filter \( p \)
- length
- sum
- and
- take \( n \)
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**?
Let us just look at `length` itself:

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

What is the space behaviour?
A (naive) definition of `length` is bad:

```haskell
length :: [a] -> Int
length [] = 0
length ((_:xs)) = 1 + length xs
```
length (enumFromTo 0 2)
= length (0 : enumFromTo (0 + 1) 2)
= 1 + length (enumFromTo (0 + 1) 2)
Equational reasoning

\[
\begin{align*}
\text{length } & (\text{enumFromTo } 0 \ 2) \\
= & \text{length } (0 : \text{enumFromTo } (0 + 1) \ 2) \\
= & 1 + \text{length } (\text{enumFromTo } (0 + 1) \ 2) \\
= & 1 + \text{length } (1 : \text{enumFromTo } (1 + 1) \ 2) \\
= & 1 + (1 + (\text{length } (\text{enumFromTo } (1 + 1) \ 2)))
\end{align*}
\]
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

\[
\begin{align*}
\text{length} & \ (\text{enumFromTo} \ 0 \ 2) \\
& = \ \text{length} \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
& = 1 + \text{length} \ (\text{enumFromTo} \ (0 + 1) \ 2) \\
& = 1 + \text{length} \ (1 : \text{enumFromTo} \ (1 + 1) \ 2) \\
& = 1 + (1 + (\text{length} \ (\text{enumFromTo} \ (1 + 1) \ 2))) \\
& = \ldots \\
& = 1 + (1 + (1 + 0)) \\
& = \ldots \\
& = 3
\end{align*}
\]

Runs in \textbf{linear} space.
An accumulating definition of `length` is potentially not much better:

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

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

\[
\text{length} \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{lengthAcc} \ (1 + 0) \ (\text{enumFromTo} \ (0 + 1) \ 2)
\]
Equational reasoning

\[
\begin{align*}
\text{length} & \ (\text{enumFromTo} \ 0 \ 2) \\
& = \ \text{lengthAcc} \ 0 \ (\text{enumFromTo} \ 0 \ 2) \\
& = \ \text{lengthAcc} \ 0 \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
& = \ \text{lengthAcc} \ (1 + 0) \ (\text{enumFromTo} \ (0 + 1) \ 2) \\
& = \ \text{lengthAcc} \ (1 + 0) \ (1 : \text{enumFromTo} \ (1 + 1) \ 2) \\
& = \ \text{lengthAcc} \ (1 + (1 + 0)) \ (\text{enumFromTo} \ (1 + 1) \ 2)
\end{align*}
\]
Equational reasoning

\[
\text{length (enumFromTo 0 2)}
\]
\[=	ext{lengthAcc 0 (enumFromTo 0 2)}
\]
\[=	ext{lengthAcc 0 (0 : enumFromTo (0 + 1) 2)}
\]
\[=	ext{lengthAcc (1 + 0) (enumFromTo (0 + 1) 2)}
\]
\[=	ext{lengthAcc (1 + 0) (1 : enumFromTo (1 + 1) 2)}
\]
\[=	ext{lengthAcc (1 + (1 + 0)) (enumFromTo (1 + 1) 2)}
\]
\[...
\]
\[=	ext{lengthAcc (1 + (1 + (1 + 0))) []}
\]
\[= 1 + (1 + (1 + 0))
\]
\[= ...
\]
\[= 3
\]

Also runs in **linear** space.
We can fix the problem by artifically making \texttt{lengthAcc} more strict:

\begin{verbatim}
length :: [a] \rightarrow \text{Int}
length = \text{lengthAcc} \ 0

lengthAcc :: \text{Int} \rightarrow [a] \rightarrow \text{Int}
lengthAcc \! acc \ [\] = acc
lengthAcc \! acc \ (_ : xs) = lengthAcc (1 + acc) \ xs
\end{verbatim}
We can fix the problem by artifically making \(\text{lengthAcc}\) more strict:

\[
\begin{align*}
\text{length} & :: \ [a] \rightarrow \text{Int} \\
\text{length} & = \text{lengthAcc} \ 0 \\
\text{lengthAcc} & :: \ \text{Int} \rightarrow \ [a] \rightarrow \text{Int} \\
\text{lengthAcc} \ !\text{acc} \ [] & = \text{acc} \\
\text{lengthAcc} \ !\text{acc} \ (_{\text{\_ : xs}}) & = \text{lengthAcc} \ (1 + \text{acc}) \ \text{xs}
\end{align*}
\]

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

\[
\begin{align*}
\text{length} \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{lengthAcc} \ (1 + 0) \ (\text{enumFromTo} \ (0 + 1) \ 2)
\end{align*}
\]
Equational reasoning

\[
\begin{align*}
\text{length}\ (\text{enumFromTo}\ 0\ 2) \\
= &\ \text{lengthAcc}\ 0\ (\text{enumFromTo}\ 0\ 2) \\
= &\ \text{lengthAcc}\ 0\ (0 : \text{enumFromTo}\ (0 + 1)\ 2) \\
= &\ \text{lengthAcc}\ (1 + 0)\ (\text{enumFromTo}\ (0 + 1)\ 2)
\end{align*}
\]
Equational reasoning

\[
\text{length} \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (\text{enumFromTo} \ 0 \ 2) \\
= \text{lengthAcc} \ 0 \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{lengthAcc} \ (1 + 0) \ (\text{enumFromTo} \ (0 + 1) \ 2) \\
= \text{lengthAcc} \ 1 \ (\text{enumFromTo} \ (0 + 1) \ 2)
\]

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

\[
\begin{align*}
\text{length} \ (\text{enumFromTo} \ 0 \ 2) &= \text{lengthAcc} \ 0 \ (\text{enumFromTo} \ 0 \ 2) \\
&= \text{lengthAcc} \ 0 \ (0 : \text{enumFromTo} \ (0 + 1) \ 2) \\
&= \text{lengthAcc} \ (1 + 0) \ (\text{enumFromTo} \ (0 + 1) \ 2) \\
&= \text{lengthAcc} \ 1 \ (\text{enumFromTo} \ (0 + 1) \ 2) \\
&= \text{lengthAcc} \ 1 \ (1 : \text{enumFromTo} \ (1 + 1) \ 2) \\
&= \text{lengthAcc} \ 2 \ (2 : \text{enumFromTo} \ (2 + 1) \ 2) \\
&= \text{lengthAcc} \ 3 \ [] \\
&= 3
\end{align*}
\]

Now runs in **constant space** (but still linear time).
Aside: more on bang patterns

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

(Why?)
Historically, Haskell has had `seq` to control evaluation.

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

```haskell
seq :: a -> b -> b
seq !_ y = y
```
Historically, Haskell has had `seq` to control evaluation.

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

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

```haskell
lengthAcc :: Int -> [a] -> Int
lengthAcc acc [] = acc
lengthAcc acc (_ : xs) = seq acc (lengthAcc (1 + acc) xs)
```
Why not

```haskell
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 \(\text{id}\). It does not create any demand that does not already exist.
Why not

\[
\text{force} :: a \to a \\
\text{force} \ x = \text{seq} \ x \ x
\]

\[
\text{lengthAcc} :: \text{Int} \to [a] \to \text{Int} \\
\text{lengthAcc} \ acc \ [] = \text{acc} \\
\text{lengthAcc} \ acc \ (_ : \ xs) = \text{lengthAcc} \ (\text{force} \ (1 + \text{acc})) \ xs
\]

\text{force} is just \text{id}. It does not create any demand that does not already exist.
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`:

```haskell
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 the accumulator, and therefore will not be optimised to use a strict accumulator.
Returning to our initial example

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

example5a :: Int -> Bool
example5a n = nullViaLength [0 .. n]
```
Returning to our initial example

```haskell
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 ...
Another variant

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

Sharing can turn something that just looks unnecessarily inefficient into a space leak.
Example 6: unfair partitioning
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

\[
\begin{align*}
\text{partition} \ (\geq 0) \ (\text{enumFromTo} \ (0 .. 2)) &= \text{partition} \ (\geq 0) \ (0 : \text{enumFromTo} \ (0 + 1) 2) \\
&= \text{case} \ \text{partition} \ (\geq 0) \ (\text{enumFromTo} \ (0 + 1) 2) \ of \\
& \quad (ys, zs) \\
& \quad \mid (\geq 0) \ 0 \to (0 : ys, zs) \\
& \quad \mid \text{otherwise} \to (ys, 0 : zs)
\end{align*}
\]
Equational reasoning

\[
\begin{align*}
\text{partition } (\geq 0) \text{ (enumFromTo } (0 .. 2)) &= \text{partition } (\geq 0) \text{ (} 0 : \text{enumFromTo } (0 + 1) 2) \\
&= \textbf{case} \ \text{partition } (\geq 0) \text{ (enumFromTo } (0 + 1) 2) \ \textbf{of} \\
&\quad (ys, zs) \\
&\quad \mid (\geq 0) 0 \rightarrow (0 : ys, zs) \\
&\quad \mid \text{otherwise} \rightarrow (ys, 0 : zs) \\
&= \ldots \\
&= \textbf{case} \ \textbf{(case} \ \text{partition } (\geq 0) \text{ (enumFromTo } (1 + 1) 2) \ \textbf{of} \\
&\quad (ys', zs') \\
&\quad \mid (\geq 0) 1 \rightarrow (1 : ys, zs) \\
&\quad \mid \text{otherwise} \rightarrow (ys, 1 : zs) \\
&\quad ) \ \textbf{of} \\
&\quad (ys, zs) \\
&\quad \mid (\geq 0) 0 \rightarrow (0 : ys, zs) \\
&\quad \mid \text{otherwise} \rightarrow (ys, 0 : zs)
\end{align*}
\]

Oh no ...
Irrefutable pattern matches

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

```haskell
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 \textit{partition}:

\begin{verbatim}
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)
\end{verbatim}
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

\[
\text{partition}( \geq 0)(\text{enumFromTo}(0..2)) = \text{partition}( \geq 0)(0:\text{enumFromTo}(0+1)2) = \text{let } r = \text{partition}( \geq 0)(\text{enumFromTo}(0+1)2) \text{ in if } (\geq 0)0 \text{ then } (0:\text{fst } r, \text{snd } r) \text{ else } (\text{fst } r, 0:\text{snd } r)
\]
Equational reasoning

\[
\text{partition } (\geq 0) (\text{enumFromTo } (0 \text{ .. } 2)) \\
= \text{partition } (\geq 0) (0 : \text{enumFromTo } (0 + 1) 2) \\
= \text{let } r = \text{partition } (\geq 0) (\text{enumFromTo } (0 + 1) 2) \\
\quad \text{in } \text{if } (\geq 0) 0 \\
\quad \quad \text{then } (0 : \text{fst } r, \text{snd } r) \\
\quad \quad \text{else } (\text{fst } r, 0 : \text{snd } r) \\
= \text{let } r = \text{partition } (\geq 0) (\text{enumFromTo } (0 + 1) 2) \\
\quad \text{in } (0 : \text{fst } r, \text{snd } r)
\]

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 \( \text{fst} \, r \):

```haskell
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 \( \text{fst } r \):

\[
\begin{align*}
\text{let } r &= \text{partition } (\geq 0) \text{ (enumFromTo } (0 + 1) 2) \\
\text{in } (0 : \text{fst } r, \text{snd } r) \\
= \text{let } r &= \text{let } r' &= \text{partition } (\geq 0) \text{ (enumFromTo } (1 + 1) 2) \\
\text{in } (1 : \text{fst } r', \text{snd } r') \\
\text{in } (0 : \text{fst } r, \text{snd } r)
\end{align*}
\]

Isn’t there still a problem here?
Let’s assume we place more demand on the first component of the result pair, i.e., on `fst r`:

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

```haskell
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?
The **garbage collector** will reduce **selector thunks** if possible, even if there’s no explicit demand on them.
example6 :: Int -> (Int, Int)
example6 n =
  case partition (>= 0) [0 .. n] of
    (xs, ys) -> (sum xs, sum ys)
Equational reasoning

\[
\text{case partition (} \geq 0 \text{) (enumFromTo 0 2) of }
\]
\[
(x, y) \rightarrow (\text{sum } x, \text{sum } y)
\]
Equational reasoning

\[
\text{case partition } (\geq 0) (\text{enumFromTo } 0 2) \text{ of } \\
\hspace{1cm} (xs, ys) \rightarrow (\text{sum } xs, \text{sum } ys) \\
\]

\[
= \text{case } \text{let } r = \text{partition } (\geq 0) (\text{enumFromTo } (0 + 1) 2) \text{ in } (0 : \text{fst } r, \text{snd } r) \text{ of } \\
\hspace{1cm} (xs, ys) \rightarrow (\text{sum } xs, \text{sum } ys)
\]
Equational reasoning

\[
\text{case } \text{partition } (\geq 0) \ (\text{enumFromTo } 0 \ 2) \ of \\
\hspace{1em} (xs, ys) \rightarrow (\text{sum } xs, \text{sum } ys) \\
= \text{case } (\text{let } r = \text{partition } (\geq 0) \ (\text{enumFromTo } (0 + 1) \ 2) \\
\hspace{2em} \text{in } (0 : \text{fst } r, \text{snd } r)) \ of \\
\hspace{2em} (xs, ys) \rightarrow (\text{sum } xs, \text{sum } ys) \\
= \text{let } r = \text{partition } (\geq 0) \ (\text{enumFromTo } (0 + 1) \ 2) \\
\hspace{2em} \text{in } (\text{sum } (0 : \text{fst } r), \text{sum } (\text{snd } r))
\]

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

```haskell
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

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

While we are evaluating the first component of the pair, the second component grows larger ...
Equational reasoning

\[
\text{case partition even (enumFromTo 0 2) of}
\]
\[
(x, y) \rightarrow (\text{sum } x, \text{sum } y)
\]
\[
= \text{case (let } r = \text{partition even (enumFromTo (0 + 1) 2)}
\]
\[
\text{in } (0 : \text{fst } r, \text{snd } r)) \text{ of}
\]
\[
(x, y) \rightarrow (\text{sum } x, \text{sum } y)
\]
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 ...
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 
((\text{Int}, \text{Int}))

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

```haskell
partitionEvenSums :: \text{Int} \to (\text{Int}, \text{Int})
partitionEvenSums = partitionEvenSumsAcc (0, 0)

partitionEvenSumsAcc :: (\text{Int}, \text{Int}) \to \text{Int} \to (\text{Int}, \text{Int})
partitionEvenSumsAcc (!x, !y) [] = (x, y)
p
```
Revisiting the example

\[\text{example7b} :: \text{Int} \rightarrow (\text{Int, Int})\]
\[\text{example7b} \ n = \text{partitionEvenSums} \ [0 \ldots n]\]

This works in \textit{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.
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
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
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

\[
\begin{align*}
\text{traverse pure (enumFromTo 0 2)} &= \text{traverse pure (0 : enumFromTo (0 + 1) 2)} \\
&= \text{pure (:)} <\times> \text{pure 0} \\
&<\times> \text{traverse pure (enumFromTo (0 + 1) 2)}
\end{align*}
\]
Equational reasoning

\[
\begin{align*}
\text{traverse pure (enumFromTo 0 2)} \\
&= \text{traverse pure (0 : enumFromTo (0 + 1) 2)} \\
&= \text{pure (:) <<< pure 0} \\
&\quad <<< \text{traverse pure (enumFromTo (0 + 1) 2)} \\
&= \text{Identity (runIdentity (pure (:)) <<< runIdentity (pure 0))} \\
&\quad <<< \text{traverse pure (enumFromTo (0 + 1) 2)}
\end{align*}
\]

This looks fine (and it is). Runs in constant space.
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

\[
\begin{align*}
\text{traverse pure (enumFromTo 0 2)} & = \text{traverse pure (0 : enumFromTo (0 + 1) 2)} \\
& = \text{pure (:) <> pure 0} \\
& \quad <> \text{traverse pure (enumFromTo (0 + 1) 2)} \\
& = \text{Identity (runIdentity (pure ())) <> runIdentity (pure 0))} \\
& \quad <> \text{traverse pure (enumFromTo (0 + 1) 2)} \\
& = \text{Identity ((':') 0)} \\
& \quad <> \text{traverse pure (enumFromTo (0 + 1) 2)} \\
& = \text{Identity} \\
& \quad (0 : \text{runIdentity (traverse pure (enumFromTo (0 + 1) 2))))}
\end{align*}
\]

This looks fine (and it is).

Runs in \textit{constant space}. 
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