Functional Programming in Haskell
Part 2: Abstract datatypes and “infinite” structures

Madhavan Mukund
Chennai Mathematical Institute
92 G N Chetty Rd, Chennai 600 017, India
madhavan@cmi.ac.in
http://www.cmi.ac.in/~madhavan
Program ≜ Collection of function definitions

isDigit :: Char → Bool

isDigit '0' = True
isDigit '1' = True
...isDigit '9' = True
isDigit c = False

| (c >= '0' && c <= '9') = True |
| otherwise = False |
Haskell review

- Program ≡ Collection of function definitions
- Computation ≡ Rewriting using definitions

```haskell
isDigit :: Char -> Bool
isDigit '0' = True
isDigit '1' = True
...isDigit '9' = True
isDigit c = False
```

| (c >= '0' && c <= '9') = True |
| otherwise = False |

Madras Christian College, 10 December 2003 – p.2
Haskell review

- Program $\equiv$ Collection of function definitions
- Computation $\equiv$ Rewriting using definitions
- Functions are associated with input and output types

```haskell
isDigit :: Char -> Bool
isDigit '0' = True
isDigit '1' = True
...isDigit '9' = True
isDigit c = False

| (c >= '0' && c <= '9') = True |
| otherwise = False |
```
Haskell review

- Program ≡ Collection of function definitions
- Computation ≡ Rewriting using definitions
- Functions are associated with input and output types

```haskell
isDigit :: Char -> Bool
isDigit c = (c >= '0' && c <= '9') || (c >= 'a' && c <= 'z')
```

Madras Christian College, 10 December 2003 – p.2
Haskell review

- Program ≡ Collection of function definitions
- Computation ≡ Rewriting using definitions
- Functions are associated with input and output types

```haskell
isDigit :: Char -> Bool
isDigit '0' = True
isDigit '1' = True
...
isDigit '9' = True
isDigit c = False
```
Haskell review

- Program ≡ Collection of function definitions
- Computation ≡ Rewriting using definitions
- Functions are associated with input and output types

```haskell
isDigit :: Char -> Bool
isDigit '0' = True
isDigit '1' = True
...
isDigit '9' = True
isDigit c = False
```

```haskell
isDigit c
| (c >= '0' && c <= '9') = True
| otherwise = False
```
Basic collective type is a list

\[
\text{Example}
\]

Adding up a list of integers

\[
\text{sum} :: \{\text{Int}\} \rightarrow \text{Int}
\]

\[
\text{sum} [] = 0
\]

\[
\text{sum} (x: l) = x + (\text{sum} l)
\]

(Conditional) polymorphism

Most general type of \(\text{sum}\) is

\[
\text{sum} :: (\text{Num} a) \Rightarrow \{a\} \rightarrow a
\]

where \(\text{Num} a\) is true for any type \(a\) that supports basic arithmetic operations \(+, -\), . . .
Basic collective type is a list

Define list functions by induction on structure

Example

Adding up a list of integers

```
sum :: [Int] -> Int
sum [] = 0
sum (x:l) = x + (sum l)
```

(Conditional) polymorphism

Most general type of `sum` is

```
sum :: (Num a) => [a] -> a
```

where `Num a` is true for any type `a` that supports basic arithmetic operations `+`, `-`, . . .
Basic collective type is a list

Define list functions by induction on structure

Example Adding up a list of integers
Basic collective type is a list

Define list functions by induction on structure

**Example** Adding up a list of integers

\[
\text{sum} :: \ [\text{Int}] \rightarrow \text{Int} \\
\text{sum} \ [\ ] = 0 \\
\text{sum} \ (x:1) = x + (\text{sum} \ l)
\]
Basic collective type is a list

Define list functions by induction on structure

**Example** Adding up a list of integers

```haskell
sum :: [Int] -> Int
sum [] = 0
sum (x:l) = x + (sum l)
```

(Conditional) polymorphism
Basic collective type is a list
Define list functions by induction on structure

Example Adding up a list of integers

```
sum :: [Int] -> Int
sum [] = 0
sum (x:l) = x + (sum l)
```

(Conditional) polymorphism

Most general type of `sum` is

```
sum :: (Num a) => [a] -> a
```

where `Num a` is true for any type `a` that supports basic arithmetic operations `+`, `−`, ...
Today’s agenda

- Adding new types
Today’s agenda

- Adding new types
- Defining *abstract datatypes*

Provide an interface that “hides” the implementation
Today’s agenda

- Adding new types
- Defining **abstract datatypes**
  
  Provide an interface that “hides” the implementation
- Using “infinite” data structures
The `data` declaration adds new datatypes

- Enumerated types
  
  ```e
  data Signal = Red | Yellow | Green
  ```
  
  Can use this type in a function such as
  
  ```hs
  stopwhen :: Signal -> Bool
  stopwhen Red = True
  stopwhen c = False
  ```

  What if we write instead
  
  ```hs
  stopwhen2 :: Signal -> Bool
  stopwhen2 c | (c == Red) = True
               | otherwise = False
  ```
User defined datatypes

- The `data` declaration adds new datatypes
- Enumerated types

```haskell
data Signal = Red | Yellow | Green
```

User defined datatypes

- The `data` declaration adds new datatypes
- Enumerated types
  
  ```haskell
data Signal = Red | Yellow | Green
```
- Can use this type in a function such as
  
  ```haskell
stopwhen :: Signal -> Bool
stopwhen Red = True
stopwhen c = False
```
User defined datatypes

- The `data` declaration adds new datatypes
  
  **Enumerated types**
  
  ```haskell
  data Signal = Red | Yellow | Green
  ```
  
  Can use this type in a function such as
  
  ```haskell
  stopwhen :: Signal -> Bool
  stopwhen Red = True
  stopwhen c   = False
  ```
  
  - What if we write instead
    
    ```haskell
    stopwhen2 :: Signal -> Bool
    stopwhen2 c  | (c == Red) = True
                | otherwise = False
    ```
stopwhen2 requires \texttt{Eq \ Signal}

How about

\begin{verbatim}
nextlight \texttt{Green} = \texttt{Yellow}
nextlight \texttt{Yellow} = \texttt{Red}
nextlight \texttt{Red} = \texttt{Green}
\end{verbatim}

Displaying result of \texttt{nextlight} requires \texttt{Show \ Signal}

\texttt{Show} a \texttt{is} true of type \texttt{a} if there is a function \texttt{show :: a -> String} that allows values of \texttt{a} to be displayed
Stop when requires Eq Signal

How about

nextlight :: Signal -> Signal

nextlight Green = Yellow
nextlight Yellow = Red
nextlight Red = Green
User defined types and type classes

- stopwhen2 requires Eq Signal

- How about

  nextlight :: Signal -> Signal
  nextlight Green = Yellow
  nextlight Yellow = Red
  nextlight Red = Green

- Displaying result of nextlight requires Show Signal
User defined types and type classes

- `stopwhen2 requires Eq Signal`

- How about
  
  ```haskell
  nextlight :: Signal -> Signal
  nextlight Green = Yellow
  nextlight Yellow = Red
  nextlight Red = Green
  ```

- Displaying result of `nextlight` requires `Show Signal`

- `Show a` is true of type `a` if there is a function
  
  ```haskell
  show :: a -> String
  ```

  that allows values of `a` to be displayed
Simplest solution is

```haskell
data Signal = Red | Yellow | Green
  deriving (Eq, Show, Ord)
```
Adding user defined types to type classes

- Simplest solution is
  \[
  \text{data Signal} = \text{Red} \mid \text{Yellow} \mid \text{Green}
  \]
  deriving (Eq, Show, Ord)

- Fixes default values
Adding user defined types to type classes

- Simplest solution is
  ```haskell
data Signal = Red | Yellow | Green
deriving (Eq, Show, Ord)
```
- Fixes default values
  ```haskell
  Red == Red, Red /= Yellow,
  ```
Adding user defined types to type classes

- Simplest solution is
  ```haskell
data Signal = Red | Yellow | Green
  deriving (Eq, Show, Ord)
```
- Fixes default values
  - `Red == Red, Red /= Yellow, ...`
  - `show Red = "Red", show Yellow = "Yellow", ...`
Simplest solution is

```haskell
data Signal = Red | Yellow | Green
deriving (Eq, Show, Ord)
```

Fixes default values

- Red == Red, Red /= Yellow, ...
- show Red = "Red",
  show Yellow = "Yellow", ...
- Red < Yellow < Green
data Signal = Red | Yellow | Green

instance Eq Signal where
  c == c = True

instance Show Signal where
  show Yellow = "Yellow"
  show c = "Black"

instance Ord Signal where
  Green <= Yellow = True
  Yellow <= Red = True
  Red <= Green = True
  x <= y = False

In the class `Ord`, >, >=, . . . are defined in terms of <=

Note: <= need not even be a consistent ordering!

Or, provide your own functions
Or, provide your own functions

```haskell
data Signal = Red | Yellow | Green
deriving (Eq)
```

In the class `Ord`, `>`, `>=`, . . . are defined in terms of `<=`.

Note: `<=` need not even be a consistent ordering!
Or, provide your own functions

data Signal = Red | Yellow | Green
deriving (Eq)

instance Show Signal where
  show Yellow = "Yellow"
  show c = "Black"
Or, provide your own functions

data Signal = Red | Yellow | Green
  deriving (Eq)

instance Show Signal where
  show Yellow = "Yellow"
  show c = "Black"

instance Ord Signal where
  Green <= Yellow = True
  Yellow <= Red = True
  Red <= Green = True
  x <= y = False
Adding user defined types to type classes . . .

Or, provide your own functions

```haskell
data Signal = Red | Yellow | Green
  deriving (Eq)

instance Show Signal where
  show Yellow = "Yellow"
  show c = "Black"

instance Ord Signal where
  Green <= Yellow = True
  Yellow <= Red = True
  Red <= Green = True
  x <= y = False
```

In the class `Ord`, `>`, `>=`, . . . are defined in terms of `<=`
Or, provide your own functions

```haskell
data Signal = Red | Yellow | Green
    deriving (Eq)

instance Show Signal where
    show Yellow = "Yellow"
    show c = "Black"

instance Ord Signal where
    Green <= Yellow = True
    Yellow <= Red = True
    Red <= Green = True
    x <= y = False
```

In the class `Ord`, `>`, `>=`, ... are defined in terms of `<=`

Note: `<=` need not even be a consistent ordering!
Recursive datatypes

- A binary tree to store integers at each node

```haskell
data Btreeint = Nil | Node Int Btreeint Btreeint
```
A binary tree to store integers at each node

```haskell
data Btreeint =
  Nil |
  Node Int Btreeint Btreeint
```

Nil and Node are constructors
Recursive datatypes

- A binary tree to store integers at each node

```haskell
data Btreeint =
  Nil |
  Node Int Btreeint Btreeint
```

- `Nil` and `Node` are **constructors**
- The constructor `Nil` takes zero arguments
Recursive datatypes

- A binary tree to store integers at each node
  
  ```hs
  data BtreeInt =
    Nil |
    Node Int BtreeInt BtreeInt
  ```

- Nil and Node are constructors

- The constructor Nil takes zero arguments

- A constant, like Red, Green, ...
Recursive datatypes

- A binary tree to store integers at each node

```haskell
data BtreeInt = 
  Nil |
  Node Int BtreeInt BtreeInt
```

- Nil and Node are **constructors**
- The constructor Nil takes zero arguments
  A **constant**, like Red, Green, ...
- The constructor Node has three arguments
Recursive datatypes

- A binary tree to store integers at each node

```haskell
data BtreeInt = Nil | Node Int BtreeInt BtreeInt
```

- Nil and Node are **constructors**

- The constructor `Nil` takes zero arguments

- The constructor `Node` has three arguments
  - First component `Int`: value stored at the node
Recursive datatypes

- A binary tree to store integers at each node

```haskell
data Btreeint =
  Nil |
  Node Int Btreeint Btreeint
```

- **Nil** and **Node** are constructors
- The constructor **Nil** takes zero arguments
  - A constant, like **Red**, **Green**, ...
- The constructor **Node** has three arguments
  - First component **Int**: value stored at the node
  - Other two components **Btreeint**: left and right subtrees
Recursive datatypes ...

Example

The tree

would be written as

Node 6 (Node 4 Nil Nil)
(Node 8 (Node 7 Nil Nil) Nil)
Functions on recursive datatypes

- Define by induction on the structure of datatype
Functions on recursive datatypes

- Define by induction on the structure of datatype
- How many values are there in the tree?

size :: Btreeint -> Int
size Nil = 0
size (Node n t1 t2) = 1 + (size t1) + (size t2)

listout :: Btreeint -> [Int]
listout Nil = []
listout (Node n t1 t2) = [n] ++ listout t1 ++ listout t2
Functions on recursive datatypes

- Define by induction on the structure of datatype
- How many values are there in the tree?

\[
\text{size} :: \text{Btree} \rightarrow \text{Int} \\
\text{size \ Nil} = 0 \\
\text{size (Node \ n \ t1 \ t2)} = 1 + (\text{size \ t1}) + (\text{size \ t2})
\]
Functions on recursive datatypes

- Define by induction on the structure of datatype

- How many values are there in the tree?

  \[\text{size} :: \text{Btreeint} \rightarrow \text{Int}\]
  \[\text{size \ Nil} = 0\]
  \[\text{size (Node n t1 t2)} = 1 + (\text{size t1}) + (\text{size t2})\]

- List out all values in the tree
Functions on recursive datatypes

- Define by induction on the structure of datatype
- How many values are there in the tree?

\[
\text{size} :: \text{Btreeint} \rightarrow \text{Int} \\
\text{size } \text{Nil} = 0 \\
\text{size } (\text{Node } n \ t1 \ t2) = 1 + (\text{size } t1) + (\text{size } t2)
\]

- List out all values in the tree

\[
\text{listout} :: \text{Btreeint} \rightarrow [\text{Int}] \\
\text{listout } \text{Nil} = [] \\
\text{listout } (\text{Node } n \ t1 \ t2) = [n] ++ \text{listout } t1 ++ \text{listout } t2
\]
A binary tree to store arbitrary values at each node?

```haskell
data Btree a =
  Nil |
  Node a (Btree a) (Btree a)
```
Polymorphic recursive datatypes

- A binary tree to store arbitrary values at each node?
  
  ```haskell
  data Btree a =
    Nil |
    Node a (Btree a) (Btree a)
  ```

- What if we want to use `Btree a` as a **search tree**
Polymorphic recursive datatypes

- A binary tree to store arbitrary values at each node?

```haskell
data Btree a =
    Nil | Node a (Btree a) (Btree a)
```

- What if we want to use `Btree a` as a search tree?
  Values in the tree must have a natural ordering
Polymorphic recursive datatypes

- A binary tree to store arbitrary values at each node?
  
  ```haskell
  data Btree a =
    Nil |
    Node a (Btree a) (Btree a)
  ```

- What if we want to use `Btree a` as a search tree? Values in the tree must have a natural ordering.

- Conditional polymorphism!
  
  ```haskell
  (Ord a) => data Btree a =
    Nil |
    Node a (Btree a) (Btree a)
  ```
Built in list type is a polymorphic recursive datatype
Built in list type is a polymorphic recursive datatype

```haskell
data Mylist a = Emptylist | Append a (Mylist a)
```
Polymorphic recursive datatypes . . .

- Built in list type is a polymorphic recursive datatype

```haskell
data Mylist a = Emptylist | Append a (Mylist a)
```

- Since lists are built in, they can use special symbols `[]` and `:` for constructors `Emptylist` and `Append`
Can inherit type classes from underlying type
Can inherit type classes from underlying type

data Btree a =
    Nil |
    Node a (Btree a) (Btree a)
deriving (Eq, Show)
Can inherit type classes from underlying type

```haskell
data Btree a =
  Nil |
  Node a (Btree a) (Btree a)
deriving (Eq, Show)
```

**Note:** Not `Eq (Btree a)`

```haskell
but Eq a => Eq (Btree a)
```
Adding recursive datatypes to type classes

- Can inherit type classes from underlying type

```haskell
data Btree a =
  Nil |
  Node a (Btree a) (Btree a)
deriving (Eq, Show)
```

**Note:** Not `Eq (Btree a)`

- Derived `==` checks that trees have same structure
Can inherit type classes from underlying type

```haskell
data Btree a =
  Nil |
  Node a (Btree a) (Btree a)
deriving (Eq, Show)
```

**Note:** Not `Eq (Btree a)`
but `Eq a => Eq (Btree a)`

- Derived `==` checks that trees have same structure
Adding recursive datatypes to type classes . . .

- Or we can define our own functions
Or we can define our own functions

```haskell
instance (Eq a) => Eq (Btree a) where
t1 == t2 = (listout t1 == listout t2)
```
Or we can define our own functions

```haskell
instance (Eq a) => Eq (Btree a) where
    t1 == t2 = (listout t1 == listout t2)
```

because

```
[6,8] == [6,8]
```
Or we can define our own functions

```
instance (Eq a) => Eq (Btree a) where
t1 == t2 = (listout t1 == listout t2)
```

because

```
[6,8] == [6,8]
```
Or we can define our own functions

```haskell
instance (Eq a) => Eq (Btree a) where
    t1 == t2 = (listout t1 == listout t2)
```

because

\[ [6,8] == [6,8] \]
Declarative programming with abstract datatypes

**Rotate right** — transformation used to balance trees

\[
\text{Rotate right (Node } x (\text{Node } y t_1 t_2) t_3) = \text{Node } y t_1 (\text{Node } x t_2 t_3)
\]
Rotate right — transformation used to balance trees

\[
\text{rotateright} \ (\text{Node } x \ (\text{Node } y \ t_1 \ t_2) \ t_3) \ = \ \text{Node } y \ t_1 \ (\text{Node } x \ t_2 \ t_3)
\]
Example Queues

Queues store sequence of values in FIFO fashion. Append items at the tail of queue. Want a datatype Queue with functions:

- `addq :: (Queue a) -> a -> (Queue a)`
- `removeq :: (Queue a) -> (a, Queue a)`
- `isemptyq :: (Queue a) -> Bool`
Abstract datatypes

- Example Queues
- Stores sequence of values in FIFO fashion
Example Queues

Stores sequence of values in FIFO fashion

Append items at the tail of queue
Abstract datatypes

Example Queues

- Stores sequence of values in FIFO fashion
- Append items at the tail of queue

Want a datatype `Queue a` with functions

```haskell
addq :: (Queue a) -> a -> (Queue a)
removeq :: (Queue a) -> (a, Queue a)
isemptyq :: (Queue a) -> Bool
emptyqueue :: (Queue a)
```
Implement a queue as a list
Implement a queue as a list

```
data Queue a = Qu [a]
```
Implement a queue as a list

data Queue a = Qu [a]

addq :: (Queue a) -> a -> (Queue a)
addq (Qu l) d = Qu (d:l)

removeq :: (Queue a) -> (a,Queue a)
removeq (Qu l)) = (last l,Qu (init l))

isemptyq :: (Queue a) -> Bool
isemptyq (Qu []) = True
isemptyq q = False

emptyqueue :: (Queue a)
emptyqueue = Qu []
Group together the definitions for `Queue a` in a separate reusable `module`.
Group together the definitions for `Queue a` in a separate reusable **module**

```haskell
module Queue where

data Queue a = Qu [a] addq :: (Queue a) -> a -> (Queue a)
...
emptyqueue :: (Queue a) ...
```
Group together the definitions for `Queue a` in a separate reusable `module`.

```
module Queue where

data Queue a = Qu [a] addq :: (Queue a) -> a -> (Queue a)
...
emptyqueue :: (Queue a)...
```

Use these definitions in another file

```
import Queue
```
Group together the definitions for `Queue a` in a separate reusable **module**

```haskell
module Queue where

data Queue a = Qu [a]

addq :: (Queue a) -> a -> (Queue a)

emptyqueue :: (Queue a)...
```

Use these definitions in another file

```haskell
import Queue
```

How do we prevent unauthorized access to a queue?

```haskell
remsec :: (Queue a) -> (a, Queue a)
remsec (Qu (x:y:l)) = (y, Qu (x:l))
```
Restrict visibility outside module
Restrict visibility outside module

```haskell
module Queue (addq, removeq, isemptyq, emptyqueue)
  where ...
```
- Restrict visibility outside module

```
module Queue (addq, removeq, isemptyq, emptyqueue)
where ...
```

- Constructor `Qu` is not visible if you do

```
import Queue
```
Restrict visibility outside module

```haskell
module Queue (addq, removeq, isemptyq, emptyqueue)
    where ...
```

Constructor `Qu` is not visible if you do `import Queue`

Can override imported function with local definition
Restrict visibility outside module

```haskell
module Queue (addq, removeq, isemptyq, emptyqueue)
where ... 
```

Constructor `Qu` is not visible if you do.

```haskell
import Queue
```

Can override imported function with local definition.

All Haskell programs implicitly import `Prelude`
- Restrict visibility outside module
  ```haskell
  module Queue(addq, removeq, isemptyq, emptyqueue)
  where ...
  ```
- Constructor `Qu` is not visible if you do `import Queue`
- Can override imported function with local definition
  All Haskell programs implicitly import `Prelude`
  Redefine builtin functions using `import Prelude hiding (max)`
Restrict visibility outside module

```haskell
module Queue (addq, removeq, isemptyq, emptyqueue)
    where ...
```

Constructor `Qu` is not visible if you do
```haskell
import Queue
```

Can override imported function with local definition

All Haskell programs implicitly import `Prelude`

Redefine builtin functions using
```haskell
import Prelude hiding (max)
```
More than one expression may qualify for rewriting

- \( \sqrt{x} = x \times x \)
- \( \sqrt{4+3} \)
- \( \sqrt{7} \)
- \( 7 \times 7 \)
- \( 49 \)
- \((4+3) \times (4+3)\)
- \((4+3) \times 7\)
- \( 7 \times 7 \)
- \( 49 \)
More than one expression may qualify for rewriting

\[ \text{sqr } x = x \times x \]
More than one expression may qualify for rewriting

- $sqr \ x = x^2$
- $sqr \ (4+3)$
More than one expression may qualify for rewriting

- \( \text{sqr } x = x^2 \)
- \( \text{sqr } (4+3) \)

\[ \text{sqr } 7 \rightarrow 7 \times 7 \rightarrow 49 \]
More than one expression may qualify for rewriting

\[\text{sqr } x = x^2\]

\[\text{sqr } (4+3)\]

\[\approx \text{sqr } 7 \approx 7 \times 7 \approx 49\]

\[\approx (4+3) \times (4+3) \approx (4+3) \times 7 \approx 7 \times 7 \approx 49\]
More than one expression may qualify for rewriting

\[ \text{sqr } x = x^2 \]

\[ \text{sqr } (4+3) \]
\[ \rightarrow \text{sqr } 7 \rightarrow 7 \times 7 \rightarrow 49 \]
\[ \rightarrow (4+3) \times (4+3) \rightarrow (4+3) \times 7 \rightarrow 7 \times 7 \rightarrow 49 \]

If there are multiple expressions to rewrite, Haskell chooses \textit{outermost} expression
More than one expression may qualify for rewriting

\[ \text{\texttt{sqr}} \quad \text{\texttt{x = x}}^2 \text{\texttt{}} \]

\[ \text{\texttt{sqr}} \quad (4+3) \]

\[ \rightarrow \text{\texttt{sqr}} \quad 7 \rightarrow 7 \times 7 \rightarrow 49 \]

\[ \rightarrow (4+3) \times (4+3) \rightarrow (4+3) \times 7 \rightarrow 7 \times 7 \rightarrow 49 \]

If there are multiple expressions to rewrite, Haskell chooses **outermost** expression.

Outermost reduction \( \equiv \) ‘Lazy” rewriting
Evaluate argument to a function only when needed.
Rewriting revisited

- More than one expression may qualify for rewriting
  - `sqr x = x*x`
  - `sqr (4+3)`
    \[ \sim \rightarrow \text{sqr } 7 \sim \rightarrow 7\times7 \sim \rightarrow 49 \]
    \[ \sim \rightarrow (4+3) \times (4+3) \sim \rightarrow (4+3) \times 7 \sim \rightarrow 7\times7 \sim \rightarrow 49 \]
- If there are multiple expressions to rewrite, Haskell chooses \textit{outermost} expression
- Outermost reduction \(\equiv\) ‘Lazy” rewriting
  Evaluate argument to a function only when needed.
- “Eager” rewriting — evaluate arguments before evaluating function
Lazy rewriting

- Haskell evaluates arguments only when needed
Lazy rewriting

- Haskell evaluates arguments only when needed

```haskell
power :: Float -> Int -> Float
power x n = if (n == 0) then 1.0
             else x * (power x (n-1))
```

```haskell
power (8.0/0.0) 0
```

1.0
Haskell evaluates arguments only when needed

```
power :: Float -> Int -> Float

power x n = if (n == 0) then 1.0
             else x * (power x (n-1))

power (8.0/0.0) 0 \sim 1.0
```
Infinite lists!

- The following definition makes sense in Haskell

\[ \text{from } n = n : \text{from } (n+1) \]

Limit is the infinite list \([2,3,4,5,\ldots]\)

Haskell can (and will) generate it incrementally, till you stop it, or it runs out of memory.

Can write \([2.\ldots]\) to denote \([2,3,4,\ldots]\)
\textbf{Infinite lists!}

- The following definition makes sense in Haskell

\[
\text{from } n = n : \text{ from } (n+1)
\]

\textbf{from 2}
The following definition makes sense in Haskell

```
from n = n : from (n+1)
```

```
from 2
\sim 2:(from 3)
\sim 2:(3:(from 4))
\sim 2:(3:(4:(from 5)))
...
```
The following definition makes sense in Haskell

```
from n = n : from (n+1)
```

```
from 2
~> 2:(from 3)
~> 2:(3:(from 4))
~> 2:(3:(4:(from 5)))
...
```

Limit is the infinite list \([2, 3, 4, 5, \ldots]\)
The following definition makes sense in Haskell:

\[ \text{from } n = n : \text{ from } (n+1) \]

\[
\text{from } 2
\]

\[
\leadsto 2 : (\text{from } 3)
\]

\[
\leadsto 2 : (3 : (\text{from } 4))
\]

\[
\leadsto 2 : (3 : (4 : (\text{from } 5)))
\]

\[
\ldots
\]

Limit is the infinite list \([2, 3, 4, 5, \ldots]\).

Haskell can (and will) generate it incrementally, till you stop it, or it runs out of memory.
The following definition makes sense in Haskell

\[
\text{from } n = n : \text{ from } (n+1)
\]

\[
\text{from 2} \quad \Rightarrow 2 : (\text{from 3}) \quad \Rightarrow 2 : (3 : (\text{from 4})) \quad \Rightarrow 2 : (3 : (4 : (\text{from 5})))
\]

\[\ldots\]

Limit is the infinite list \([2, 3, 4, 5, \ldots]\)

Haskell can (and will) generate it incrementally, till you stop it, or it runs out of memory

Can write \([2 \ldots]\) to denote \([2, 3, 4, \ldots]\)
Why infinite lists?

- Can sometimes simplify a problem
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number

Idea: generate all prime numbers and wait for the $n^{th}$ entry

The Sieve of Eratosthenes

Start with $[2,3,4,...]$ and
Transfer smallest number into list of primes and delete all its multiples
Repeat second step forever!
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
- The Sieve of Eratosthenes
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
- The Sieve of Eratosthenes
  - Start with $[2, 3, 4, \ldots]$
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
- The Sieve of Eratosthenes
  - Start with $[2, 3, 4, \ldots ]$
  - Transfer smallest number into list of primes and delete all its multiples
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
- The Sieve of Eratosthenes
  - Start with $[2, 3, 4, \ldots ]$
  - Transfer smallest number into list of primes and delete all its multiples
  - Repeat second step
Why infinite lists?

- Can sometimes simplify a problem
- Consider the problem of computing the $n^{th}$ prime number
- Idea: generate all prime numbers and wait for the $n^{th}$ entry
- The Sieve of Eratosthenes
  - Start with $[2, 3, 4, \ldots]$  
  - Transfer smallest number into list of primes and delete all its multiples
  - Repeat second step forever!
primes = sieve [2..]  
  where sieve (x:l) =  
    x : sieve [y | y <- l, mod y x > 0]
The Sieve of Eratosthenes

primes = sieve [2..]
  where sieve (x:l) =
    x : sieve [y | y <- l, mod y x > 0]

How does this work?
The Sieve of Eratosthenes

\[
\text{primes} = \text{sieve} \ [2..] \\
\text{where sieve} \ (x:l) = \\
x : \ \text{sieve} \ [y \mid y <- l, \ \text{mod} \ y \ x > 0]
\]

How does this work?

\text{sieve} \ [2..]

The Sieve of Eratosthenes

primes = sieve [2..]
    where sieve (x:l) =
        x : sieve [y | y <- l, mod y x > 0]

How does this work?

sieve [2..]
〜2:sieve [y | y <- [3..], mod y 2 > 0]
The Sieve of Eratosthenes

primes = sieve [2..]
    where sieve (x:l) =
        x : sieve [y | y <- l, mod y x > 0]

How does this work?

sieve [2..]
⇒ 2:sieve [y | y <- [3..], mod y 2 > 0]
⇒ 2:sieve (3:[y | y <- [4..], mod y 2 > 0])
The Sieve of Eratosthenes

```haskell
primes = sieve [2..]
    where sieve (x:l) =
        x : sieve [y | y <- l, mod y x > 0]
```

How does this work?

```haskell
sieve [2..]
⇒ 2:sieve [y | y <- [3..], mod y 2 > 0]
⇒ 2:sieve (3:[y | y <- [4..], mod y 2 > 0])
⇒ 2:3:sieve [z | z <- [y <- [4..],
                         mod y 2 > 0], mod z 3 > 0]
...
The Sieve of Eratosthenes

\[
\text{primes} = \text{sieve} \ [2..]
\]
\[
\text{where sieve} \ (x:l) =
\]
\[
x : \ \text{sieve} \ [y \mid y <- l, \ \text{mod} \ y \ x > 0]
\]

How does this work?

sieve \ [2..]
\[
\leadsto 2 : \text{sieve} \ [y \mid y <- [3..], \ \text{mod} \ y \ 2 > 0]
\]
\[
\leadsto 2 : \text{sieve} \ (3 : [y \mid y <- [4..], \ \text{mod} \ y \ 2 > 0])
\]
\[
\leadsto 2 : 3 : \text{sieve} \ [z \mid z <- [y <- [4..],
\]
\[
\quad \text{mod} \ y \ 2 > 0], \ \text{mod} \ z \ 3 > 0]
\]
\[
\ldots
\]
\[
\leadsto 2 : 3 : \text{sieve} \ [z \mid z <- [5,7,9...],
\]
\[
\quad \text{mod} \ z \ 3 > 0] \ldots
\]
The Sieve of Eratosthenes

primes = sieve [2..]
  where sieve (x:l) =
      x : sieve [y | y <- l, mod y x > 0]

How does this work?

sieve [2..]
  \rightarrow 2:sieve [y | y <- [3..], mod y 2 > 0]
  \rightarrow 2:sieve (3:[y | y <- [4..], mod y 2 > 0])
  \rightarrow 2:3:sieve [z | z <- [y <- [4..],
      mod y 2 > 0], mod z 3 > 0]
...
  \rightarrow 2:3:sieve [z | z <- [5,7,9...],
      mod z 3 > 0]...
  \rightarrow 2:3:sieve [5,7,11...] \rightarrow...
The $n^{th}$ prime

- We now have an infinite list $\text{primes}$ of primes.
The $n^{th}$ prime

- We now have an infinite list $\text{primes}$ of primes
- $\text{nthprime } n = \text{head} \ (\text{drop} \ (n-1) \ \text{primes})$
The $n^{th}$ prime

- We now have an infinite list $\text{primes}$ of primes
- $\text{n}th\text{prime } n = \text{head}\text{ (drop}\text{ (n-1) primes)}$
- Drop the first $n-1$ numbers from $\text{primes}$
The $n^{th}$ prime

- We now have an infinite list $\text{primes}$ of primes
- $\text{nthprime } n = \text{head (drop (n-1) primes)}$
- Drop the first $n-1$ numbers from $\text{primes}$
- To take the $\text{head}$ of the rest, only need to compute one more entry in the list
The $n^{th}$ prime

- We now have an infinite list $\text{primes}$ of primes
- $\text{nthprime } n = \text{head (drop (n-1) primes)}$
- Drop the first $n-1$ numbers from $\text{primes}$
- To take the $\text{head}$ of the rest, only need to compute one more entry in the list
- Once “enough” has been computed, the rest of $\text{primes}$ is ignored!
Functional programming provides a framework for declarative programming
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
  - Rapid prototyping
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
  - Rapid prototyping
- Haskell has a powerful typing mechanism
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
  - Rapid prototyping
- Haskell has a powerful typing mechanism
  - Conditional polymorphism
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
  - Rapid prototyping
- Haskell has a powerful typing mechanism
  - Conditional polymorphism
  - Modules with hiding and overriding for abstract datatypes
Concluding remarks

- Functional programming provides a framework for declarative programming
  - Provably correct programs
  - Rapid prototyping
- Haskell has a powerful typing mechanism
  - Conditional polymorphism
  - Modules with hiding and overriding for abstract datatypes
- Lazy evaluation permits infinite data structures
For more information

Software and other resources

- http://www.haskell.org

Quick tutorial

- *A Gentle Introduction to Haskell*
  by Paul Hudak et al

Textbooks

- *The Craft of Functional Programming*
  by Simon Thompson

- *Introduction to Functional Programming in Haskell*
  by Richard Bird