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:

\[
\text{root } a \ b \ c = \frac{(-b + sd)}{(2 \times a)}
\]

**where**

\[
sd = \sqrt{d} \\
\quad d = b \times b - 4 \times a \times c
\]

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

\[
> \text{root } 1 \ 0 \ (-2) \\
1.4142135623730951
\]
Lazy evaluation

- The purity of Haskell functions (= absence 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

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

\[ a :: \text{Bool} \]
\[ a = \neg a \]

What happens if we evaluate \( a \)?
Example 1

\[
\begin{align*}
a &:: \text{Bool} \\
a &= \neg a \\
\end{align*}
\]

What happens if we evaluate \(a\)?

\[
\begin{align*}
b &:: \text{Bool} \\
b &= \text{False} \\
c &:: \text{Bool} \\
c &= b \land a \\
\end{align*}
\]

What happens if we evaluate \(c\)?
Example 1

\[ a :: \text{Bool} \]
\[ a = \neg a \]

What happens if we evaluate \( a \)?

\[ b :: \text{Bool} \]
\[ b = \text{False} \]
\[ c :: \text{Bool} \]
\[ c = b \land a \]

What happens if we evaluate \( c \)?

\[ c' :: \text{Bool} \]
\[ c' = a \land b \]

What about \( c' \)?
Example 1

\[ a :: \text{Bool} \]
\[ a = \neg a \]

What happens if we evaluate \( a \)?

\[ b :: \text{Bool} \]
\[ b = \text{False} \]
\[ c :: \text{Bool} \]
\[ c = b \land a \]

What happens if we evaluate \( c \)?

\[ c' :: \text{Bool} \]
\[ c' = a \land b \]

What about \( c' \)?

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

Note that \((\land)\) and booleans \text{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

\[
\text{myIf} :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a
\]

\[
\text{myIf True } v \_ = v
\]

\[
\text{myIf False } \_ v = v
\]

What happens when we evaluate

\[
\text{myIf True expr1 expr2}
\]

?
Example 2

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

\[
myIf :: \text{Bool} \rightarrow a \rightarrow a \\
myIf \ True \ v _ = v \\
myIf \ False \ _ \ v = v
\]

What happens when we evaluate

\[
myIf \ True \ expr1 \ expr2
\]

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 successively applying definitions, until no more simplifications are possible.
Reducible expressions

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

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

Function application is also known as a \(\beta\)-reduction.
Reducible expressions

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

Examples:

\((\lambda x \to x \ast 2)\)
Reducible expressions

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

Examples:

\[(\lambda x \to x \ast 2)\] Not reducible
Reducible expressions

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

Examples:
- \((\lambda x \rightarrow x \times 2)\) Not reducible
- \((\lambda x \rightarrow x \times 2)\) 5
Reducible expressions

• We know that Haskell evaluates expressions by successively applying definitions, until no more simplifications are possible.

• Expressions which can be simplified are called “reducible expressions”.

Examples:

\((\lambda x \to x \ast 2)\)  
Not reducible

\((\lambda x \to x \ast 2) 5\)  
Reducible to \(5 \ast 2\), which is reducible to \(10\).
Reducible expressions

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

**Examples:**

- \((\lambda x \rightarrow x \ast 2)\) Not reducible
- \((\lambda x \rightarrow x \ast 2)\ 5\) Reducible to \(5 \ast 2\), which is reducible to 10
- \((\lambda x\ y \rightarrow 13 \ast x + y)\ 2\)
Reducible expressions

• We know that Haskell evaluates expressions by successively applying definitions, until no more simplifications are possible.

• Expressions which can be simplified are called “reducible expressions”.

Examples:

(λx → x * 2) Not reducible
(λx → x * 2) 5 Reducible to 5 * 2, which is reducible to 10
(λx y → 13 * x + y) 2 Reducible to (λy → 13 * 2 + y).
Then subexpression 13 * 2 is reducible.
Reducible expressions

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

Examples:

\[(\lambda x \to x \times 2)\] Not reducible

\[(\lambda x \to x \times 2) 5\] Reducible to \(5 \times 2\), which is reducible to 10

\[(\lambda x \ y \to 13 \times x + y) \ 2\] Reducible to \(\lambda y \to 13 \times 2 + y\). Then subexpression \(13 \times 2\) is reducible.

\[(7, 5 + 3)\]
Reducible expressions

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

Examples:

\[(\lambda x \to x \times 2)\] Not reducible

\[(\lambda x \to x \times 2)\ 5\] Reducible to \(5 \times 2\), which is reducible to 10

\[(\lambda x\ y \to 13 \times x + y)\ 2\] Reducible to \((\lambda y \to 13 \times 2 + y)\).
Then subexpression \(13 \times 2\) is reducible.

\[(7, 5 + 3)\] Not reducible, but subexpression \(5 + 3\) is.

Function application is also known as a \(\beta\)-reduction.
Reduction strategies for function application

Consider the function \( s qr \ x = x \ast x \) and the expression \( s qr \ (4 + 2) \)

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

  \[
  s qr \ (4 + 2) 
  \]
Reduction strategies for function application
Consider the function \( sqr \ x = x \ast x \) and the expression \( sqr \ (4 + 2) \)

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

\[
sqr \ (4 + 2) \equiv sqr \ 6 \equiv 6 \ast 6 \equiv 36
\]
Reduction strategies for function application

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

- **Innermost reduction**: [call-by-value]
  \[
  sqr \ (4 + 2) \equiv sqr \ 6 \equiv 6 \times 6 \equiv 36
  \]

- **Outermost reduction**: [call-by-need]
  \[
  sqr \ (4 + 2)
  \]
Reduction strategies for function application

Consider the function \( sqr \, x = x \times x \) and the expression \( sqr \, (4 + 2) \)

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

\[
sqr \, (4 + 2) \equiv sqr \, 6 \equiv 6 \times 6 \equiv 36
\]

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

\[
sqr \, (4 + 2) \equiv (4 + 2) \times (4 + 2) \equiv 6 \times (4 + 2) \equiv 6 \times 6 \equiv 36
\]
Reduction strategies for function application

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

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

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

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

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

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

  $$sqr\ (4 + 2)$$
Reduction strategies for function application

Consider the function \( \text{sqr} \ x = x \times x \) and the expression \( \text{sqr} \ (4 + 2) \)

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

\[
\text{sqr} \ (4 + 2) \equiv \text{sqr} \ 6 \equiv 6 \times 6 \equiv 36
\]

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

\[
\text{sqr} \ (4 + 2) \equiv (4 + 2) \times (4 + 2) \equiv 6 \times (4 + 2) \equiv 6 \times 6 \equiv 36
\]

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

\[
\text{sqr} \ (4 + 2) \equiv \text{let} \ x = 4 + 2 \ \text{in} \ x \times x \equiv 6 \times 6 \equiv 36
\]

The general evaluation rule of Haskell can be described as

Leftmost outermost reduction with sharing
One more example

\[
\text{ackermann } m \; n
\]

\[
\begin{align*}
| m \equiv 0 & \quad = n + 1 \\
| m > 0 \land n \equiv 0 & \quad = \text{ackermann} \; (m - 1) \; 1 \\
| m > 0 \land n > 0 & \quad = \text{ackermann} \; (m - 1) \; (\text{ackermann} \; m \; (n - 1))
\end{align*}
\]

How does Haskell evaluate?

\[
\text{const 5 } (\text{ackermann} \; 4 \; 2)
\]
Primitive arithmetic operations $(+), (-), (\ast), (\div)$

Cannot be evaluated in outermost manner:

$$3 \ast 4 + 3 \ast 4$$
Primitive arithmetic operations $(+), (−), (∗), (/)$

Cannot be evaluated in outermost manner:

$$3 \ast 4 + 3 \ast 4 \rightarrow \{ \text{first operand not done} \}$$

$$3 \ast 4$$

$$12$$

We say that the functions are strict in both arguments.
Primitive arithmetic operations (+), (−), (∗), (/)

Cannot be evaluated in outermost manner:

\[ 3 \times 4 + 3 \times 4 \]

\[ \rightarrow \{ \text{first operand not done} \} \]

\[ 3 \times 4 \]

\[ \rightarrow \{ \text{both operands done, arithmetic} \} \]

\[ 12 \]

\[ \ldots 12 + 3 \times 4 \]
Primitive arithmetic operations \((+), (−), (∗), (\div)\)

Cannot be evaluated in outermost manner:

\[
3 \times 4 + 3 \times 4
\]

\rightarrow \{\text{first operand not done}\}

\[
3 \times 4
\]

\rightarrow \{\text{both operands done, arithmetic}\}

\[
12
\]

... \[12 + 3 \times 4\]

\rightarrow \{\text{second operand not done}\}

\[
3 \times 4
\]
Primitive arithmetic operations (+), (−), (∗), (⁄)

Cannot be evaluated in outermost manner:

\[ 3 \times 4 + 3 \times 4 \]
\[ \rightarrow \{ \text{first operand not done} \} \]
\[ 3 \times 4 \]
\[ \rightarrow \{ \text{both operands done, arithmetic} \} \]
\[ 12 \]

... \[ 12 + 3 \times 4 \]
\[ \rightarrow \{ \text{second operand not done} \} \]
\[ 3 \times 4 \]
\[ \rightarrow \{ \text{both operands done, arithmetic} \} \]
\[ 12 \]

... \[ 12 + 12 \]
Primitive arithmetic operations \((+), (-), (*), (/)\)

Cannot be evaluated in outermost manner:

\[ 3 \times 4 + 3 \times 4 \]
\[ \rightarrow \{ \text{first operand not done}\} \]
\[ 3 \times 4 \]
\[ \rightarrow \{ \text{both operands done, arithmetic}\} \]
\[ 12 \]
... \[ 12 + 3 \times 4 \]
\[ \rightarrow \{ \text{second operand not done}\} \]
\[ 3 \times 4 \]
\[ \rightarrow \{ \text{both operands done, arithmetic}\} \]
\[ 12 \]
... \[ 12 + 12 \]
\[ \rightarrow \{ \text{both operands done, arithmetic}\} \]
\[ 24 \]

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

Consider again the function

\[ \text{sqr} \ x = x \times x \]

How does Haskell evaluate

\[ \text{sqr} \ (\text{sqr} \ 2) \]
Another example

Consider again the function

\[ \text{sqr } x = x \times x \]

How does Haskell evaluate

\[ \text{sqr } (\text{sqr } 2) \equiv \text{let } x = \text{sqr } 2 \text{ in } x \times x \quad \text{-- (apply outer } \text{sqr}) \]
\[ \equiv \text{let } x = 2 \times 2 \text{ in } x \times x \quad \text{-- (apply inner } \text{sqr}) \]
\[ \equiv \text{let } x = 4 \text{ in } x \times x \quad \text{-- (apply inner } (\ast)) \]
\[ \equiv 16 \quad \text{-- (apply outer } (\ast)) \]
Innermost and outermost reductions

Which is innermost and which is outermost?
(λx → x) ((λy → y) z)

• when an expression contains no reducible subexpression, it is in **normal form**
• when an expression contains no reducible subexpression, it is in **normal form**
• some expressions might not have a normal form
(\lambda x \rightarrow x) ( (\lambda y \rightarrow y) z )

- 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
• 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
• 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 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 occurrences of a formal argument must all point to the same actual argument (sharing)
Example (function which is not a lambda term)

\[(\text{if True then } (\lambda c \rightarrow (c, 0)) \text{ else } (\lambda c \rightarrow (c, 1))) \text{ blah}\]
Example (function which is not a lambda term)

\[
\textbf{if} \ True \ \textbf{then} \ (\lambda c \rightarrow (c, 0)) \ \textbf{else} \ (\lambda c \rightarrow (c, 1))) \ \textbf{blah} \\
\rightarrow \ \{ \text{need function} \} \\
\quad \textbf{if} \ True \ \textbf{then} \ (\lambda c \rightarrow (c, 0)) \ \textbf{else} \ (\lambda c \rightarrow (c, 1))
\]
Example (function which is not a lambda term)

\[(\text{if } True \text{ then } (\lambda c \to (c, 0)) \text{ else } (\lambda c \to (c, 1))) \text{ blah} \]
\[\to \{\text{need function}\}\]
\[\text{if } True \text{ then } (\lambda c \to (c, 0)) \text{ else } (\lambda c \to (c, 1))\]
\[\to \{\text{if-then-else}\}\]
\[(\lambda c \to (c, 0))\]
\[... (\lambda c \to (c, 0)) \text{ blah}\]
Example (function which is not a lambda term)

\[
(\textit{if } \text{True} \textit{ then } (\lambda c \rightarrow (c, 0)) \textit{ else } (\lambda c \rightarrow (c, 1))) \textit{ blah} \\
\rightarrow \text{\{need function\}} \\
\textit{if } \text{True} \textit{ then } (\lambda c \rightarrow (c, 0)) \textit{ else } (\lambda c \rightarrow (c, 1)) \\
\rightarrow \text{\{if-then-else\}} \\
(\lambda c \rightarrow (c, 0)) \\
\ldots \ (\lambda c \rightarrow (c, 0)) \textit{ blah} \\
\rightarrow \text{\{function application\}} \\
(\textit{blah}, 0)
\]
Example (function with multiple arguments)

\((\lambda c \rightarrow c) \ True \ (3 \ast 4 + 3 \ast 4)\)
Example (function with multiple arguments)

$(\lambda c \ _ \ → \ c) \ True \ (3 \ * \ 4 \ + \ 3 \ * \ 4)$

$\rightarrow \ \{\text{need function}\}$

$(\lambda c \ _ \ → \ c) \ True$
Example (function with multiple arguments)

\[(\lambda x \to x) \ True \ (3 \cdot 4 + 3 \cdot 4)\]

\[
\to \ \{\text{need function}\} \\
(\lambda x \to x) \ True \\
\to \ \{\text{function application}\} \\
(\_ \to \ True) \\
\ldots \ (\_ \to \ True) \ (3 \cdot 4 + 3 \cdot 4)\]
Example (function with multiple arguments)

\[(\lambda c \_ \rightarrow c) \ True \ (3 \ast 4 + 3 \ast 4)\]
\[
\rightarrow \ \{\text{need function}\}\n
\[(\lambda c \_ \rightarrow c) \ True\]
\[
\rightarrow \ \{\text{function application}\}\n
\[\_ \rightarrow True\]
\[
... \ (\_ \rightarrow True) \ (3 \ast 4 + 3 \ast 4)\]
\[
\rightarrow \ \{\text{function application}\}\n
True\]
Quiz time!

What does the following expressions evaluate to?

(\lambda x \to x) \ True \ = \ True
(\lambda x \to x) \ \bot \ = \ \bot
(\lambda x \to () \ ) \ \bot \ = \ ()
(\lambda x \to \bot \ ) \ () \ = \ \bot
(\lambda x \ f \to f \ x) \ \bot \ = \ \bot
length \ (map \ \bot \ [1,2]) \ = \
Quiz time!

What does the following expressions evaluate to?

$$(\lambda x \to x)\; True = True$$

$$\ (\lambda x \to x) \ \bot =$$

$$\ (\lambda x \to \ () ) \ \bot =$$

$$\ (\lambda x \to \bot ) \ () =$$

$$\ (\lambda x \ f \to f \ x) \ \bot =$$

$$length\ (map\ \bot\ [1,2]) =$$
Quiz time!

What does the following expressions evaluate to?

\[
\begin{align*}
(\lambda x \to x) \text{ True} & = \text{ True} \\
(\lambda x \to x) \perp & = \perp \\
(\lambda x \to ()) \perp & = \\
(\lambda x \to \perp) () & = \\
(\lambda x f \to f x) \perp & = \\
\text{length } (\text{map } \perp [1,2]) & =
\end{align*}
\]
Quiz time!

What does the following expressions evaluate to?

\[(\lambda x \to x) \ True \quad = \ True\]
\[(\lambda x \to x) \ \bot \quad = \ \bot\]
\[(\lambda x \to () \} \ \bot \quad = \ ()\]
\[(\lambda x \to \bot) \ () \quad = \]
\[(\lambda x f \to f \ x) \ \bot \quad = \]
\[\text{length} \ (\text{map} \ \bot \ [1,2]) \quad = \]
Quiz time!

What does the following expressions evaluate to?

\[
\begin{align*}
(\lambda x \to x) \ True & = True \\
(\lambda x \to x) \ \bot & = \bot \\
(\lambda x \to ()) \ \bot & = () \\
(\lambda x \to \bot) \ () & = \bot \\
(\lambda x f \to f \ x) \ \bot & = \\
\text{length} \ (\text{map} \ \bot \ [1, 2]) & =
\end{align*}
\]
Quiz time!

What does the following expressions evaluate to?

\[(\lambda x \to x) \; True \] = 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  
\[\text{length} \; (\text{map} \; \bot \; [1, 2]) \] =
Quiz time!

What does the following expressions evaluate to?

\[(\lambda x \to x) \ True = True\]
\[(\lambda x \to x) \ \bot = \bot\]
\[(\lambda x \to (())) \ \bot = ()\]
\[(\lambda x \to \bot) () = \bot\]
\[\lambda x f \to f x \bot = \lambda f \to f \ \bot\]
\[\text{length} (\text{map} \ \bot [1,2]) = 2\]
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.

\[
\text{case const } (\text{Just } a) \ b \ \text{of } \{ \text{Just } \_ \rightarrow \text{True}; \text{Nothing} \rightarrow \text{False}\} \\
\rightarrow \{\text{evaluate argument for pattern Just } \_ \} \\
\text{const } (\text{Just } a) \ b
\]
**Pattern matching**

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

```java
    case const (Just a) b of {Just _ → True; Nothing → False}
    → {evaluate argument for pattern Just _ }
        const (Just a) b
    → {function application}
        Just a

    ... case Just a of {Just _ → True; Nothing → False}
```
Pattern matching

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

```plaintext
  case const (Just a) b of {Just _ → True; Nothing → False}
    → {evaluate argument for pattern Just _ }
      const (Just a) b
    → {function application}
      Just a
  ...
  case Just a of {Just _ → True; Nothing → False}
    → {match pattern}
      True
```
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, Just (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`, `Just (square 42)` or `(:) 1 []`
- a built-in function applied to too few arguments (perhaps none) like `(+) 2` or `sqrt`.
- or a lambda abstraction `\( \lambda x \rightarrow \text{expression} \).`
Ex 1/4: Pattern matching (Level 1) - Built-in primitive

\[
\text{case } 3 \ast 4 \text{ of } \{ 0 \rightarrow \text{True}; \_ \rightarrow \text{False} \}
\]
Ex 1/4: Pattern matching (Level 1) - Built-in primitive

\[
\begin{align*}
\text{case } 3 \times 4 \text{ of } & \{ 0 \rightarrow True; \_ \rightarrow False \} \\
\rightarrow & \{ \text{evaluate argument for pattern 0} \} \\
3 \times 4 & 
\end{align*}
\]
case 3 * 4 of \{ 0 \rightarrow True; _ \rightarrow False \} \\
\rightarrow \{ \text{evaluate argument for pattern 0} \} \\
3 * 4 \\
\rightarrow \{ \text{arithmetic} \} \\
12 \\
... \ case 12 of \{ 0 \rightarrow True; _ \rightarrow False \}
Ex 1/4: Pattern matching (Level 1) - Built-in primitive

\[
\text{case } 3 \times 4 \text{ of } \{ 0 \rightarrow \text{True}; \_ \rightarrow \text{False} \} \\
\rightarrow \{ \text{evaluate argument for pattern 0} \} \\
\quad 3 \times 4 \\
\rightarrow \{ \text{arithmetic} \} \\
\quad 12 \\
\ldots \text{ case } 12 \text{ of } \{ 0 \rightarrow \text{True}; \_ \rightarrow \text{False} \} \\
\rightarrow \{ \text{match pattern} \} \\
False
\]
Ex 2/4: Pattern matching (Level 0) - Variable or wildcard

```
case 3 * 4 of {x → True}
```
Ex 2/4: Pattern matching (Level 0) - Variable or wildcard

case 3 * 4 of \{ x \rightarrow True \} \\
\rightarrow \{ \text{match pattern} \} \\
True
Ex 3/4: Pattern matching (Level 1) - Variable in pattern

\[
\text{case } \text{Just } (3 \times 4) \text{ of } \{ \text{Just } x \rightarrow (x, x); \text{Nothing } \rightarrow (0, 0) \}\n\]
Ex 3/4: Pattern matching (Level 1) - Variable in pattern

\[
\text{case } Just (3 \times 4) \text{ of } \{ \text{Just } x \rightarrow (x, x); \text{Nothing } \rightarrow (0, 0) \} \\
\rightarrow \{ \text{match pattern} \} \\
\text{let } x = 3 \times 4 \text{ in } (x, x)
\]
Ex 4/4: Pattern matching in function definitions

\[
\text{cond} :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a
\]

\[
\text{cond True } x \ y = x
\]

\[
\text{cond False } x \ y = y
\]

Evaluate

\[
\text{cond } (4 < 2) \ (5 \ast 17) \ (2 \ast 3)
\]
Ex 4/4: Pattern matching in function definitions

\[
\text{cond} :: \text{Bool} \to a \to a \to a \\
\text{cond True } x \ y = x \\
\text{cond False } x \ y = y
\]

Evaluate

\[
\text{cond } (4 < 2) \ (5 \times 17) \ (2 \times 3) \\
\rightarrow \{\text{check if expression matches True}\} \\
4 < 2
\]
Ex 4/4: Pattern matching in function definitions

\[ \text{cond :: Bool} \rightarrow a \rightarrow a \rightarrow a \]
\[ \text{cond True } x \ y = x \]
\[ \text{cond False } x \ y = y \]

Evaluate

\[ \text{cond \ (4 < 2) \ (5 \ast 17) \ (2 \ast 3)} \]
\[ \rightarrow \ \{ \text{check if expression matches True} \} \]
\[ \quad 4 < 2 \]
\[ \rightarrow \ \{ \text{primitive comparison} \} \]
\[ \quad False \]
\[ ... \ \text{cond False \ (5 \ast 7) \ (2 \ast 3)} \]
Ex 4/4: Pattern matching in function definitions

\[
cond :: \text{Bool} \rightarrow a \rightarrow a \rightarrow a
\]

\[
cond \ True \ x \ y = x
\]

\[
cond \ False \ x \ y = y
\]

Evaluate

\[
cond \ (4 < 2) \ (5 \times 17) \ (2 \times 3)
\]

\[
\rightarrow \ \{\text{check if expression matches True}\}
\]

\[
4 < 2
\]

\[
\rightarrow \ \{\text{primitive comparison}\}
\]

\[
False
\]

...  \ cond \ False \ (5 \times 7) \ (2 \times 3)

\[
\rightarrow \ \{\text{check if expression matches False}\}
\]

\[
2 \times 3
\]
Ex 4/4: Pattern matching in function definitions

\[
\begin{align*}
\text{cond} :: \text{Bool} & \rightarrow a \rightarrow a \rightarrow a \\
\text{cond True } x \ y & = x \\
\text{cond False } x \ y & = y
\end{align*}
\]

Evaluate

\[
\begin{align*}
\text{cond } (4 < 2) (5 \times 17) (2 \times 3) \\
\rightarrow & \{\text{check if expression matches True}\} \\
4 < 2 \\
\rightarrow & \{\text{primitive comparison}\} \\
\text{False} \\
\text{... cond False } (5 \times 7) (2 \times 3) \\
\rightarrow & \{\text{check if expression matches False}\} \\
2 \times 3 \\
\rightarrow & \{\text{primitive arithmetic}\} \\
6
\end{align*}
\]
Lists

How do we proceed when evaluating

\[\text{map } (1+) \ (\text{map } (2*) \ [1, 2, 3])\]

in the leftmost outermost reduction with sharing?

Recall that

\[\text{map } f \ [\ ] = [\ ]\]

\[\text{map } f \ (x : xs) = f \ x : \text{map } f \ xs\]
Lists

How do we proceed when evaluating

\[ map\ (1+)\ (map\ (2\ast)\ [1, 2, 3])\]

in the leftmost outermost reduction with sharing?

Recall that

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

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!
Lazy evaluation allows us to program with **infinite lists** of values!
Consider the recursive definition

\[ \textit{ones} :: [\textit{Int}] \]
\[ \textit{ones} = 1 : \textit{ones} \]
Lazy evaluation allows us to program with infinite lists of values!
Consider the recursive definition

$$
\text{ones} :: [\text{Int}]
\text{ones} = 1 : \text{ones}
$$

Unfolding the recursion a few times gives

$$
\text{ones} = 1 : \text{ones}
\quad = 1 : 1 : \text{ones}
\quad = 1 : 1 : 1 : \text{ones}
\quad = ... 
$$

That is, \text{ones} is the infinite list of 1’s.
Innermost vs. outermost

1. Now consider evaluating \( \text{head ones} \) using innermost reduction:

\[
\text{head ones} = \text{head} (1 : \text{ones}) \\
= \text{head} (1 : 1 : \text{ones}) \\
= \text{head} (1 : 1 : 1 : \text{ones}) \\
= \ldots
\]

Does not terminate!
Innermost vs. outermost

1. Now consider evaluating \textit{head ones} using innermost reduction:

\begin{align*}
\text{head ones} &= \text{head} \ (1 : \text{ones}) \\
&= \text{head} \ (1 : 1 : \text{ones}) \\
&= \text{head} \ (1 : 1 : 1 : \text{ones}) \\
&= \ldots
\end{align*}

Does not terminate!

2. And with outermost reduction:

\begin{align*}
\text{head ones} &= \text{head} \ (1 : \text{ones}) \\
&= 1
\end{align*}

Terminates!
We now see that

\[ \textit{ones} = 1 : \textit{ones} \]

really defines a \textbf{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

- \textit{take 5 ones} = [1, 1, 1, 1, 1]
- \textit{take 5 [1...]} = [1, 2, 3, 4, 5]
Modular programming

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

- \( \text{take 5 ones} = [1, 1, 1, 1, 1] \)
- \( \text{take 5 [1..]} = [1, 2, 3, 4, 5] \)

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

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

Consider

\[ replicate :: \text{Int} \rightarrow a \rightarrow [a] \]
Modular programming: Example

Consider

\[
\text{replicate} :: \text{Int} \rightarrow a \rightarrow [a]
\]

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

\[
\text{replicate} \ 0 \ v = [] \\
\text{replicate} \ n \ v = v : \text{replicate} \ (n - 1) \ v
\]
Modular programming: Example

Consider

\[
\text{replicate} :: \text{Int} \to a \to [a]
\]

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

\[
\begin{align*}
\text{replicate} \ 0 \ v &= [] \\
\text{replicate} \ n \ v &= v : \text{replicate} \ (n - 1) \ v
\end{align*}
\]

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

\[
\text{repeat} :: a \to [a]
\]

\[
\text{repeat} \ v = vs \text{ where } vs = v : vs
\]

and define \text{replicate} by

\[
\text{replicate} \ n = \text{take} \ n \circ \text{repeat}
\]
Primes: The sieve of Eratosthenes

primes :: [Integer]
primes = sieve [2 ..]

seive :: [Integer] → [Integer]
seive (p : xs) = p : seive [x | x ← xs, x `mod` p ≠ 0]

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

\[\text{fibs} :: [\text{Integer}]\]
\[\text{fibs} = 0 : 1 : \text{zipWith (+) \text{fibs} (\text{tail \text{fibs})}}\]
**Cyclic structures**

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

\[
\begin{align*}
\text{ones} &= \text{forever} \ 1 \\
\text{forever} \ x &= x : \text{forever} \ x \\
\text{ones'} &= \text{forever'} \ 1 \\
\text{forever'} \ x &= zs \text{ where } zs = x : zs
\end{align*}
\]
Cyclic structures

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

\[
\begin{align*}
\text{ones} & = \text{forever} \ 1 \\
\text{forever} \ x & = x : \text{forever} \ x \\
\text{ones}' & = \text{forever}' \ 1 \\
\text{forever}' \ x & = zs \ \text{where} \ zs = x : zs
\end{align*}
\]
Cyclic structures

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

\[
\begin{align*}
\text{ones} & = \text{forever} \ 1 \\
\text{forever} \ x & = x : \text{forever} \ x \\
\text{ones'} & = \text{forever'} \ 1 \\
\text{forever'} \ x & = zs \ \text{where} \ zs = x : zs
\end{align*}
\]
Cyclic structures

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

\[
\begin{align*}
\text{ones} & = \text{forever } 1 \\
\text{forever } x & = x : \text{forever } x \\
\text{ones'} & = \text{forever' } 1 \\
\text{forever' } x & = zs \text{ where } zs = x : zs
\end{align*}
\]
The efficiency of \( \text{iterate} \)

\[
\text{iterate } f \; x = x : \text{iterate } f \; (f \; x)
\]
The efficiency of \textit{iterate}

\[
\text{iterate}' \ f \ x = x : \text{map} \ f \ (\text{iterate}' \ f \ x)
\]

Demo with :sprint.
Laziness for memoization

\[ \text{fibs} = \text{map fib'} [0 \ldots] \]
\[ \text{fib'} 0 = 0 \]
\[ \text{fib'} 1 = 1 \]
\[ \text{fib'} n = \text{fib} (n - 1) + \text{fib} (n - 2) \]
\[ \text{fib} n = \text{fibs} !! n \]

> fib 20000
2531162323732361242240155...
(2.01 secs, 21312464 bytes)
> fib 20000
2531162323732361242240155...
(0.02 secs, 4668504 bytes)
Laziness for memoization

\[ \text{fibs} = \text{map fib'} [0..] \]
\[ \text{fib'} 0 = 0 \]
\[ \text{fib'} 1 = 1 \]
\[ \text{fib'} n = \text{fib} (n - 1) + \text{fib} (n - 2) \]
\[ \text{fib} n = \text{fibs} !! n \]

*Main> :sprint fibs
fibs = _
*Main> take 1 fibs
[0]
*Main> :sprint fibs
fibs = 0 : _
*Main> fib 10
55
*Main> :sprint fibs
*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**

\[
\begin{align*}
\text{foldr } f \ z \ [\ ] & = z \\
\text{foldr } f \ z \ (x : xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

\[
\text{foldr } (+) \ 0 \ (1 : 2 : 3 : [])
\]
**Summing up numbers**

\[
\text{foldr } f \ z \ [\ ] = z \\
\text{foldr } f \ z \ (x : xs) = f \ x \ (\text{foldr } f \ z \ xs)
\]

\[
\text{foldr } (+) \ 0 \ (1 : 2 : 3 : []) = 1 + \text{foldr } (+) \ 0 \ (2 : 3 : [])
\]
Summing up numbers

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

\[foldr \ (+) \ 0 \ (1 : 2 : 3 : [\ ]) = 1 + foldr \ (+) \ 0 \ (2 : 3 : [\ ])\]
\[= 1 + (2 + foldr \ (+) \ 0 \ (3 : [\ ]))\]
Summing up numbers

\[ \text{foldr}\ f\ z\ [ ] = z \]
\[ \text{foldr}\ f\ z\ (x:xs) = f\ x\ (\text{foldr}\ f\ z\ xs) \]

\[ \text{foldr}\ (+)\ 0\ (1:2:3:[]) = 1 + \text{foldr}\ (+)\ 0\ (2:3:[]) \]
\[ = 1 + (2 + \text{foldr}\ (+)\ 0\ (3:[])) \]
\[ = 1 + (2 + (3 + \text{foldr}\ (+)\ 0\ [])) \]
Summing up numbers

\[ \text{foldr } f \text{ } z \text{ } [] = z \]
\[ \text{foldr } f \text{ } z \text{ } (x : xs) = f \text{ } x \text{ } (\text{foldr } f \text{ } z \text{ } xs) \]

\[ \text{foldr } (+) \text{ } 0 \text{ } (1 : 2 : 3 : []) = 1 + \text{foldr } (+) \text{ } 0 \text{ } (2 : 3 : []) \]
\[ = 1 + (2 + \text{foldr } (+) \text{ } 0 \text{ } (3 : [])) \]
\[ = 1 + (2 + (3 + \text{foldr } (+) \text{ } 0 \text{ } [])) \]
\[ = 1 + (2 + (3 + 0)) \]

Overflow for large lists!
Summing up numbers

\[ \text{foldr } f \ z \ [\ ] = z \]
\[ \text{foldr } f \ z \ (x : xs) = f \ x \ (\text{foldr } f \ z \ xs) \]

\[ \text{foldr } (+) \ 0 \ (1 : 2 : 3 : []) = 1 + \text{foldr } (+) \ 0 \ (2 : 3 : []) \]
\[ = 1 + (2 + \text{foldr } (+) \ 0 \ (3 : [])) \]
\[ = 1 + (2 + (3 + \text{foldr } (+) \ 0 \ []) ) \]
\[ = 1 + (2 + (3 + 0)) \]
\[ = 1 + (2 + 3) \]
Summing up numbers

\[ \text{foldr} \ f \ z \ [ ] = z \]
\[ \text{foldr} \ f \ z \ (x : xs) = f \ x \ (\text{foldr} \ f \ z \ xs) \]

\[ \text{foldr} \ (+) \ 0 \ (1 : 2 : 3 : [ ]) = 1 + \text{foldr} \ (+) \ 0 \ (2 : 3 : [ ]) \]
\[ = 1 + (2 + \text{foldr} \ (+) \ 0 \ (3 : [ ])) \]
\[ = 1 + (2 + (3 + \text{foldr} \ (+) \ 0 \ [ ])) \]
\[ = 1 + (2 + (3 + 0)) \]
\[ = 1 + (2 + 3) \]
\[ = 1 + 5 \]
Summing up numbers

\[
\text{foldr } f \ z \ [\ ] = z
\]
\[
\text{foldr } f \ z \ (x : xs) = f \ x \ (\text{foldr } f \ z \ xs)
\]

\[
\text{foldr } (+) \ 0 \ (1 : 2 : 3 : [\ ]) = 1 + \text{foldr } (+) \ 0 \ (2 : 3 : [\ ])
\]
\[
= 1 + (2 + \text{foldr } (+) \ 0 \ (3 : [\ ]))
\]
\[
= 1 + (2 + (3 + \text{foldr } (+) \ 0 \ [\ ]))
\]
\[
= 1 + (2 + (3 + 0))
\]
\[
= 1 + (2 + 3)
\]
\[
= 1 + 5
\]
\[
= 6
\]

\(O(n)\) space! Overflow for large lists!
Summing up numbers

\[
\text{foldl } f \ z \ [\ ] = z
\]
\[
\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]

\[
\text{foldl } (+) \ 0 \ (1 : 2 : 3 : [])
\]
Summing up numbers

\[ \text{foldl } f \; z \; [] = z \]
\[ \text{foldl } f \; z \; (x : xs) = \text{foldl } f \; (f \; z \; x) \; xs \]

\[ \text{foldl} \; (+) \; 0 \; (1 : 2 : 3 : []) = \text{foldl} \; (+) \; (0 + 1) \; (2 : 3 : []) \]
Summing up numbers

\[ \text{foldl } f \ z \ [ ] = z \]
\[ \text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs \]

\[ \text{foldl } (+) \ 0 \ (1 : 2 : 3 : []) = \text{foldl } (+) \ (0 + 1) \ (2 : 3 : []) \]
\[ = \text{foldl } (+) \ ((0 + 1) + 2) \ (3 : []) \]
Summing up numbers

\[
\text{foldl } f \ z \ [] = z \\
\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]

\[
\text{foldl } (+) \ 0 \ (1 : 2 : 3 : []) = \text{foldl } (+) \ (0 + 1) \ (2 : 3 : []) \\
= \text{foldl } (+) \ ((0 + 1) + 2) \ (3 : []) \\
= \text{foldl } (+) \ (((0 + 1) + 2) + 3) \ ([]) 
\]
Summing up numbers

\[
\text{foldl } f \ z \ [] = z \\
\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]

\[
\text{foldl } (+) \ 0 \ (1 : 2 : 3 : []) = \text{foldl } (+) \ (0 + 1) \ (2 : 3 : []) \\
= \text{foldl } (+) \ (((0 + 1) + 2) \ 3 : []) \\
= \text{foldl } (+) \ (((0 + 1) + 2) + 3) \ [] \\
= ((0 + 1) + 2) + 3
\]
Summing up numbers

\[
\text{foldl } f \ z \ [ ] = z \\
\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]

\[
\text{foldl } (+) \ 0 \ (1 : 2 : 3 : [ ]) = \text{foldl } (+) \ ((0 + 1) \ (2 : 3 : [ ])) \\
= \text{foldl } (+) \ (((0 + 1) + 2) \ (3 : [ ])) \\
= \text{foldl } (+) \ (((0 + 1) + 2) + 3) \ ([ ]) \\
= ((0 + 1) + 2) + 3 \\
= (1 + 2) + 3
\]
Summing up numbers

\[
\text{foldl } f \; z \; [] = z \\
\text{foldl } f \; z \; (x : xs) = \text{foldl } f \; (f \; z \; x) \; xs
\]

\[
\text{foldl } (+) \; 0 \; (1 : 2 : 3 : []) = \text{foldl } (+) \; (0 + 1) \; (2 : 3 : []) \\
= \text{foldl } (+) \; (((0 + 1) + 2) \; 3 : []) \\
= \text{foldl } (+) \; ((((0 + 1) + 2) + 3) \; []) \\
= ((0 + 1) + 2) + 3 \\
= (1 + 2) + 3 \\
= 3 + 3
\]
Summing up numbers

\[
\text{foldl } f \ z \ [\ ] = z
\]
\[
\text{foldl } f \ z \ (x : xs) = \text{foldl } f \ (f \ z \ x) \ xs
\]

\[
\text{foldl } (+) \ 0 \ (1 : 2 : 3 : [\ ]) = \text{foldl } (+) \ (0 + 1) \ (2 : 3 : [\ ])
\]
\[
= \text{foldl } (+) \ (((0 + 1) + 2) \ 3 : [\ ])
\]
\[
= \text{foldl } (+) \ (((0 + 1) + 2) + 3) \ [\ ]
\]
\[
= ((0 + 1) + 2) + 3
\]
\[
= (1 + 2) + 3
\]
\[
= 3 + 3
\]
\[
= 6
\]

\(O(n)\) space again! What do we do?
Forcing evaluation

Haskell has the following primitive function

\[ \text{seq} :: a \rightarrow b \rightarrow b \quad --\text{primitive} \]

The call

\[ \text{seq} \; x \; y \]

evaluates \( x \) before returning \( y \).

The function \( \text{seq} \) can be used to define strict function application:

\[
(\_!) :: (a \rightarrow b) \rightarrow a \rightarrow b \\
f \_! \; x = x \; \text{`seq'} \; f \; x
\]
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?

\[
(\lambda x \rightarrow ()) \mathbin{!} \bot =
\]

\[
\text{seq} (\bot, \bot) () =
\]

\[
\text{snd} \mathbin{!} (\bot, \bot) =
\]

\[
(\lambda x \rightarrow ()) \mathbin{!} (\lambda x \rightarrow \bot) =
\]

\[
\text{length} \mathbin{!} \text{map} \ \bot [1, 2] =
\]

\[
\text{seq} (\bot + \bot) () =
\]

\[
\text{seq} (\text{foldr} \ \bot \ \bot) () =
\]

\[
\text{seq} (1 : \bot) () =
\]
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?

\[(\lambda x \to () \) \ unfold \ (\bot) = \bot\]
\[seq (\bot, \bot) () = \]
\[snd \ unfold (\bot, \bot) = \]
\[(\lambda x \to () \ unfold (\lambda x \to \bot) = \]
\[length \ unfold \ (map \ (\bot) \ [1, 2] = \]
\[seq (\bot + \bot) () = \]
\[seq (foldr \ (\bot, \bot) () = \]
\[seq (1 : \bot) () = \]

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?

\[(\lambda x \to ()) \!\! \bot \quad = \bot\]
\[seq (\bot, \bot) () \quad = ()\]
\[snd \!\! (\bot, \bot) \quad = \]
\[(\lambda x \to ()) \!\! \bot (\lambda x \to \bot) \quad = \]
\[length \!\! \map \bot [1, 2] \quad = \]
\[seq (\bot + \bot) () \quad = \]
\[seq (\text{foldr} \bot \bot) () \quad = \]
\[seq (1 : \bot) () \quad = \]
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?

\[
\begin{align*}
(\lambda x \to ()) \downarrow & \perp = \perp \\
\text{seq} (\perp, \perp) () & = () \\
\text{snd} \downarrow (\perp, \perp) & = \perp \\
(\lambda x \to ()) \downarrow (\lambda x \to \perp) & = \\
\text{length} \downarrow \text{map} \perp [1, 2] & = \\
\text{seq} (\perp + \perp) () & = \\
\text{seq} (\text{foldr} \perp \perp) () & = \\
\text{seq} (1 : \perp) () & = \\
\end{align*}
\]
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?

\[(\lambda x \to ()) \downarrow \bot \quad = \bot\]
\[seq (\bot, \bot) () \quad = ()\]
\[snd \uparrow (\bot, \bot) \quad = \bot\]
\[(\lambda x \to ()) \downarrow (\lambda x \to \bot) = ()\]
\[length \uparrow map \bot [1, 2] =\]
\[seq (\bot + \bot) () \quad =\]
\[seq (foldr \bot \bot) () \quad =\]
\[seq (1 : \bot) () \quad =\]
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?

\[(\lambda x \to \text{()}) \; \mid \; \bot = \bot\]
\[\text{seq} \; (\bot, \bot) \; () = ()\]
\[\text{snd} \; \mid \; \bot, \bot = \bot\]
\[(\lambda x \to \text{()}) \; \mid \; (\lambda x \to \bot) = ()\]
\[\text{length} \; \mid \; \text{map} \; \bot \; [1, 2] = 2\]
\[\text{seq} \; (\bot + \bot) \; () = \]
\[\text{seq} \; (\text{foldr} \; \bot \; \bot) \; () = \]
\[\text{seq} \; (1 : \bot) \; () = \]
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?

\[
(\lambda x \rightarrow ()) \Downarrow \bot = \bot \\
seq (\bot, \bot) () = () \\
snd \Downarrow (\bot, \bot) = \bot \\
(\lambda x \rightarrow ()) \Downarrow (\lambda x \rightarrow \bot) = () \\
length \Downarrow map \bot [1, 2] = 2 \\
seq (\bot + \bot) () = \bot \\
seq (foldr \bot \bot) () = \\
seq (1 : \bot) () = \\
\]
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?

\[(\lambda x \rightarrow ()) \downarrow \perp = \perp\]
\[\text{seq} (\perp, \perp) () = ()\]
\[\text{snd} \downarrow (\perp, \perp) = \perp\]
\[(\lambda x \rightarrow ()) \downarrow (\lambda x \rightarrow \perp) = ()\]
\[\text{length} \downarrow \text{map} \perp [1, 2] = 2\]
\[\text{seq} (\perp + \perp) () = \perp\]
\[\text{seq} (\text{foldr} \perp \perp) () = ()\]
\[\text{seq} (1 : \perp) () = \]
Forcing only evaluates to weak head normal form (i.e., a lambda abstraction, literal or constructor application)

What does the following expressions evaluate to?

\[
\begin{align*}
(\lambda x \to ()) \text{! } \bot & = \bot \\
\text{seq } (\bot, \bot) () & = () \\
\text{snd } \text{! } (\bot, \bot) & = \bot \\
(\lambda x \to ()) \text{! } (\lambda x \to \bot) & = () \\
\text{length } \text{! } \text{map } \bot [1, 2] & = 2 \\
\text{seq } (\bot + \bot) () & = \bot \\
\text{seq } (\text{foldr } \bot \bot) () & = () \\
\text{seq } (1 : \bot) () & = ()
\end{align*}
\]
Summing up numbers

\[
\begin{align*}
\text{foldl}' \; f \; z \; [] & = z \\
\text{foldl}' \; f \; z \; (x : xs) & = \text{let } z' = f \; z \; x \text{ in } z' \; '\text{seq}' \; (\text{foldl} \; f \; z' \; xs) \\
\text{foldl}' \; (+) \; 0 \; (1 : 2 : 3 : []) & = 6
\end{align*}
\]
Summing up numbers

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

\[
\text{foldl' } (+) \ 0 \ (1 : 2 : 3 : [\ ])
= \text{let } z' = (0 + 1) \ \text{in } z' \ 'seq' \ \text{foldl' } (+) \ z' \ (2 : 3 : [\ ])
\]
Summing up numbers

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

\[ \text{foldl'} \ (+) \ 0 \ (1 : 2 : 3 : []) \]
\[ = \textbf{let} \ z' = (0 + 1) \ \textbf{in} \ z' \ 'seq' \ \text{foldl'} \ (+) \ z' \ (2 : 3 : []) \]
\[ = \text{foldl'} \ (+) \ 1 \ (2 : 3 : []) \]
Summing up numbers

```
foldl' f z [] = z
foldl' f z (x:xs) = let z' = f z x in z' 'seq' (foldl f z' xs)
```

```
foldl' (+) 0 (1:2:3:[]) =  
  let z' = (0+1) in z' 'seq' foldl' (+) z' (2:3:[])  
  = foldl' (+) 1 (2:3:[])  
  = let z' = (1+2) in z' 'seq' foldl' (+) z' (3:[])  
```
Summing up numbers

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

\[
\text{foldl}' (+) 0 (1:2:3:[]) \\
= \text{let } z' = (0+1) \text{ in } z' \ 'seq' \ \text{foldl}' (+) z' (2:3:[]) \\
= \text{foldl}' (+) 1 (2:3:[]) \\
= \text{let } z' = (1+2) \text{ in } z' \ 'seq' \ \text{foldl}' (+) z' (3:[]) \\
= \text{foldl}' (+) 3 (3:[]) \\
\]
Summing up numbers

\[
\begin{align*}
\text{foldl'} f z [] &= z \\
\text{foldl'} f z (x:xs) &= \text{let } z' = f z x \text{ in } z' \text{'seq'} (\text{foldl } f z' xs)
\end{align*}
\]

\[
\begin{align*}
\text{foldl'} (+) 0 (1:2:3:[]) &= \text{let } z' = (0+1) \text{ in } z' \text{'seq'} \text{foldl'} (+) z' (2:3:[]) \\
&= \text{foldl'} (+) 1 (2:3:[]) \\
&= \text{let } z' = (1+2) \text{ in } z' \text{'seq'} \text{foldl'} (+) z' (3:[]) \\
&= \text{foldl'} (+) 3 (3:[]) \\
&= \text{let } z' = (3+3) \text{ in } z' \text{'seq'} \text{foldl'} (+) z' (3:[])
\end{align*}
\]
Summing up numbers

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

\[
foldl' \ (+) \ 0 \ (1:2:3:[]) = \text{let } z' = (0 + 1) \ \text{in } z' \ 'seq' \ foldl' \ (+) \ z' \ (2:3:[])
\]
\[
= foldl' \ (+) \ 1 \ (2:3:[])
\]
\[
= \text{let } z' = (1 + 2) \ \text{in } z' \ 'seq' \ foldl' \ (+) \ z' \ (3:[])
\]
\[
= foldl' \ (+) \ 3 \ (3:[])
\]
\[
= \text{let } z' = (3 + 3) \ \text{in } z' \ 'seq' \ foldl' \ (+) \ z' \ (3:[])
\]
\[
= foldl' \ (+) \ 6 \ []
\]
Summing up numbers

\[
\begin{align*}
    \text{foldl}' \ f \ z \ [ ] &= z \\
    \text{foldl}' \ f \ z \ (x : xs) &= \text{let} \ z' = f \ z \ x \ \text{in} \ z' \ 'seq' \ (\text{foldl} \ f \ z' \ xs)
\end{align*}
\]

\[
\begin{align*}
    \text{foldl}' \ (+) \ 0 \ (1 : 2 : 3 : []) &= \\
    &= \text{let} \ z' = (0 + 1) \ \text{in} \ z' \ 'seq' \ \text{foldl}' \ (+) \ z' \ (2 : 3 : []) \\
    &= \text{foldl}' \ (+) \ 1 \ (2 : 3 : []) \\
    &= \text{let} \ z' = (1 + 2) \ \text{in} \ z' \ 'seq' \ \text{foldl}' \ (+) \ z' \ (3 : []) \\
    &= \text{foldl}' \ (+) \ 3 \ (3 : []) \\
    &= \text{let} \ z' = (3 + 3) \ \text{in} \ z' \ 'seq' \ \text{foldl}' \ (+) \ z' \ (3 : []) \\
    &= \text{foldl}' \ (+) \ 6 \ [] \\
    &= 6
\end{align*}
\]

\(O(1)\) space.