Functional Programming in Haskell

Part 2 : Abstract dataypes 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

Haskell review

- Program $\equiv$ Collection of function definitions

Haskell review

- Program $\equiv$ Collection of function definitions
- Computation $\equiv$ Rewriting using definitions

Haskell review

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

Haskell review

- Program $\equiv$ Collection of function definitions
- Computation $\equiv$ Rewriting using definitions
- Functions are associated with input and output types
- `isDigit ::  Char -> Bool`

Haskell review

- Program ≡ Collection of function definitions
- Computation ≡ Rewriting using definitions
- Functions are associated with input and output types
- `isDigit ::  Char -> Bool`

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

Haskell review

- Program $\equiv$ Collection of function definitions
- Computation $\equiv$ Rewriting using definitions
- Functions are associated with input and output types
- `isDigit ::  Char -> Bool`

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

- `isDigit c`

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

Haskell review . . .

- Basic collective type is a list

Haskell review . . .

- Basic collective type is a list
- Define list functions by induction on structure

Haskell review . . .

- Basic collective type is a list
- Define list functions by induction on structure
- Example Adding up a list of integers

Haskell review . . .

- 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)
```

Haskell review . . .

- 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

- 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

User defined datatypes

- The `data` declaration adds new datatypes

User defined datatypes

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

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

User defined datatypes

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

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

- Can use this type in a function such as

```
stopwhen ::  Signal -> Bool
stopwhen Red  =  True
stopwhen c    =  False
```

User defined datatypes

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

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

- Can use this type in a function such as

```
stopwhen ::  Signal -> Bool
stopwhen Red  =  True
stopwhen c    =  False
```

- What if we write instead

```
stopwhen2 ::  Signal -> Bool
stopwhen2 c |  (c == Red)  =  True
            |  otherwise   =  False
```

User defined types and type classes

- `stopwhen2` requires `Eq Signal`

User defined types and type classes

- `stopwhen2` 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

```
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

```
show ::  a -> String
```

  that allows values of `a` to be displayed

Adding user defined types to type classes

- Simplest solution is

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

Adding user defined types to type classes

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

Adding user defined types to type classes

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

Adding user defined types to type classes

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

Adding user defined types to type classes

- Simplest solution is
  `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`

Adding user defined types to type classes . . .

Or, provide your own functions

Adding user defined types to type classes . . .

Or, provide your own functions

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

Adding user defined types to type classes . . .

Or, provide your own functions

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

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

Adding user defined types to type classes . . .

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

```
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 `<=`

Adding user defined types to type classes . . .

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
```

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

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

Recursive datatypes

- A binary tree to store integers at each node

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

- `Nil` and `Node` are constructors

Recursive datatypes

- A binary tree to store integers at each node

```
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

```
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

```
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

```
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

Recursive datatypes

- A binary tree to store integers at each node

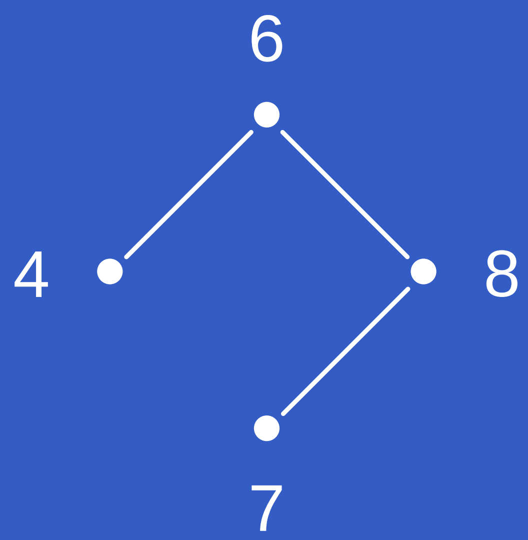
```
data Btreeint =
                Nil |
                Node Int Btreeint Btreeint
```

- `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

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?

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)
```

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)
```

- 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?

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

- List out all values in the tree

```
listout ::  Btreeint -> [Int]
listout Nil             = []
listout (Node n t1 t2)  =
  [n] ++ listout t1 ++ listout t2
```

Polymorphic recursive datatypes

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

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

Polymorphic recursive datatypes

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

```
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?

```
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?

```
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!

```
(Ord a) => data Btree a =
              Nil |
              Node a (Btree a) (Btree a)
```

Polymorphic recursive datatypes . . .

- Built in list type is a polymorphic recursive datatype

Polymorphic recursive datatypes . . .

- Built in list type is a polymorphic recursive datatype

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

Polymorphic recursive datatypes . . .

- Built in list type is a polymorphic recursive datatype

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

- Since lists are built in, they can use special symbols `[]` and `:` for constructors `Emptylist` and `Append`

Adding recursive datatypes to type classes

- Can inherit type classes from underlying type

Adding recursive datatypes to type classes

- Can inherit type classes from underlying type

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

Adding recursive datatypes to type classes

- Can inherit type classes from underlying type

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

Note: Not `Eq (Btree a)`

but `Eq a => Eq (Btree a)`

Adding recursive datatypes to type classes

- Can inherit type classes from underlying type

```
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

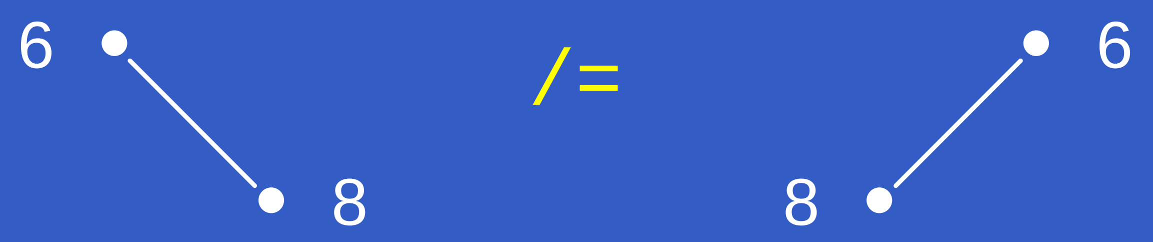
- Can inherit type classes from underlying type

```
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

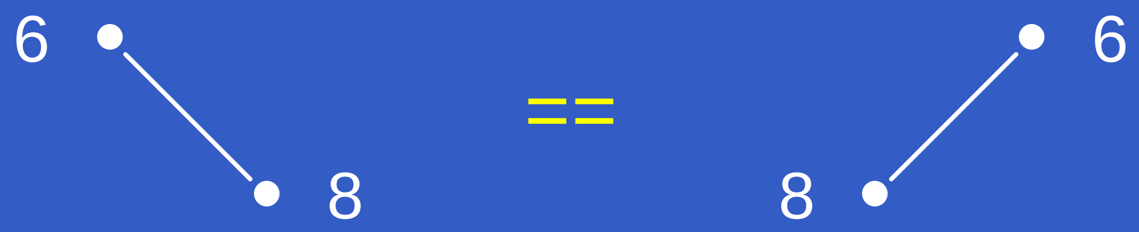
Adding recursive datatypes to type classes . . .

- Or we can define our own functions

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

- Or we can define our own functions

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



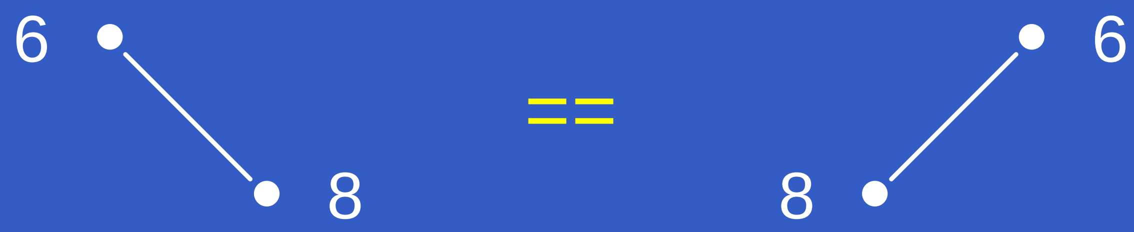
because

```
[6,8] == [6,8]
```

Adding recursive datatypes to type classes . . .

- Or we can define our own functions

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



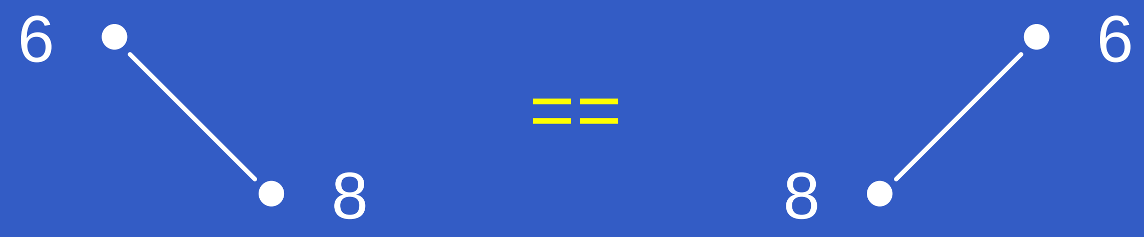
because

```
[6,8] == [6,8]
```

Adding recursive datatypes to type classes . . .

- 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]
```

- Rotate right — transformation used to balance trees

Declarative programming with abstract datatypes

- Rotate right — transformation used to balance trees

- `rotateright (Node x (Node y t1 t2) t3) =`
  `Node y t1 (Node x t2 t3)`

- Example Queues

Abstract datatypes

- Example Queues
- Stores sequence of values in FIFO fashion

Abstract datatypes

- 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

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

Abstract datatypes . . .

- Implement a queue as a list

Abstract datatypes . . .

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

Abstract datatypes . . .

- 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 []
  ```
  ```

Modules

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

Modules

- 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) ...
```

Modules

- 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`

Modules

- 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`
- How do we prevent unauthorized access to a queue?

```
remsec ::  (Queue a) -> (a,Queue a)
remsec (Qu (x:y:l)) = (y, Qu (x:l))
```

Modules . . .

- Restrict visibility outside module

Modules . . .

- Restrict visibility outside module

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

Modules . . .

- Restrict visibility outside module

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

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

Modules . . .

- Restrict visibility outside module

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

- Constructor `Qu` is not visible if you do `import Queue`
- Can override imported function with local definition

Modules . . .

- Restrict visibility outside module

  ```
  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`

Modules . . .

- Restrict visibility outside module

  ```
  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)`

Modules . . .

- Restrict visibility outside module

  ```
  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)`

Rewriting revisted

- More than one expression may qualify for rewriting

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

$\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `49`

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `49`
- If there are multiple expressions to rewrite, Haskell chooses outermost expression

Rewriting revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `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 revisted

- More than one expression may qualify for rewriting
- `sqr x = x*x`
- `sqr (4+3)`

  $\leadsto$ `sqr 7` $\leadsto$ `7*7` $\leadsto$ `49`

  $\leadsto$ `(4+3)*(4+3)` $\leadsto$ `(4+3)*7` $\leadsto$ `7*7` $\leadsto$ `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.
- “Eager” rewriting — evaluate arguments before evaluating function

Lazy rewriting

- Haskell evaluates arguments only when needed

Lazy rewriting

- 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))
```

Lazy rewriting

- 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` $\leadsto$ `1.0`

Infinite lists!

- The following definition makes sense in Haskell

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

Infinite lists!

- The following definition makes sense in Haskell

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

```
from 2
```

Infinite lists!

- The following definition makes sense in Haskell

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

```
from 2
```

$\leadsto$ `2:(from 3)`
$\leadsto$ `2:(3:(from 4))`
$\leadsto$ `2:(3:(4:(from 5)))`

...

Infinite lists!

- 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,...]`

Infinite lists!

- The following definition makes sense in Haskell

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

```
from 2
```

$\leadsto$ `2:(from 3)`
$\leadsto$ `2:(3:(from 4))`
$\leadsto$ `2:(3:(4:(from 5)))`

```
...
```

- Limit is the infinite list `[2,3,4,5,...]`
- Haskell can (and will) generate it incrementally, till you stop it, or it runs out of memory

Infinite lists!

- 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,...]`
- Haskell can (and will) generate it incrementally, till you stop it, or it runs out of memory
- Can write `[2..]` to denote `[2,3,4,...]`

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

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,...]`

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,...]`
  - 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,...]`
  - 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,...]`
  - Transfer smallest number into list of primes and delete all its multiples
  - Repeat second step forever!

The Sieve of Eratosthenes

```
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

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

How does this work?

```
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

```
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])
⇝2:3:sieve [z | z <- [y <- [4..],
                  mod y 2 > 0], mod z 3 > 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])
⤳2:3:sieve [z | z <- [y <- [4..],
                 mod y 2 > 0], mod z 3 > 0]
...
⤳2:3:sieve [z | z <- [5,7,9...],
                           mod z 3 > 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])
⇝2:3:sieve [z | z <- [y <- [4..],
                 mod y 2 > 0], mod z 3 > 0]
...
⇝2:3:sieve [z | z <- [5,7,9...],
                           mod z 3 > 0]...
⇝2:3:sieve [5,7,11...] ⇝...
```

The n^{th} prime

- We now have an infinite list `primes` of primes

The n^{th} prime

- We now have an infinite list `primes` of primes
- `nthprime n = head (drop (n-1) primes)`

The n^{th} prime

- We now have an infinite list `primes` of primes
- `nthprime n = head (drop (n-1) primes)`
- Drop the first $n-1$ numbers from `primes`

The n^{th} prime

- We now have an infinite list `primes` of primes
- `nthprime n = head (drop (n-1) primes)`
- Drop the first $n-1$ numbers from `primes`
- To take the `head` of the rest, only need to compute one more entry in the list

The n^{th} prime

- We now have an infinite list `primes` of primes
- `nthprime n = head (drop (n-1) primes)`
- Drop the first $n-1$ numbers from `primes`
- To take the `head` of the rest, only need to compute one more entry in the list
- Once “enough” has been computed, the rest of `primes` is ignored!

Concluding remarks

- 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