Lecture 5: Lazy Evaluation and Infinite Data Structures

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

October 3, 2017

How does Haskell evaluate a program?

Since there are no side-effects in Haskell programs, the evaluation order of expressions is not specified by the order of the expressions:

$$
\begin{aligned}
&\mathit{root}\ a\ b\ c = (-b + \mathit{sd})\ /\ (2 * a)\\
&\quad \textbf{where}\\
&\qquad \mathit{sd} = \mathit{sqrt}\ d\\
&\qquad d = b * b - 4 * a * c
\end{aligned}
$$

This definition is perfectly valid, even though the definition of d is written after sd.

$> \mathit{root}\ 1\ 0\ (-2)$
1.4142135623730951

Lazy evaluation

- The purity of Haskell functions (= absense of side-effects) gives more freedom to the compiler, when it decides when to evaluate each expression:
- The idea is to wait with evaluating an expression until it is *needed*.
- This concept is also known as **Lazy evaluation**.
- **Note:** Not all functional programming languages use Lazy evaluation as its evaluation model. This is just a design choice in Haskell.

Advantages of Lazy evaluation

Syddansk Universitet
University of Southern Denmark

1. Avoids doing **unnecessary evaluation**
2. Allows programs to be **more modular**
3. Allows us to program with **infinite lists**

Example 1

$$a :: Bool$$
$$a = \neg\, a$$

What happens if we evaluate a?

Example 1

$$a :: Bool$$
$$a = \neg\, a$$

What happens if we evaluate a?

$$b :: Bool$$
$$b = False$$
$$c :: Bool$$
$$c = b \wedge a$$

What happens if we evaluate c?

Example 1

$$a :: Bool$$
$$a = \neg\, a$$

What happens if we evaluate a?

$$b :: Bool$$
$$b = False$$
$$c :: Bool$$
$$c = b \wedge a$$

What happens if we evaluate c?

$$c' :: Bool$$
$$c' = a \wedge b$$

What about c'?

Example 1

$$a :: Bool$$
$$a = \neg\, a$$

What happens if we evaluate a?

$$b :: Bool$$
$$b = False$$
$$c :: Bool$$
$$c = b \wedge a$$

What happens if we evaluate c?

$$c' :: Bool$$
$$c' = a \wedge b$$

What about c'?

In other languages this behavior is called **short circuiting**, and is only possible in those languages because $(\wedge)$ and $(\vee)$ are built in to the language.

Note that $(\wedge)$ and booleans *Bool* are *not* primitives in the Haskell language. They are just an ordinary function and an ordinary algebraic data type!

Example 2

In Haskell we can define our own "if"-function by

$$
\begin{aligned}
&\mathit{myIf} :: \mathit{Bool} \to a \to a \to a \\
&\mathit{myIf}\ \mathit{True}\ v\ \_\ = v \\
&\mathit{myIf}\ \mathit{False}\ \_\ v = v
\end{aligned}
$$

What happens when we evaluate

$$\mathit{myIf}\ \mathit{True}\ \mathit{expr1}\ \mathit{expr2}$$

?

Example 2

SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

In Haskell we can define our own “if”-function by

$$
\begin{aligned}
&\mathit{myIf} :: \mathit{Bool} \rightarrow a \rightarrow a \rightarrow a\\
&\mathit{myIf}\ \mathit{True}\ v\ \_\ = v\\
&\mathit{myIf}\ \mathit{False}\ \_\ v = v
\end{aligned}
$$

What happens when we evaluate

$$\mathit{myIf}\ \mathit{True}\ \mathit{expr1}\ \mathit{expr2}$$

? We get the result *expr1*, and *expr2* never gets evaluated. **This is clearly different from the way other languages like Java and Python would evaluate a similar function.**

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

Examples:

$(\lambda x \to x * 2)$

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

Examples:

$(\lambda x \rightarrow x * 2)$	Not reducible

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called "reducible expressions".

Examples:

$(\lambda x \to x * 2)$	Not reducible
$(\lambda x \to x * 2)\ 5$	

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called "reducible expressions".

Examples:

$(\lambda x \rightarrow x * 2)$	Not reducible
$(\lambda x \rightarrow x * 2)\ 5$	Reducible to $5 * 2$, which is reducible to 10

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

Examples:

$(\lambda x \to x * 2)$	Not reducible
$(\lambda x \to x * 2)\ 5$	Reducible to $5 * 2$, which is reducible to 10
$(\lambda x\ y \to 13 * x + y)\ 2$	

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called "reducible expressions".

Examples:

$(\lambda x \rightarrow x * 2)$	Not reducible
$(\lambda x \rightarrow x * 2)\ 5$	Reducible to $5 * 2$, which is reducible to 10
$(\lambda x\ y \rightarrow 13 * x + y)\ 2$	Reducible to $(\lambda y \rightarrow 13 * 2 + y)$.
	Then subexpression $13 * 2$ is reducible.

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called “reducible expressions”.

Examples:

$(\lambda x \to x * 2)$	Not reducible
$(\lambda x \to x * 2)\ 5$	Reducible to $5 * 2$, which is reducible to 10
$(\lambda x\ y \to 13 * x + y)\ 2$	Reducible to $(\lambda y \to 13 * 2 + y)$.
	Then subexpression $13 * 2$ is reducible.
$(7, 5 + 3)$	

Reducible expressions

- We know that Haskell evaluates expressions by succesively applying definitons, until no more simplifications are possible.
- Expressions which can be simplified are called "reducible expressions".

Examples:

$(\lambda x \to x * 2)$	Not reducible
$(\lambda x \to x * 2)\ 5$	Reducible to $5 * 2$, which is reducible to 10
$(\lambda x\ y \to 13 * x + y)\ 2$	Reducible to $(\lambda y \to 13 * 2 + y)$.
	Then subexpression $13 * 2$ is reducible.
$(7, 5 + 3)$	Not reducible, but subexpression $5 + 3$ is.

Function application is also known as a β-reduction.

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2)$$

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

- **Outermost reduction:** [call-by-need]

$$sqr\ (4+2)$$

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

- **Outermost reduction:** [call-by-need]

$$sqr\ (4+2) \equiv (4+2) * (4+2) \equiv 6 * (4+2) \equiv 6 * 6 \equiv 36$$

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

- **Outermost reduction:** [call-by-need]

$$sqr\ (4+2) \equiv (4+2) * (4+2) \equiv 6 * (4+2) \equiv 6 * 6 \equiv 36$$

- **Outermost reduction with sharing (= Graph reduction)**

$$sqr\ (4+2)$$

Reduction strategies for function application

Consider the function $sqr\ x = x * x$ and the expression $sqr\ (4+2)$

- **Innermost reduction:** [call-by-value]

$$sqr\ (4+2) \equiv sqr\ 6 \equiv 6 * 6 \equiv 36$$

- **Outermost reduction:** [call-by-need]

$$sqr\ (4+2) \equiv (4+2) * (4+2) \equiv 6 * (4+2) \equiv 6 * 6 \equiv 36$$

- **Outermost reduction with sharing (= Graph reduction)**

$$sqr\ (4+2) \equiv \textbf{let}\ x = 4+2\ \textbf{in}\ x * x \equiv 6 * 6 \equiv 36$$

The general evaluation rule of Haskell can be described as

Leftmost outermost reduction with sharing

One more example

$$
\begin{aligned}
&\textit{ackermann}\ m\ n\\
&\quad \mid m \equiv 0 \qquad\qquad\quad\ = n + 1\\
&\quad \mid m > 0 \wedge n \equiv 0 = \textit{ackermann}\ (m-1)\ 1\\
&\quad \mid m > 0 \wedge n > 0\ = \textit{ackermann}\ (m-1)\ (\textit{ackermann}\ m\ (n-1))
\end{aligned}
$$

How does Haskell evaluate?

$$\textit{const}\ 5\ (\textit{ackermann}\ 4\ 2)$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$3 * 4 + 3 * 4$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$
\begin{aligned}
& 3 * 4 + 3 * 4 \\
\to\ & \{\text{first operand not done}\} \\
& \quad 3 * 4
\end{aligned}
$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$
\begin{array}{ll}
 & 3 * 4 + 3 * 4 \\
\rightarrow & \{\text{first operand not done}\} \\
 & \quad 3 * 4 \\
 & \quad \rightarrow \{\text{both operands done, arithmetic}\} \\
 & \quad 12 \\
\ldots & 12 + 3 * 4
\end{array}
$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$
\begin{array}{l}
\quad 3 * 4 + 3 * 4 \\
\to \{\text{first operand not done}\} \\
\qquad 3 * 4 \\
\quad\; \to \{\text{both operands done, arithmetic}\} \\
\qquad 12 \\
\ldots\; 12 + 3 * 4 \\
\to \{\text{second operand not done}\} \\
\qquad 3 * 4
\end{array}
$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$
\begin{array}{ll}
 & 3*4+3*4 \\
\rightarrow & \{\text{first operand not done}\} \\
 & \quad 3*4 \\
 & \quad \rightarrow \{\text{both operands done, arithmetic}\} \\
 & \quad 12 \\
\ldots & 12+3*4 \\
\rightarrow & \{\text{second operand not done}\} \\
 & \quad 3*4 \\
 & \quad \rightarrow \{\text{both operands done, arithmetic}\} \\
 & \quad 12 \\
\ldots & 12+12
\end{array}
$$

Primitive arithmetic operations $(+), (-), (*), (/)$

Cannot be evaluated in outermost manner:

$$
\begin{array}{ll}
 & 3 * 4 + 3 * 4 \\
\rightarrow & \{\text{first operand not done}\} \\
 & \quad 3 * 4 \\
 & \quad \rightarrow \{\text{both operands done, arithmetic}\} \\
 & \quad 12 \\
\ldots & 12 + 3 * 4 \\
\rightarrow & \{\text{second operand not done}\} \\
 & \quad 3 * 4 \\
 & \quad \rightarrow \{\text{both operands done, arithmetic}\} \\
 & \quad 12 \\
\ldots & 12 + 12 \\
\rightarrow & \{\text{both operands done, arithmetic}\} \\
 & 24
\end{array}
$$

We say that the functions are *strict* in both arguments.

Another example

Consider again the function

$$sqr\ x = x * x$$

How does Haskell evaluate

$$sqr\ (sqr\ 2)$$

Another example

Consider again the function

$$sqr\ x = x * x$$

How does Haskell evaluate

$$\begin{aligned}
sqr\ (sqr\ 2) &\equiv \mathbf{let}\ x = sqr\ 2\ \mathbf{in}\ x * x && \text{-- (apply outer } sqr\text{)} \\
&\equiv \mathbf{let}\ x = 2 * 2\ \mathbf{in}\ x * x && \text{-- (apply inner } sqr\text{)} \\
&\equiv \mathbf{let}\ x = 4\ \mathbf{in}\ x * x && \text{-- (apply inner } (*)\text{)} \\
&\equiv 16 && \text{-- (apply outer } (*)\text{)}
\end{aligned}$$

Innermost and outermost reductions

$$
\begin{tikzcd}
 & \boxed{(\lambda x \to x)\ ((\lambda y \to y)\ z)} \arrow[dl] \arrow[dr] & \\
\boxed{(\lambda x \to x)\ z} \arrow[dr] & & \boxed{(\lambda y \to y)\ z} \arrow[dl] \\
 & \boxed{z} &
\end{tikzcd}
$$

Which is innermost and which is outermost?

$$\begin{tikzcd} & \boxed{(\lambda x \to x)\ ((\lambda y \to y)\ z)} \arrow[dl, "\text{innermost}"'] \arrow[dr, "\text{outermost}"] & \\ \boxed{(\lambda x \to x)\ z} \arrow[dr] & & \boxed{(\lambda y \to y)\ z} \arrow[dl] \\ & \boxed{z} & \end{tikzcd}$$

- when an expression contains no reducible subexpression, it is in **normal form**

$$
\begin{tikzcd}
& (\lambda x \to x)\ ((\lambda y \to y)\ z) \arrow[dl, "\text{innermost}"'] \arrow[dr, "\text{outermost}"] & \\
(\lambda x \to x)\ z \arrow[dr] & & (\lambda y \to y)\ z \arrow[dl] \\
& z &
\end{tikzcd}
$$

- when an expression contains no reducible subexpression, it is in **normal form**
- some expressions might not have a normal form

$$\begin{tikzcd} & \boxed{(\lambda x \to x)\ ((\lambda y \to y)\ z)} \arrow[dl, "\text{innermost}"'] \arrow[dr, "\text{outermost}"] & \\ \boxed{(\lambda x \to x)\ z} \arrow[dr] & & \boxed{(\lambda y \to y)\ z} \arrow[dl] \\ & \boxed{z} & \end{tikzcd}$$

- when an expression contains no reducible subexpression, it is in **normal form**
- some expressions might not have a normal form
- an expression has at most one normal form

$$
\begin{tikzcd}[column sep=small]
 & (\lambda x \to x)\ ((\lambda y \to y)\ z) \arrow[dl, "\text{innermost}"'] \arrow[dr, "\text{outermost}"] & \\
(\lambda x \to x)\ z \arrow[dr] & & (\lambda y \to y)\ z \arrow[dl] \\
 & z &
\end{tikzcd}
$$

- when an expression contains no reducible subexpression, it is in **normal form**
- some expressions might not have a normal form
- an expression has at most one normal form
- **outermost reduction** always reaches the normal form if there exists a reduction strategy that reaches the normal form

$$
\begin{tikzcd}
 & (\lambda x \to x)\ ((\lambda y \to y)\ z) \arrow[dl, "\text{innermost}"'] \arrow[dr, "\text{outermost}"] & \\
(\lambda x \to x)\ z \arrow[dr] & & (\lambda y \to y)\ z \arrow[dl] \\
 & z &
\end{tikzcd}
$$

- when an expression contains no reducible subexpression, it is in **normal form**
- some expressions might not have a normal form
- an expression has at most one normal form
- **outermost reduction** always reaches the normal form if there exists a reduction strategy that reaches the normal form
- **outermost reduction with sharing** uses at most as many reduction steps as innermost reduction

Function application summarized

The evaluation rule for function application

1. Evaluate the function until it is a lambda term
2. Then plug in the arguments.
3. Multiple occurences of a formal argument must all point to the same actual argument (sharing)

Example (function which is not a lambda term)

$$(\mathbf{if}\ \mathit{True}\ \mathbf{then}\ (\lambda c \to (c, 0))\ \mathbf{else}\ (\lambda c \to (c, 1)))\ \mathit{blah}$$

Example (function which is not a lambda term)

$$
\begin{aligned}
&(\textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1)))\ \mathit{blah} \\
\to\ &\{\text{need function}\} \\
&\qquad \textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1))
\end{aligned}
$$

Example (function which is not a lambda term)

$(\textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1)))\ \mathit{blah}$

$\to$ {need function}

$\textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1))$

$\to$ {if-then-else}

$(\lambda c \to (c, 0))$

... $(\lambda c \to (c, 0))\ \mathit{blah}$

Example (function which is not a lambda term)

$$(\textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1)))\ \mathit{blah}$$

$\to$ {need function}

$$\textbf{if } \mathit{True} \textbf{ then } (\lambda c \to (c, 0)) \textbf{ else } (\lambda c \to (c, 1))$$

$\to$ {if-then-else}

$$(\lambda c \to (c, 0))$$

... $(\lambda c \to (c, 0))\ \mathit{blah}$

$\to$ {function application}

$$(\mathit{blah}, 0)$$

Example (function with multiple arguments)

$$(\lambda c \ \_ \to c)\ \mathit{True}\ (3 * 4 + 3 * 4)$$

Example (function with multiple arguments)

$$
\begin{aligned}
& (\lambda c\ \_ \to c)\ \mathit{True}\ (3 * 4 + 3 * 4) \\
\to\ & \{\text{need function}\} \\
& \quad (\lambda c\ \_ \to c)\ \mathit{True}
\end{aligned}
$$

Example (function with multiple arguments)

$$
\begin{aligned}
& (\lambda c\ \_ \rightarrow c)\ \mathit{True}\ (3 * 4 + 3 * 4) \\
\rightarrow\ & \{\text{need function}\} \\
& \qquad (\lambda c\ \_ \rightarrow c)\ \mathit{True} \\
& \quad \rightarrow \{\text{function application}\} \\
& \qquad (\backslash\_ \rightarrow \mathit{True}) \\
\ldots\ & (\backslash\_ \rightarrow \mathit{True})\ (3 * 4 + 3 * 4)
\end{aligned}
$$

Example (function with multiple arguments)

$$
\begin{aligned}
& (\lambda c\ \_ \to c)\ \mathit{True}\ (3 * 4 + 3 * 4) \\
\to\ & \{\text{need function}\} \\
& \qquad (\lambda c\ \_ \to c)\ \mathit{True} \\
& \quad \to \{\text{function application}\} \\
& \qquad (\backslash\_ \to \mathit{True}) \\
\ldots\ & (\backslash\_ \to \mathit{True})\ (3 * 4 + 3 * 4) \\
\to\ & \{\text{function application}\} \\
& \mathit{True}
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to x)\ \mathit{True} &&= \\
&(\lambda x \to x)\ \bot &&= \\
&(\lambda x \to ())\ \bot &&= \\
&(\lambda x \to \bot)\ () &&= \\
&(\lambda x\, f \to f\ x)\ \bot &&= \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) &&=
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to x)\ \mathit{True} &&= \mathit{True} \\
&(\lambda x \to x)\ \bot &&= \\
&(\lambda x \to ())\ \bot &&= \\
&(\lambda x \to \bot)\ () &&= \\
&(\lambda x\, f \to f\ x)\ \bot &&= \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) &&=
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \rightarrow x)\ \mathit{True} && = \mathit{True} \\
&(\lambda x \rightarrow x)\ \bot && = \bot \\
&(\lambda x \rightarrow ())\ \bot && = \\
&(\lambda x \rightarrow \bot)\ () && = \\
&(\lambda x\, f \rightarrow f\ x)\ \bot && = \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) && =
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to x)\ \mathit{True} &&= \mathit{True} \\
&(\lambda x \to x)\ \bot &&= \bot \\
&(\lambda x \to ())\ \bot &&= () \\
&(\lambda x \to \bot)\ () &&= \\
&(\lambda x\, f \to f\ x)\ \bot &&= \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) &&=
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to x)\ \mathit{True} && = \mathit{True} \\
&(\lambda x \to x)\ \bot && = \bot \\
&(\lambda x \to ())\ \bot && = () \\
&(\lambda x \to \bot)\ () && = \bot \\
&(\lambda x\, f \to f\ x)\ \bot && = \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) && =
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \rightarrow x)\ \mathit{True} && = \mathit{True} \\
&(\lambda x \rightarrow x)\ \bot && = \bot \\
&(\lambda x \rightarrow ())\ \bot && = () \\
&(\lambda x \rightarrow \bot)\ () && = \bot \\
&(\lambda x\, f \rightarrow f\ x)\ \bot && = \lambda f \rightarrow f\ \bot \\
&\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) && =
\end{aligned}
$$

Quiz time!

What does the following expressions evaluate to?

$$
\begin{aligned}
(\lambda x \to x)\ \mathit{True} &= \mathit{True} \\
(\lambda x \to x)\ \bot &= \bot \\
(\lambda x \to ())\ \bot &= () \\
(\lambda x \to \bot)\ () &= \bot \\
(\lambda x\, f \to f\ x)\ \bot &= \lambda f \to f\ \bot \\
\mathit{length}\ (\mathit{map}\ \bot\ [1,2]) &= 2
\end{aligned}
$$

Pattern matching

Rule: For each pattern (in order), evaluate the expression "as much as needed" to be able to verify or refute a pattern.

Pattern matching

Rule: For each pattern (in order), evaluate the expression "as much as needed" to be able to verify or refute a pattern.

$$\textbf{case}\ \mathit{const}\ (\mathit{Just}\ a)\ b\ \textbf{of}\ \{\mathit{Just}\ \_ \rightarrow \mathit{True}; \mathit{Nothing} \rightarrow \mathit{False}\}$$
$$\rightarrow \{\text{evaluate argument for pattern Just \_}\}$$
$$\mathit{const}\ (\mathit{Just}\ a)\ b$$

Pattern matching

Rule: For each pattern (in order), evaluate the expression "as much as needed" to be able to verify or refute a pattern.

$$
\begin{aligned}
& \textbf{case}\ \mathit{const}\ (\mathit{Just}\ a)\ b\ \textbf{of}\ \{\mathit{Just}\ \_ \rightarrow \mathit{True}; \mathit{Nothing} \rightarrow \mathit{False}\} \\
\rightarrow\ & \{\text{evaluate argument for pattern Just \_ }\} \\
& \quad \mathit{const}\ (\mathit{Just}\ a)\ b \\
\rightarrow\ & \quad \{\text{function application}\} \\
& \quad \mathit{Just}\ a \\
\ldots\ & \textbf{case}\ \mathit{Just}\ a\ \textbf{of}\ \{\mathit{Just}\ \_ \rightarrow \mathit{True}; \mathit{Nothing} \rightarrow \mathit{False}\}
\end{aligned}
$$

Pattern matching

Rule: For each pattern (in order), evaluate the expression "as much as needed" to be able to verify or refute a pattern.

$$
\begin{array}{ll}
 & \textbf{case}\ \mathit{const}\ (\mathit{Just}\ a)\ b\ \textbf{of}\ \{\mathit{Just}\ \_ \rightarrow \mathit{True}; \mathit{Nothing} \rightarrow \mathit{False}\} \\
\rightarrow & \{\text{evaluate argument for pattern Just \_ }\} \\
 & \quad \mathit{const}\ (\mathit{Just}\ a)\ b \\
\rightarrow & \quad \{\text{function application}\} \\
 & \quad \mathit{Just}\ a \\
\ldots & \textbf{case}\ \mathit{Just}\ a\ \textbf{of}\ \{\mathit{Just}\ \_ \rightarrow \mathit{True}; \mathit{Nothing} \rightarrow \mathit{False}\} \\
\rightarrow & \{\text{match pattern}\} \\
 & \mathit{True}
\end{array}
$$

Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

An expression is in **Weak Head Normal Form (WHNF)**, if it is either:

Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

An expression is in **Weak Head Normal Form (WHNF)**, if it is either:

- a constructor (eventually applied to arguments) like *True*, $Just\ (square\ 42)$ or $(:)\ 1\ [\,]$

Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

An expression is in **Weak Head Normal Form (WHNF)**, if it is either:

- a constructor (eventually applied to arguments) like *True*, $\mathit{Just}\ (\mathit{square}\ 42)$ or $(:)\ 1\ []$
- a built-in function applied to too few arguments (perhaps none) like $(+)\ 2$ or *sqrt*.

Pattern matching summarized

- Reduce the expression to be pattern matched to Weak Head Normal Form, and check for constructor match with outermost constructor in pattern.
- Repeat for corresponding subpatterns and subexpressions.

An expression is in **Weak Head Normal Form (WHNF)**, if it is either:

- a constructor (eventually applied to arguments) like *True*, $\mathit{Just}\ (\mathit{square}\ 42)$ or $(:)\ 1\ []$
- a built-in function applied to too few arguments (perhaps none) like $(+)\ 2$ or *sqrt*.
- or a lambda abstraction $\lambda x \rightarrow \mathit{expression}$.

Ex 1/4: Pattern matching (Level 1) - Built-in primitive

$$\textbf{case}\ 3 * 4\ \textbf{of}\ \{0 \rightarrow \mathit{True};\ \_ \rightarrow \mathit{False}\}$$

Ex 1/4: Pattern matching (Level 1) - Built-in primitive

$$\begin{aligned} & \textbf{case } 3 * 4 \textbf{ of } \{0 \to \mathit{True}; \_ \to \mathit{False}\} \\ \to\ & \{\text{evaluate argument for pattern } 0\} \\ & \quad 3 * 4 \end{aligned}$$

Ex 1/4: Pattern matching (Level 1) - Built-in primitive

$$\begin{aligned}
& \textbf{case } 3 * 4 \textbf{ of } \{0 \rightarrow \mathit{True}; _ \rightarrow \mathit{False}\} \\
\rightarrow\ & \{\text{evaluate argument for pattern } 0\} \\
& \quad 3 * 4 \\
\rightarrow & \quad \{\text{arithmetic}\} \\
& \quad 12 \\
\ldots\ & \textbf{case } 12 \textbf{ of } \{0 \rightarrow \mathit{True}; _ \rightarrow \mathit{False}\}
\end{aligned}$$

Ex 1/4: Pattern matching (Level 1) - Built-in primitive

$$
\begin{array}{ll}
 & \textbf{case}\ 3 * 4\ \textbf{of}\ \{0 \to \mathit{True};\ \_ \to \mathit{False}\} \\
\to & \{\text{evaluate argument for pattern } 0\} \\
 & \quad 3 * 4 \\
\to & \quad \{\text{arithmetic}\} \\
 & \quad 12 \\
\ldots & \textbf{case}\ 12\ \textbf{of}\ \{0 \to \mathit{True};\ \_ \to \mathit{False}\} \\
\to & \{\text{match pattern}\} \\
 & \mathit{False}
\end{array}
$$

Ex 2/4: Pattern matching (Level 0) - Variable or wildcard

$$\textbf{case}\ 3 * 4\ \textbf{of}\ \{x \rightarrow \mathit{True}\}$$

Ex 2/4: Pattern matching (Level 0) - Variable or wildcard

$$
\begin{array}{ll}
& \textbf{case}\ 3 * 4\ \textbf{of}\ \{x \rightarrow \mathit{True}\} \\
\rightarrow & \{\text{match pattern}\} \\
& \mathit{True}
\end{array}
$$

Ex 3/4: Pattern matching (Level 1) - Variable in pattern

$$\textbf{case}\ \mathit{Just}\ (3 * 4)\ \textbf{of}\ \{\mathit{Just}\ x \rightarrow (x, x); \mathit{Nothing} \rightarrow (0, 0)\}$$

Ex 3/4: Pattern matching (Level 1) - Variable in pattern

$$
\begin{aligned}
&\mathbf{case}\ \mathit{Just}\ (3*4)\ \mathbf{of}\ \{\mathit{Just}\ x \rightarrow (x,x); \mathit{Nothing} \rightarrow (0,0)\}\\
\rightarrow\ &\{\text{match pattern}\}\\
&\mathbf{let}\ x = 3*4\ \mathbf{in}\ (x,x)
\end{aligned}
$$

Ex 4/4: Pattern matching in function definitions

$$\begin{aligned}
&cond :: Bool \rightarrow a \rightarrow a \rightarrow a\\
&cond\ True\ x\ y\ \ = x\\
&cond\ False\ x\ y = y
\end{aligned}$$

Evaluate

$$cond\ (4 < 2)\ (5 * 17)\ (2 * 3)$$

Ex 4/4: Pattern matching in function definitions

$$
\begin{aligned}
&\mathit{cond} :: \mathit{Bool} \to a \to a \to a \\
&\mathit{cond}\ \mathit{True}\ x\ y\ = x \\
&\mathit{cond}\ \mathit{False}\ x\ y = y
\end{aligned}
$$

Evaluate

$$
\begin{aligned}
&\quad \mathit{cond}\ (4 < 2)\ (5 * 17)\ (2 * 3) \\
&\to \{\text{check if expression matches True}\} \\
&\qquad 4 < 2
\end{aligned}
$$

Ex 4/4: Pattern matching in function definitions

Syddansk Universitet
University of Southern Denmark

$$
\begin{aligned}
&\mathit{cond} :: \mathit{Bool} \to a \to a \to a \\
&\mathit{cond}\ \mathit{True}\ x\ y\ \ = x \\
&\mathit{cond}\ \mathit{False}\ x\ y = y
\end{aligned}
$$

Evaluate

$$
\begin{aligned}
&\quad \mathit{cond}\ (4 < 2)\ (5 * 17)\ (2 * 3) \\
\to\ &\{\text{check if expression matches True}\} \\
&\qquad 4 < 2 \\
\to\ &\qquad \{\text{primitive comparison}\} \\
&\qquad \mathit{False} \\
\ldots\ &\mathit{cond}\ \mathit{False}\ (5 * 7)\ (2 * 3)
\end{aligned}
$$

Ex 4/4: Pattern matching in function definitions

$$
\begin{aligned}
&\mathit{cond} :: \mathit{Bool} \rightarrow a \rightarrow a \rightarrow a\\
&\mathit{cond}\ \mathit{True}\ x\ y\ \ = x\\
&\mathit{cond}\ \mathit{False}\ x\ y = y
\end{aligned}
$$

Evaluate

$$
\begin{aligned}
& \quad \mathit{cond}\ (4 < 2)\ (5 * 17)\ (2 * 3)\\
\rightarrow\ & \{\text{check if expression matches True}\}\\
& \qquad 4 < 2\\
\rightarrow\ & \qquad \{\text{primitive comparison}\}\\
& \qquad \mathit{False}\\
\ldots\ & \quad \mathit{cond}\ \mathit{False}\ (5 * 7)\ (2 * 3)\\
\rightarrow\ & \{\text{check if expression matches False}\}\\
& \quad 2 * 3
\end{aligned}
$$

Ex 4/4: Pattern matching in function definitions

$$
\begin{aligned}
&\mathit{cond} :: \mathit{Bool} \rightarrow a \rightarrow a \rightarrow a \\
&\mathit{cond}\ \mathit{True}\ x\ y\ \ = x \\
&\mathit{cond}\ \mathit{False}\ x\ y = y
\end{aligned}
$$

Evaluate

$$
\begin{aligned}
& \mathit{cond}\ (4 < 2)\ (5 * 17)\ (2 * 3) \\
\rightarrow\ & \{\text{check if expression matches True}\} \\
& \quad 4 < 2 \\
\rightarrow\ & \quad \{\text{primitive comparison}\} \\
& \quad \mathit{False} \\
\ldots\ & \mathit{cond}\ \mathit{False}\ (5 * 7)\ (2 * 3) \\
\rightarrow\ & \{\text{check if expression matches False}\} \\
& 2 * 3 \\
\rightarrow\ & \{\text{primitive aritmetic}\} \\
& 6
\end{aligned}
$$

Lists

How do we proceed when evaluating

$$map\ (1+)\ (map\ (2*)\ [1,2,3])$$

in the **leftmost outermost reduction with sharing**?
Recall that

$$\begin{aligned} map\ f\ []\quad &= [] \\ map\ f\ (x:xs) &= f\ x : map\ f\ xs \end{aligned}$$

How do we proceed when evaluating

$$map\ (1+)\ (map\ (2*)\ [1,2,3])$$

in the **leftmost outermost reduction with sharing**?
Recall that

$$\begin{aligned} map\ f\ []\ &= [] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \end{aligned}$$

See that we calculate elements of the resulting list one by one. So the list we are working on could potentially be infinite, if we only consume finitely many elements!

Infinite lists

Lazy evaluation allows us to program with **infinite lists** of values!

Consider the recursive definition

$$\begin{aligned} &\mathit{ones} :: [\mathit{Int}] \\ &\mathit{ones} = 1 : \mathit{ones} \end{aligned}$$

Infinite lists

Lazy evaluation allows us to program with **infinite lists** of values!

Consider the recursive definition

$$ones :: [Int]$$
$$ones = 1 : ones$$

Unfolding the recursion a few times gives

$$\begin{aligned} ones &= 1 : ones \\ &= 1 : 1 : ones \\ &= 1 : 1 : 1 : ones \\ &= \ldots \end{aligned}$$

That is, *ones* is the **infinite list** of 1's.

Innermost vs. outermost

1. Now consider evaluating *head ones* using innermost reduction:

$$\begin{aligned} head\ ones &= head\ (1 : ones) \\ &= head\ (1 : 1 : ones) \\ &= head\ (1 : 1 : 1 : ones) \\ &= \ ... \end{aligned}$$

   Does not terminate!

Innermost vs. outermost

1. Now consider evaluating *head ones* using innermost reduction:

$$\begin{aligned} head\ ones &= head\ (1 : ones) \\ &= head\ (1 : 1 : ones) \\ &= head\ (1 : 1 : 1 : ones) \\ &= \ \ldots \end{aligned}$$

   Does not terminate!

2. And with outermost reduction:

$$\begin{aligned} head\ ones &= head\ (1 : ones) \\ &= 1 \end{aligned}$$

   Terminates!

Infinite lists

We now see that

$$ones = 1 : ones$$

really defines a **potentially infinite list** that is only evaluated as much as needed by the context in which it is used.

Modular programming

We can generate **finite** lists by taking elements from infinite lists. For example

- $take\ 5\ ones = [1, 1, 1, 1, 1]$
- $take\ 5\ [1\,..] = [1, 2, 3, 4, 5]$

Modular programming

We can generate **finite** lists by taking elements from infinite lists. For example

- $\mathit{take}\ 5\ \mathit{ones} = [1, 1, 1, 1, 1]$
- $\mathit{take}\ 5\ [1\,..] = [1, 2, 3, 4, 5]$

Lazy evaluation allows us to make programs **more modular**, by separating control from data:

$$\underbrace{\mathit{take}\ 5}_{\text{control}}\ \underbrace{[1\,..]}_{\text{data}}$$

Using lazy evaluation, the data is only evaluated as much as required by the control part.

Modular programming: Example

Consider

$$replicate :: Int \to a \to [a]$$

Modular programming: Example

Consider

$$replicate :: Int \to a \to [a]$$

In a non-lazy language, we would define it like:

$$replicate\ 0\ v = [\,]$$
$$replicate\ n\ v = v : replicate\ (n-1)\ v$$

Modular programming: Example

Consider

$$replicate :: Int \to a \to [a]$$

In a non-lazy language, we would define it like:

$$replicate\ 0\ v = [\,]$$
$$replicate\ n\ v = v : replicate\ (n-1)\ v$$

Since Haskell is lazy, we can make the even simpler function

$$repeat :: a \to [a]$$
$$repeat\ v = vs\ \textbf{where}\ vs = v : vs$$

and define *replicate* by

$$replicate\ n = take\ n \circ repeat$$

Primes: The seive of Eratosthenes

Syddansk Universitet
University of Southern Denmark

$$primes :: [Integer]$$
$$primes = seive\ [2\,..]$$

$$seive :: [Integer] \rightarrow [Integer]$$
$$seive\ (p : xs) = p : seive\ [x \mid x \leftarrow xs, x\ \text{`}mod\text{`}\ p \not\equiv 0]$$

$$primes = [2, 3, 5, 7, 11, 13, 17, 19...]$$

Fibonacci numbers

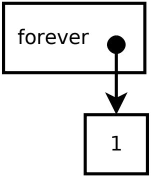
$$
\begin{aligned}
&\mathit{fibs} :: [\mathit{Integer}]\\
&\mathit{fibs} = 0 : 1 : \mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})
\end{aligned}
$$

Cyclic structures

Two similar structures with different representation in the computer's memory:

$$
\begin{aligned}
ones &= forever\ 1 \\
forever\ x &= x : forever\ x
\end{aligned}
$$

$$
\begin{aligned}
ones' &= forever'\ 1 \\
forever'\ x &= zs\ \textbf{where}\ zs = x : zs
\end{aligned}
$$

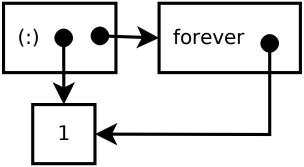


Cyclic structures

Two similar structures with different representation in the computer's memory:

$$
\begin{aligned}
&\mathit{ones} &&= \mathit{forever}\ 1\\
&\mathit{forever}\ x &&= x : \mathit{forever}\ x
\end{aligned}
$$

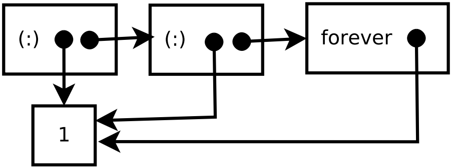
$$
\begin{aligned}
&\mathit{ones}' &&= \mathit{forever}'\ 1\\
&\mathit{forever}'\ x &&= \mathit{zs}\ \textbf{where}\ \mathit{zs} = x : \mathit{zs}
\end{aligned}
$$



Two similar structures with different representation in the computer's memory:

$$\begin{aligned} \mathit{ones} &= \mathit{forever}\ 1 \\ \mathit{forever}\ x &= x : \mathit{forever}\ x \end{aligned}$$

$$\begin{aligned} \mathit{ones}' &= \mathit{forever}'\ 1 \\ \mathit{forever}'\ x &= \mathit{zs}\ \textbf{where}\ \mathit{zs} = x : \mathit{zs} \end{aligned}$$

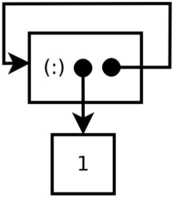


Cyclic structures

Two similar structures with different representation in the computer's memory:

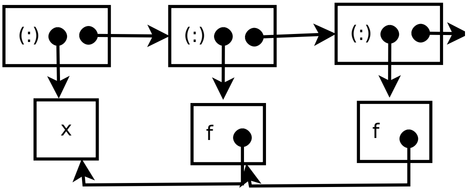
$$\begin{aligned} ones &= forever\ 1 \\ forever\ x &= x : forever\ x \end{aligned}$$

$$\begin{aligned} ones' &= forever'\ 1 \\ forever'\ x &= zs\ \mathbf{where}\ zs = x : zs \end{aligned}$$



The efficiency of *iterate*

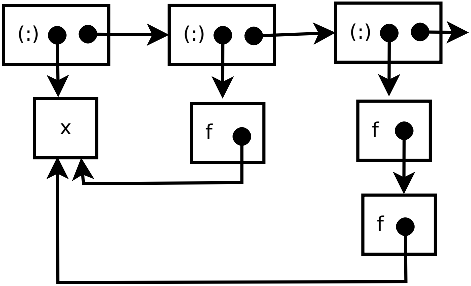
$$iterate\ f\ x = x : iterate\ f\ (f\ x)$$



The efficiency of *iterate*

$$iterate'\ f\ x = x : map\ f\ (iterate'\ f\ x)$$

Demo with :sprint.



Laziness for memoization

$$fibs = map\ fib'\ [0\,..]$$

$$fib'\ 0 = 0$$

$$fib'\ 1 = 1$$

$$fib'\ n = fib\ (n-1) + fib\ (n-2)$$

$$fib\ n = fibs\ !!\ n$$

```
> fib 20000
253116232373236124224O155...
(2.01 secs, 21312464 bytes)
> fib 20000
253116232373236124224O155...
(0.02 secs, 4668504 bytes)
```

Laziness for memoization

$$fibs = map\ fib'\ [0\,..]$$

$$fib'\ 0 = 0$$
$$fib'\ 1 = 1$$
$$fib'\ n = fib\ (n-1) + fib\ (n-2)$$

$$fib\ n = fibs\ !!\ n$$

```
*Main> :sprint fibs
fibs = _
*Main> take 1 fibs
[0]
*Main> :sprint fibs
fibs = 0 : _
*Main> fib 10
55
*Main> :sprint fibs
fibs = 0 : 1 : 1 : 2 : 3 : 5 : 8 : 13 : 21 : 34 : 55 : _
*Main>
```

The dark side of Lazy Evaluation

- We have seen, that *lazy evaluation* always uses the same or fewer number of reduction steps as *eager evaluation*.

The dark side of Lazy Evaluation

- We have seen, that *lazy evaluation* always uses the same or fewer number of reduction steps as *eager evaluation*.
- But: If you are not careful, your algorithm might use more **space** than needed. The unevaluated expressions take up space!

Summing up numbers

$$\mathit{foldr}\ f\ z\ [\,] = z$$
$$\mathit{foldr}\ f\ z\ (x : \mathit{xs}) = f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs})$$

$$\mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,])$$

Summing up numbers

$$foldr\ f\ z\ [\,] = z$$
$$foldr\ f\ z\ (x : xs) = f\ x\ (foldr\ f\ z\ xs)$$

$$foldr\ (+)\ 0\ (1 : 2 : 3 : [\,]) = 1 + foldr\ (+)\ 0\ (2 : 3 : [\,])$$

Summing up numbers

$$\begin{aligned} foldr\ f\ z\ [\,] &= z \\ foldr\ f\ z\ (x : xs) &= f\ x\ (foldr\ f\ z\ xs) \end{aligned}$$

$$\begin{aligned} foldr\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= 1 + foldr\ (+)\ 0\ (2 : 3 : [\,]) \\ &= 1 + (2 + foldr\ (+)\ 0\ (3 : [\,])) \end{aligned}$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldr}\ f\ z\ [\,] &= z \\
\mathit{foldr}\ f\ z\ (x : xs) &= f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= 1 + \mathit{foldr}\ (+)\ 0\ (2 : 3 : [\,]) \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3 : [\,])) \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,]))
\end{aligned}
$$

Summing up numbers

$$
\begin{aligned}
&\mathit{foldr}\ f\ z\ [\,] = z \\
&\mathit{foldr}\ f\ z\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= 1 + \mathit{foldr}\ (+)\ 0\ (2 : 3 : [\,]) \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3 : [\,])) \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,])) \\
&= 1 + (2 + (3 + 0))
\end{aligned}
$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldr}\ f\ z\ [\,] &= z \\
\mathit{foldr}\ f\ z\ (x : xs) &= f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= 1 + \mathit{foldr}\ (+)\ 0\ (2 : 3 : [\,]) \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3 : [\,])) \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,])) \\
&= 1 + (2 + (3 + 0)) \\
&= 1 + (2 + 3)
\end{aligned}
$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldr}\ f\ z\ [\,] &= z \\
\mathit{foldr}\ f\ z\ (x : xs) &= f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldr}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= 1 + \mathit{foldr}\ (+)\ 0\ (2 : 3 : [\,]) \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3 : [\,])) \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,])) \\
&= 1 + (2 + (3 + 0)) \\
&= 1 + (2 + 3) \\
&= 1 + 5
\end{aligned}
$$

Summing up numbers

$$\begin{aligned}
&\mathit{foldr}\ f\ z\ [\,] = z \\
&\mathit{foldr}\ f\ z\ (x : xs) = f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}$$

$$\begin{aligned}
\mathit{foldr}\ (+)\ 0\ (1:2:3:[\,]) &= 1 + \mathit{foldr}\ (+)\ 0\ (2:3:[\,]) \\
&= 1 + (2 + \mathit{foldr}\ (+)\ 0\ (3:[\,])) \\
&= 1 + (2 + (3 + \mathit{foldr}\ (+)\ 0\ [\,])) \\
&= 1 + (2 + (3 + 0)) \\
&= 1 + (2 + 3) \\
&= 1 + 5 \\
&= 6
\end{aligned}$$

$O(n)$ space! Overflow for large lists!

Summing up numbers

$$foldl\ f\ z\ [\,] = z$$
$$foldl\ f\ z\ (x : xs) = foldl\ f\ (f\ z\ x)\ xs$$

$$foldl\ (+)\ 0\ (1 : 2 : 3 : [\,])$$

Summing up numbers

$$foldl\ f\ z\ [\,] = z$$
$$foldl\ f\ z\ (x : xs) = foldl\ f\ (f\ z\ x)\ xs$$

$$foldl\ (+)\ 0\ (1 : 2 : 3 : [\,]) = foldl\ (+)\ (0 + 1)\ (2 : 3 : [\,])$$

Summing up numbers

$$
\begin{aligned}
&\mathit{foldl}\ f\ z\ [\,] = z \\
&\mathit{foldl}\ f\ z\ (x : xs) = \mathit{foldl}\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldl}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= \mathit{foldl}\ (+)\ (0 + 1)\ (2 : 3 : [\,]) \\
&= \mathit{foldl}\ (+)\ ((0 + 1) + 2)\ (3 : [\,])
\end{aligned}
$$

Summing up numbers

$$\begin{aligned} \mathit{foldl}\ f\ z\ [\,] &= z \\ \mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs \end{aligned}$$

$$\begin{aligned} \mathit{foldl}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= \mathit{foldl}\ (+)\ (0 + 1)\ (2 : 3 : [\,]) \\ &= \mathit{foldl}\ (+)\ ((0 + 1) + 2)\ (3 : [\,]) \\ &= \mathit{foldl}\ (+)\ (((0 + 1) + 2) + 3)\ ([\,]) \end{aligned}$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldl}\ f\ z\ [\,] &= z \\
\mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldl}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= \mathit{foldl}\ (+)\ (0 + 1)\ (2 : 3 : [\,]) \\
&= \mathit{foldl}\ (+)\ ((0 + 1) + 2)\ (3 : [\,]) \\
&= \mathit{foldl}\ (+)\ (((0 + 1) + 2) + 3)\ ([\,]) \\
&= ((0 + 1) + 2) + 3
\end{aligned}
$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldl}\ f\ z\ [\,] &= z \\
\mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldl}\ (+)\ 0\ (1:2:3:[\,]) &= \mathit{foldl}\ (+)\ (0+1)\ (2:3:[\,]) \\
&= \mathit{foldl}\ (+)\ ((0+1)+2)\ (3:[\,]) \\
&= \mathit{foldl}\ (+)\ (((0+1)+2)+3)\ ([\,]) \\
&= ((0+1)+2)+3 \\
&= (1+2)+3
\end{aligned}
$$

Summing up numbers

$$
\begin{aligned}
\mathit{foldl}\ f\ z\ [\,] &= z \\
\mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

$$
\begin{aligned}
\mathit{foldl}\ (+)\ 0\ (1 : 2 : 3 : [\,]) &= \mathit{foldl}\ (+)\ (0 + 1)\ (2 : 3 : [\,]) \\
&= \mathit{foldl}\ (+)\ ((0 + 1) + 2)\ (3 : [\,]) \\
&= \mathit{foldl}\ (+)\ (((0 + 1) + 2) + 3)\ ([\,]) \\
&= ((0 + 1) + 2) + 3 \\
&= (1 + 2) + 3 \\
&= 3 + 3
\end{aligned}
$$

Summing up numbers

$$\begin{aligned} \mathit{foldl}\ f\ z\ [\,] &= z \\ \mathit{foldl}\ f\ z\ (x : xs) &= \mathit{foldl}\ f\ (f\ z\ x)\ xs \end{aligned}$$

$$\begin{aligned} \mathit{foldl}\ (+)\ 0\ (1:2:3:[\,]) &= \mathit{foldl}\ (+)\ (0+1)\ (2:3:[\,]) \\ &= \mathit{foldl}\ (+)\ ((0+1)+2)\ (3:[\,]) \\ &= \mathit{foldl}\ (+)\ (((0+1)+2)+3)\ ([\,]) \\ &= ((0+1)+2)+3 \\ &= (1+2)+3 \\ &= 3+3 \\ &= 6 \end{aligned}$$

$O(n)$ space again! What do we do?

Forcing evaluation

Haskell has the following primitive function

$$seq :: a \to b \to b \quad \text{-- primitive}$$

The call

$$seq\ x\ y$$

evaluates x before returning y.
The function *seq* can be used to define strict function application:

$$(\$!) :: (a \to b) \to a \to b$$
$$f\ \$!\ x = x\ `seq`\ f\ x$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
(\lambda x \to ())\ \$!\ \bot &= \\
\mathit{seq}\ (\bot, \bot)\ () &= \\
\mathit{snd}\ \$!\ (\bot, \bot) &= \\
(\lambda x \to ())\ \$!\ (\lambda x \to \bot) &= \\
\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1,2] &= \\
\mathit{seq}\ (\bot + \bot)\ () &= \\
\mathit{seq}\ (\mathit{foldr}\ \bot\ \bot)\ () &= \\
\mathit{seq}\ (1 : \bot)\ () &=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to ())\ \$!\ \bot &&= \bot \\
&\mathit{seq}\ (\bot, \bot)\ () &&= \\
&\mathit{snd}\ \$!\ (\bot, \bot) &&= \\
&(\lambda x \to ())\ \$!\ (\lambda x \to \bot) &&= \\
&\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1, 2] &&= \\
&\mathit{seq}\ (\bot + \bot)\ () &&= \\
&\mathit{seq}\ (\mathit{foldr}\ \bot\ \bot)\ () &&= \\
&\mathit{seq}\ (1 : \bot)\ () &&=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
(\lambda x \to ())\ \$!\ \bot &= \bot \\
seq\ (\bot, \bot)\ () &= () \\
snd\ \$!\ (\bot, \bot) &= \\
(\lambda x \to ())\ \$!\ (\lambda x \to \bot) &= \\
length\ \$!\ map\ \bot\ [1, 2] &= \\
seq\ (\bot + \bot)\ () &= \\
seq\ (foldr\ \bot\ \bot)\ () &= \\
seq\ (1 : \bot)\ () &=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to ())\ \$!\ \bot &&= \bot\\
&\mathit{seq}\ (\bot, \bot)\ () &&= ()\\
&\mathit{snd}\ \$!\ (\bot, \bot) &&= \bot\\
&(\lambda x \to ())\ \$!\ (\lambda x \to \bot) &&=\\
&\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1,2] &&=\\
&\mathit{seq}\ (\bot + \bot)\ () &&=\\
&\mathit{seq}\ (\mathit{foldr}\ \bot\ \bot)\ () &&=\\
&\mathit{seq}\ (1 : \bot)\ () &&=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
(\lambda x \to ()) \ \$! \ \bot &= \bot \\
\mathit{seq} \ (\bot, \bot) \ () &= () \\
\mathit{snd} \ \$! \ (\bot, \bot) &= \bot \\
(\lambda x \to ()) \ \$! \ (\lambda x \to \bot) &= () \\
\mathit{length} \ \$! \ \mathit{map} \ \bot \ [1,2] &= \\
\mathit{seq} \ (\bot + \bot) \ () &= \\
\mathit{seq} \ (\mathit{foldr} \ \bot \ \bot) \ () &= \\
\mathit{seq} \ (1 : \bot) \ () &=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
(\lambda x \to ())\ \$!\ \bot &= \bot \\
seq\ (\bot, \bot)\ () &= () \\
snd\ \$!\ (\bot, \bot) &= \bot \\
(\lambda x \to ())\ \$!\ (\lambda x \to \bot) &= () \\
length\ \$!\ map\ \bot\ [1, 2] &= 2 \\
seq\ (\bot + \bot)\ () &= \\
seq\ (foldr\ \bot\ \bot)\ () &= \\
seq\ (1 : \bot)\ () &=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \rightarrow ()) \,\$!\, \bot &&= \bot\\
&seq\ (\bot, \bot)\ () &&= ()\\
&snd \,\$!\, (\bot, \bot) &&= \bot\\
&(\lambda x \rightarrow ()) \,\$!\, (\lambda x \rightarrow \bot) &&= ()\\
&length \,\$!\, map\ \bot\ [1, 2] &&= 2\\
&seq\ (\bot + \bot)\ () &&= \bot\\
&seq\ (foldr\ \bot\ \bot)\ () &&=\\
&seq\ (1 : \bot)\ () &&=
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)
What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \rightarrow ())\ \$!\ \bot && = \bot \\
&\mathit{seq}\ (\bot, \bot)\ () && = () \\
&\mathit{snd}\ \$!\ (\bot, \bot) && = \bot \\
&(\lambda x \rightarrow ())\ \$!\ (\lambda x \rightarrow \bot) && = () \\
&\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1,2] && = 2 \\
&\mathit{seq}\ (\bot + \bot)\ () && = \bot \\
&\mathit{seq}\ (\mathit{foldr}\ \bot\ \bot)\ () && = () \\
&\mathit{seq}\ (1 : \bot)\ () && =
\end{aligned}
$$

Quiz time!

Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

$$
\begin{aligned}
&(\lambda x \to ())\ \$!\ \bot && = \bot \\
&\mathit{seq}\ (\bot, \bot)\ () && = () \\
&\mathit{snd}\ \$!\ (\bot, \bot) && = \bot \\
&(\lambda x \to ())\ \$!\ (\lambda x \to \bot) && = () \\
&\mathit{length}\ \$!\ \mathit{map}\ \bot\ [1, 2] && = 2 \\
&\mathit{seq}\ (\bot + \bot)\ () && = \bot \\
&\mathit{seq}\ (\mathit{foldr}\ \bot\ \bot)\ () && = () \\
&\mathit{seq}\ (1 : \bot)\ () && = ()
\end{aligned}
$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \textbf{let}\ z' = f\ z\ x\ \textbf{in}\ z'\ `seq`\ (foldl\ f\ z'\ xs)$$

$$foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,])$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \mathbf{let}\ z' = f\ z\ x\ \mathbf{in}\ z'\ \text{`}seq\text{`}\ (foldl\ f\ z'\ xs)$$

$$foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,])$$
$$= \mathbf{let}\ z' = (0 + 1)\ \mathbf{in}\ z'\ \text{`}seq\text{`}\ foldl'\ (+)\ z'\ (2 : 3 : [\,])$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \mathbf{let}\ z' = f\ z\ x\ \mathbf{in}\ z'\ `seq`\ (foldl\ f\ z'\ xs)$$

$$\begin{aligned} & foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,]) \\ & = \mathbf{let}\ z' = (0 + 1)\ \mathbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (2 : 3 : [\,]) \\ & = foldl'\ (+)\ 1\ (2 : 3 : [\,]) \end{aligned}$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \mathbf{let}\ z' = f\ z\ x\ \mathbf{in}\ z'\ \text{`}seq\text{`}\ (foldl\ f\ z'\ xs)$$

$$\begin{aligned}
&foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,]) \\
&= \mathbf{let}\ z' = (0 + 1)\ \mathbf{in}\ z'\ \text{`}seq\text{`}\ foldl'\ (+)\ z'\ (2 : 3 : [\,]) \\
&= foldl'\ (+)\ 1\ (2 : 3 : [\,]) \\
&= \mathbf{let}\ z' = (1 + 2)\ \mathbf{in}\ z'\ \text{`}seq\text{`}\ foldl'\ (+)\ z'\ (3 : [\,])
\end{aligned}$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \textbf{let}\ z' = f\ z\ x\ \textbf{in}\ z'\ `seq`\ (foldl\ f\ z'\ xs)$$

$$\begin{aligned}
&foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,])\\
&= \textbf{let}\ z' = (0 + 1)\ \textbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (2 : 3 : [\,])\\
&= foldl'\ (+)\ 1\ (2 : 3 : [\,])\\
&= \textbf{let}\ z' = (1 + 2)\ \textbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (3 : [\,])\\
&= foldl'\ (+)\ 3\ (3 : [\,])
\end{aligned}$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \mathbf{let}\ z' = f\ z\ x\ \mathbf{in}\ z'\ `seq`\ (foldl\ f\ z'\ xs)$$

$$\begin{aligned}
& foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,]) \\
& = \mathbf{let}\ z' = (0 + 1)\ \mathbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (2 : 3 : [\,]) \\
& = foldl'\ (+)\ 1\ (2 : 3 : [\,]) \\
& = \mathbf{let}\ z' = (1 + 2)\ \mathbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (3 : [\,]) \\
& = foldl'\ (+)\ 3\ (3 : [\,]) \\
& = \mathbf{let}\ z' = (3 + 3)\ \mathbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (3 : [\,])
\end{aligned}$$

Summing up numbers

$$foldl'\ f\ z\ [\,] = z$$
$$foldl'\ f\ z\ (x : xs) = \textbf{let}\ z' = f\ z\ x\ \textbf{in}\ z'\ `seq`\ (foldl\ f\ z'\ xs)$$

$$
\begin{aligned}
& foldl'\ (+)\ 0\ (1 : 2 : 3 : [\,]) \\
& = \textbf{let}\ z' = (0 + 1)\ \textbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (2 : 3 : [\,]) \\
& = foldl'\ (+)\ 1\ (2 : 3 : [\,]) \\
& = \textbf{let}\ z' = (1 + 2)\ \textbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (3 : [\,]) \\
& = foldl'\ (+)\ 3\ (3 : [\,]) \\
& = \textbf{let}\ z' = (3 + 3)\ \textbf{in}\ z'\ `seq`\ foldl'\ (+)\ z'\ (3 : [\,]) \\
& = foldl'\ (+)\ 6\ [\,]
\end{aligned}
$$

Summing up numbers

$$\begin{aligned}
&\mathit{foldl}'\ f\ z\ []=z\\
&\mathit{foldl}'\ f\ z\ (x:xs)=\textbf{let}\ z'=f\ z\ x\ \textbf{in}\ z'\ \text{`}\mathit{seq}\text{`}\ (\mathit{foldl}\ f\ z'\ xs)
\end{aligned}$$

$$\begin{aligned}
&\mathit{foldl}'\ (+)\ 0\ (1:2:3:[])\\
&\ =\textbf{let}\ z'=(0+1)\ \textbf{in}\ z'\ \text{`}\mathit{seq}\text{`}\ \mathit{foldl}'\ (+)\ z'\ (2:3:[])\\
&\ =\mathit{foldl}'\ (+)\ 1\ (2:3:[])\\
&\ =\textbf{let}\ z'=(1+2)\ \textbf{in}\ z'\ \text{`}\mathit{seq}\text{`}\ \mathit{foldl}'\ (+)\ z'\ (3:[])\\
&\ =\mathit{foldl}'\ (+)\ 3\ (3:[])\\
&\ =\textbf{let}\ z'=(3+3)\ \textbf{in}\ z'\ \text{`}\mathit{seq}\text{`}\ \mathit{foldl}'\ (+)\ z'\ (3:[])\\
&\ =\mathit{foldl}'\ (+)\ 6\ []\\
&\ =6
\end{aligned}$$

$O(1)$ space.