Functional Programming in Haskell

Part I : Basics

Madhavan Mukund
Chennai Mathematical Institute
92 G N Chetty Rd, Chennai 600 017, India
madhavan@cmi.ac.in
http://www.cmi.ac.in/~madhavan
Functions

- Transform inputs to output
- Operate on specific types

These functions are different
Functional programming

- Program $\Leftrightarrow$ set of function definitions
- Function definition $\Leftrightarrow$ how to “calculate” the value
- Declarative programming
  - Provably correct programs
    Functional program closely follows mathematical definition
  - Rapid prototyping
    Easy to go from specification (what we require) to implementation (working program)
Functional programming in Haskell

- **Built-in types** `Int`, `Float`, `Bool`, ... with basic operations `+`, `−`, `∗`, `/`, `||`, `&&`, ...

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

Multiple arguments are consumed "one at a time"

- Not `. . .`

- `power :: Float, Int -> Float` `power (x,n) = ...`

- Need not be a "total" function

- What is `power 2.0 -1`?
Built-in types `Int`, `Float`, `Bool`, ...
with basic operations `+`, `−`, `∗`, `/`, `||`, `&&`, ...

Defining a new function (and its type)

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

Multiple arguments are consumed "one at a time"

Not . . .

`power :: Float Int -> Float` `power (x,n) = ...`

Need not be a "total" function

What is `power 2.0 -1`?
Functional programming in Haskell

- Built-in types `Int`, `Float`, `Bool`, ...
  with basic operations `+`, `−`, `∗`, `/`, `||`, `&&`, ...

- **Defining a new function (and its type)**

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

- Multiple arguments are consumed “one at a time”

  Not ...

  ```haskell
  power :: Float × Int → Float
  power (x,n) = ...
  ```
- **Built-in types** `Int, Float, Bool, ...` with basic operations `+, -, *, /, |\|, &&, ...`

- **Defining a new function (and its type)**
  ```haskell
  power :: Float -> Int -> Float
  power x n = if (n == 0) then 1.0
               else x * (power x (n-1))
  ```

- **Multiple arguments are consumed “one at a time”**
  Not ...  
  ```haskell
  power :: Float \times Int -> Float
  power (x,n) = ...
  ```

- **Need not be a “total” function**
  What is `power 2.0 -1`?
Ways of defining functions

- Multiple definitions, read top to bottom
Ways of defining functions

- Multiple definitions, read top to bottom
- Definition by cases

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n = x * (power x (n-1))
```
Ways of defining functions

- Multiple definitions, read top to bottom
- Definition by cases

\[
\begin{align*}
\text{power} &:: \text{Float} \to \text{Int} \to \text{Float} \\
power x 0 & = 1.0 \\
power x n & = x \times (\text{power} x (n-1))
\end{align*}
\]

- Implicit “pattern matching” of arguments

\[
\begin{align*}
\text{xor} &:: \text{Bool} \to \text{Bool} \to \text{Bool} \\
xor \text{ True True} & = \text{False} \\
xor \text{ False False} & = \text{False} \\
xor x y & = \text{True}
\end{align*}
\]
Multiple options with conditional guards

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

\[
\text{max } i \ j \mid (i \geq j) = i \\
\mid (i < j) = j
\]
Ways of defining functions . . .

- Multiple options with conditional guards

  \[
  \text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
  \text{max } i \ j \ | \ (i \geq j) = i \\
  \quad \mid (i < j) = j
  \]

- Default conditional value — \text{otherwise}

  \[
  \text{max3} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
  \text{max3 } i \ j \ k \ | \ (i \geq j) \land (i \geq k) = i \\
  \quad \mid (j \geq k) = j \\
  \quad \mid \text{otherwise} = k
  \]

Note: Conditional guards are evaluated top to bottom!
Ways of defining functions . . .

- **Multiple options with conditional guards**

  \[
  \text{max} :: \text{Int} \to \text{Int} \to \text{Int}
  \]
  \[
  \text{max } i \; j \; | \; (i \geq j) = i \\
  \; | \; (i < j) = j
  \]

- **Default conditional value — otherwise**

  \[
  \text{max3} :: \text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
  \]
  \[
  \text{max3 } i \; j \; k \; | \; (i \geq j) \land (i \geq k) = i \\
  \; | \; (j \geq k) = j \\
  \; | \; \text{otherwise} = k
  \]

- **Note:** Conditional guards are evaluated top to bottom!
Pairs, triples, …

- Can form $n$-tuples of types

  $(x,y,z) :: (\text{Float},\text{Float},\text{Float})$

  represents a point in 3D

  Can define a function

  $\text{distance3D} :: (\text{Float},\text{Float},\text{Float}) \to (\text{Float},\text{Float},\text{Float}) \to \text{Float}$

  Functions can return $n$-tuples

  $\text{maxAndMinOf3} :: \text{Int} \to \text{Int} \to \text{Int} \to (\text{Int},\text{Int})$
Pairs, triples, ...

- Can form $n$-tuples of types
- $(x, y, z) :: (\text{Float, Float, Float})$
  represents a point in 3D
Can form \( n \)-tuples of types

\[(x, y, z) :: (\text{Float, Float, Float})\]

represents a point in 3D

Can define a function

\[
distance3D :: (\text{Float, Float, Float}) -> (\text{Float, Float, Float}) -> \text{Float}
\]
Pairs, triples, …

- Can form \( n \)-tuples of types

- \((x, y, z) :: (\text{Float}, \text{Float}, \text{Float})\)
  represents a point in 3D

- Can define a function

  \[
  \text{distance3D} :: \\
  (\text{Float}, \text{Float}, \text{Float}) \to \\
  (\text{Float}, \text{Float}, \text{Float}) \to \text{Float}
  \]

- Functions can return \( n \)-tuples

  \[
  \text{maxAndMinOf3} :: \\
  \text{Int} \to \text{Int} \to \text{Int} \to (\text{Int}, \text{Int})
  \]
Local definitions using `where`

**Example:** Compute distance between two points in 2D

```haskell
distance ::
    (Float,Float) -> (Float,Float) -> Float

distance (x1,y1) (x2,y2) =
    sqrt((sqr xdistance) + (sqr ydistance))
where

    xdistance = x2 - x1
    ydistance = y2 - y1

sqr :: Float -> Float
sqr z = z*z
```

Madras Christian College, 8 December 2003 – p.8
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  
  \[ 3 + 5 \Rightarrow 8 \quad \text{True} \quad | \quad \text{False} \Rightarrow \text{True} \]
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  \[ 3 + 5 \sim 8 \quad \text{True} \mid \mid \text{False} \sim \text{True} \]
- Simplifications based on user defined functions
  \[ \text{power } 3.0 \ 2 \]
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications
  
  $3 + 5 \leadsto 8 \quad \text{True} \quad \text{||} \quad \text{False} \leadsto \text{True}$
- Simplifications based on user defined functions
  
  $\text{power } 3.0 \ 2$  
  
  $\leadsto 3.0 \ * \ (\text{power } 3.0 \ (2-1))$
Use definitions to simplify expressions till no further simplification is possible

Built-in simplifications

\[ 3 + 5 \sim 8 \quad \text{True} \mid \mid \text{False} \sim \text{True} \]

Simplifications based on user-defined functions

\[
\text{power 3.0 2} \\
\sim 3.0 * (\text{power 3.0 (2-1)}) \\
\sim 3.0 * (\text{power 3.0 1})
\]
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible
- Builtin simplifications

\[ 3 + 5 \mapsto 8 \quad \text{True} \mid \mid \text{False} \mapsto \text{True} \]

- Simplifications based on user defined functions

\[ \text{power} \ 3.0 \ 2 \]
\[ \mapsto 3.0 \ * \ (\text{power} \ 3.0 \ (2-1)) \]
\[ \mapsto 3.0 \ * \ (\text{power} \ 3.0 \ 1) \]
\[ \mapsto 3.0 \ * \ 3.0 \ * \ (\text{power} \ 3.0 \ (1-1)) \]
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible

- Builtin simplifications
  
  \[3 + 5 \rightarrow 8\]
  \[True \lor False \rightarrow True\]

- Simplifications based on user defined functions

\[
\text{power } 3.0 \ 2
\]
\[
\rightarrow 3.0 \ast (\text{power } 3.0 \ (2-1))
\]
\[
\rightarrow 3.0 \ast (\text{power } 3.0 \ 1)
\]
\[
\rightarrow 3.0 \ast 3.0 \ast (\text{power } 3.0 \ (1-1))
\]
\[
\rightarrow 3.0 \ast 3.0 \ast (\text{power } 3.0 \ 0)
\]
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible

- Built-in simplifications
  - $3 + 5 \rightarrow 8$
  - True || False $\rightarrow$ True

- Simplifications based on user defined functions
  - \texttt{power 3.0 2}
  - $\rightarrow 3.0 \times (\text{power} \ 3.0 \ (2-1))$
  - $\rightarrow 3.0 \times (\text{power} \ 3.0 \ 1)$
  - $\rightarrow 3.0 \times 3.0 \times (\text{power} \ 3.0 \ (1-1))$
  - $\rightarrow 3.0 \times 3.0 \times (\text{power} \ 3.0 \ 0)$
  - $\rightarrow 3.0 \times 3.0 \times 1.0$
Computation is rewriting

- Use definitions to simplify expressions till no further simplification is possible

- Builtin simplifications
  
  \[ 3 + 5 \sim 8 \quad \text{True} \quad \text{||} \quad \text{False} \sim \text{True} \]

- Simplifications based on user defined functions

  \[ \text{power } 3.0 \ 2 \]
  \[ \sim 3.0 \ * \ (\text{power } 3.0 \ (2-1)) \]
  \[ \sim 3.0 \ * \ (\text{power } 3.0 \ 1) \]
  \[ \sim 3.0 \ * \ 3.0 \ * \ (\text{power } 3.0 \ (1-1)) \]
  \[ \sim 3.0 \ * \ 3.0 \ * \ (\text{power } 3.0 \ 0) \]
  \[ \sim 3.0 \ * \ 3.0 \ * \ 1.0 \]
  \[ \sim 9.0 \ * \ 1.0 \]
Computation is rewriting

- **Use definitions to simplify expressions till no further simplification is possible**
- **Builtin simplifications**
  
  \[
  3 + 5 \rightarrow 8 \quad \text{True} \quad \text{||} \quad \text{False} \rightarrow \text{True}
  \]
- **Simplifications based on user defined functions**

```plaintext
power 3.0 2
\rightarrow 3.0 * (power 3.0 (2-1))
\rightarrow 3.0 * (power 3.0 1)
\rightarrow 3.0 * 3.0 * (power 3.0 (1-1))
\rightarrow 3.0 * 3.0 * (power 3.0 0)
\rightarrow 3.0 * 3.0 * 1.0
\rightarrow 9.0 * 1.0 \rightarrow 9.0
```
Functions that manipulate functions

- A function with input type \( a \) and output type \( b \) has type \( \texttt{a -> b} \).

\[
\text{apply} \quad \text{::} \quad (\texttt{Int -> Int}) \rightarrow \texttt{Int} \rightarrow \texttt{Int} \\
\text{apply } f \ n = f \ n
\]

\[
\text{twice} \quad \text{::} \quad (\texttt{Int -> Int}) \rightarrow \texttt{Int} \rightarrow \texttt{Int} \\
twice f \ n = f \ (f \ n)
\]

\[
\text{twice } \texttt{sqr} \ 7 \\
twice \texttt{sqr} \ (\texttt{sqr} \ 7) \\
\texttt{sqr} \ (7 \times 7) \\
49 \times 49 \\
2401
\]
A function with input type \( a \) and output type \( b \) has type \( a \rightarrow b \).

Recall useful convention that all functions read one argument at a time!
A function with input type \( a \) and output type \( b \) has type \( a \rightarrow b \)

Recall useful convention that all functions read one argument at a time!

A function can take another function as argument
A function with input type \(a\) and output type \(b\) has type \(a \to b\).

Recall useful convention that all functions read one argument at a time!

A function can take another function as argument

\[
\text{apply} :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int}
\]

\[
\text{apply } f \ n = f \ n
\]
A function with input type \( a \) and output type \( b \) has type \( a \to b \)

Recall useful convention that all functions read one argument at a time!

A function can take another function as argument

\[
\text{apply} :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \\
\text{apply} f n = f n
\]

\[
\text{twice} :: (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \\
\text{twice} f n = f (f n)
\]
Functions that manipulate functions

- A function with input type \( a \) and output type \( b \) has type \( a \to b \).
- Recall useful convention that all functions read one argument at a time!
- A function can take another function as argument

**apply**: \( (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \)

\[
\text{apply} \ f \ n = f \ n
\]

**twice**: \( (\text{Int} \to \text{Int}) \to \text{Int} \to \text{Int} \)

\[
\text{twice} \ f \ n = f \ (f \ n)
\]

\[
\text{twice} \ \text{sqr} \ 7 \to \text{sqr} \ (\text{sqr} \ 7) \to \text{sqr} \ (7 \times 7) \to \cdots \to 49 \times 49 \to 2401
\]
Running Haskell programs

- **hugs** — A Haskell interpreter
  Available for Linux, Windows, ...
- **ghc** — the Glasgow Haskell Compiler
- Look at [http://www.haskell.org](http://www.haskell.org)
Basic collective type is a list

- All items of a list must be of the same type:
  - Int — list of Int
  - (Float, Float) — list of pairs of Float

Lists are written as follows:
- [2, 3, 1, 7]
- [(3.0, 7.5), (7.6, 9.2), (3.3, 7.868)]

- Empty list is denoted [] (for all types)
- Can have lists of lists (to any depth)
- [[Int]] — list of [Int]
- [[[2, 3, 1, 7], [], [8, 3], [9]]]
Basic collective type is a list

All items of a list must be of the same type

\[[\text{Int}]\] — list of \text{Int},

\[[\text{(Float,Float)}]\] — list of pairs of \text{Float}
• Basic collective type is a list
• All items of a list must be of the same type
  \([\text{Int}]\) — list of \text{Int},
  \([(\text{Float,Float})]\) — list of pairs of \text{Float}
• Lists are written as follows:
  \([2,3,1,7]\)
  \([(3.0,7.5),(7.6,9.2),(3.3,7.868)]\)
Collections

- Basic collective type is a list
- All items of a list must be of the same type
  
  \[ \text{[Int]} \] — list of \text{Int}, 
  
  \[ \text{[(Float,Float)]} \] — list of pairs of \text{Float}

- Lists are written as follows:
  
  \[ [2,3,1,7] \]
  
  \[ [(3.0,7.5),(7.6,9.2),(3.3,7.868)] \]

- Empty list is denoted \[ [] \] (for all types)
Basic operations on lists

- ++ concatenates lists

\[[1, 3] ++ [5, 7] = [1, 3, 5, 7]\]
Basic operations on lists

- ++ concatenates lists
  \[ [1,3] ++ [5,7] = [1,3,5,7] \]

- Unique way of decomposing a nonempty list
  - head: first element of the list
  - tail: the rest (may be empty!)
  \[ \text{head} \ [1,3,5,7] = 1 \]
  \[ \text{tail} \ [1,3,5,7] = [3,5,7] \]
Basic operations on lists

- `++` concatenates lists
  
  
  \[
  [1, 3] \; ++ \; [5, 7] = [1, 3, 5, 7]
  \]

- Unique way of decomposing a nonempty list
  
  - `head` : first element of the list
  - `tail` : the rest (may be empty!)

  \[
  \begin{align*}
  \text{head} \; [1, 3, 5, 7] &= 1 \\
  \text{tail} \; [1, 3, 5, 7] &= [3, 5, 7]
  \end{align*}
  \]

- **Note the types**: head is an element, tail is a list
Basic operations on lists

- ++ concatenates lists
  \[ [1, 3] ++ [5, 7] = [1, 3, 5, 7] \]

- Unique way of decomposing a nonempty list
  - **head**: first element of the list
  - **tail**: the rest (may be empty!)
    \[\text{head } [1, 3, 5, 7] = 1\]
    \[\text{tail } [1, 3, 5, 7] = [3, 5, 7]\]

- **Note the types**: head is an element, tail is a list

- Write \textbf{x: l} to denote the list with head \textbf{x}, tail \textbf{l}
Functions on lists

- Define functions by induction on list structure

```haskell
length :: [Int] -> Int
length [] = 0
length (x:l) = 1 + length l
```

```haskell
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:l) = reverse l ++ [x]
```
Define functions by induction on list structure

- Base case
  Value of \( f \) on \([ \]\)

- Step
  Extend value of \( f \) on \( l \) to \( f \) on \( x: l \)

\[
\text{length :: [Int] -> Int}
\]
\[
\text{length \([ \]\) = 0}
\]
\[
\text{length \((x:l)\) = 1 + length \(l\)}
\]

\[
\text{reverse :: [Int] -> [Int]}
\]
\[
\text{reverse \([ \]\) = [ ]}
\]
\[
\text{reverse \((x:l)\) = \((\text{reverse \(l\)}) \) ++ \([x\]}
\]
Define functions by *induction on list structure*

- **Base case**  
  Value of $f$ on $[]$

- **Step**  
  Extend value of $f$ on $l$ to $f$ on $x:l$

- $\text{length} :: [\text{Int}] \rightarrow \text{Int}$
- $\text{length} \ [\ ] = 0$
- $\text{length} \ (x:l) = 1 + \text{length} \ l$
Functions on lists

Define functions by **induction on list structure**

- **Base case**
  Value of \( f \) on \([\ ]\)

- **Step**
  Extend value of \( f \) on \( l \) to \( f \) on \( x:l \)

- \( \text{length} :: [\text{Int}] \rightarrow \text{Int} \)
  \[
  \text{length} \; [\ ] \; = \; 0 \\
  \text{length} \; (x:l) \; = \; 1 + \text{length} \; l
  \]

- \( \text{reverse} :: [\text{Int}] \rightarrow [\text{Int}] \)
  \[
  \text{reverse} \; [\ ] \; = \; [] \\
  \text{reverse} \; (x:l) \; = \; (\text{reverse} \; l) \; ++ \; [x]
  \]
Some builtin list functions

- `length l`, `reverse l`, `sum l`, ...

Dually, `init l`, `last l`

- `init [1,2,3] = [1,2]`
- `last [1,2,3] = 3`

- `take n l` — extract the first `n` elements of `l`
- `drop n l` — drop the first `n` elements of `l`

Shortcut list notation

- `[m..n]` abbreviates the list `[m,m+1,...,n]`

Example
- `[3..7] = [3,4,5,6,7]`

Arithmetic progressions
- `[1,3..8] = [1,3,5,7]`
- `[9,8..5] = [9,8,7,6,5]`
Some builtin list functions

- `length l`, `reverse l`, `sum l`, ...
- `head l`, `tail l`
Some builtin list functions

- `length l, reverse l, sum l, ...`
- `head l, tail l`
- **Dually,** `init l, last l`
  - `init [1,2,3] = [1,2], last [1,2,3] = 3`
Some builtin list functions

- `length l`, `reverse l`, `sum l`, ...
- `head l`, `tail l`
- Dually, `init l`, `last l`
  - `init [1,2,3] = [1,2]`, `last [1,2,3] = 3`
- `take n l` — extract the first `n` elements of `l`
- `drop n l` — drop the first `n` elements of `l`
Some builtin list functions

- `length l, reverse l, sum l, ...
- `head l, tail l
- Dually, `init l, last l

  \[
  \text{init } \[1,2,3\] = \[1,2\], \text{last } \[1,2,3\] = 3
  \]

- `take n l — extract the first \(n\) elements of \(l\)
- `drop n l — drop the first \(n\) elements of \(l\)

- Shortcut list notation
Some builtin list functions

- `length l`, `reverse l`, `sum l`, ...
- `head l`, `tail l`

Dually, `init l`, `last l`

init \([1,2,3]\) = \([1,2]\), last \([1,2,3]\) = 3

- `take n l` — extract the first `n` elements of `l`
- `drop n l` — drop the first `n` elements of `l`

Shortcut list notation

- \([m..n]\) abbreviates the list \([m, m+1, \ldots, n]\)
  
  Example \([3..7]\) = \([3, 4, 5, 6, 7]\)
Some builtin list functions

- length l, reverse l, sum l, ...
- head l, tail l
- Dually, init l, last l
  init [1,2,3] = [1,2], last [1,2,3] = 3
- take n l — extract the first n elements of l
- drop n l — drop the first n elements of l

Shortcut list notation

- [m..n] abbreviates the list [m, m+1, ..., n]
  Example [3..7] = [3, 4, 5, 6, 7]
- Arithmetic progressions
  [1,3..8] = [1, 3, 5, 7]
  [9,8..5] = [9, 8, 7, 6, 5]
Operating on each element of a list

- `map` \( f \) \( l \) applies \( f \) to each item of \( l \)
map \ f \ l \text{ applies } f \text{ to each item of } l

\begin{align*}
\text{square} & \:: \text{ Int } \rightarrow \text{ Int} \\
\text{square } n & = n^2 \\
\text{map square } \{1, 2, 4, 9\} & \leadsto \{2, 4, 9, 81\}
\end{align*}
Operating on each element of a list

- **map** $f$ $l$ applies $f$ to each item of $l$

  ```haskell
  square :: Int -> Int
  square n = n*n
  map square [1,2,4,9] \rightarrow [2,4,9,81]
  ```

- **filter** $p$ $l$ selects items from $l$ that satisfy $p$

  ```haskell
  even :: Int -> Bool
  even x = (mod x 2 == 0)
  filter even [1,2,4,9] \rightarrow [2,4]
  ```

Can compose these functions:

```haskell
map square (filter even [1..10]) \rightarrow [4,16,36,64,100]
```
Operating on each element of a list

- **map f l** applies \( f \) to each item of \( l \)
  - \( \text{square} :: \text{Int} \rightarrow \text{Int} \)
  - \( \text{square} \ n = n^2 \)
  - \( \text{map} \ \text{square} \ [1,2,4,9] \rightarrow [2,4,9,81] \)

- **filter p l** selects items from \( l \) that satisfy \( p \)
  - \( \text{even} :: \text{Int} \rightarrow \text{Bool} \)
  - \( \text{even} \ x = (\text{mod} \ x \ 2 \ = \ 0) \)
  - \( \text{filter} \ \text{even} \ [1,2,4,9] \rightarrow [2,4] \)
Operating on each element of a list

- **map f l** applies f to each item of l

  ```
  square :: Int -> Int
  square n = n*n
  map square [1,2,4,9] ↦ [2,4,9,81]
  ```

- **filter p l** selects items from l that satisfy p

  ```
  even :: Int -> Bool
  even x = (mod x 2 == 0)
  filter even [1,2,4,9] ↦ [2,4]
  ```

- Can compose these functions
Operating on each element of a list

- **map f l** applies f to each item of l
  
  ```haskell
  square :: Int -> Int
  square n = n*n
  map square [1,2,4,9] \mapsto [2,4,9,81]
  ```

- **filter p l** selects items from l that satisfy p
  
  ```haskell
  even :: Int -> Bool
  even x = (mod x 2 == 0)
  filter even [1,2,4,9] \mapsto [2,4]
  ```

- Can compose these functions
  
  ```haskell
  map square (filter even [1..10]) \mapsto [4,16,36,64,100]
  ```
The set of squares of the even numbers between 1 and 10

\[ \{ x^2 \mid x \in \{1, \ldots, 10\}, \text{even}(x) \} \]
List comprehension: New lists from old

- The set of squares of the even numbers between 1 and 10

\[ \{ x^2 \mid x \in \{1, \ldots, 10\}, \text{even}(x) \} \]

- The list of squares of the even numbers between 1 and 10

\[
[ \text{square } x \mid x \leftarrow [1..10], \text{even } x ]
\]

where even \( x = (\text{mod } x 2 == 0) \)

\( \text{square } x = x^2 \)
Using list comprehensions . . .

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

\[
\text{divisors } n = [ m | m <- [1..n], \text{mod } n \text{ m } = 0 ]
\]
Using list comprehensions...

- **divisors** :: Int -> [Int]
  
divisors n = [ m | m <- [1..n],
                mod n m == 0 ]

- **prime** :: Int -> Bool
  
prime n = (divisors n == [1,n])
Example: Quicksort

- Choose an element of the list as a splitter and create sublists of elements smaller than the splitter and larger than the splitter
- Recursively sort these sublists and combine
Example: Quicksort

- Choose an element of the list as a splitter and create sublists of elements smaller than the splitter and larger than the splitter
- Recursively sort these sublists and combine

```haskell
qsort [] = []
qsort l =
    qsort lower ++ [splitter] ++ qsort upper
where
    splitter = head l
    lower = [i | i <- tail l, i <= splitter]
    upper = [i | i <- tail l, i > splitter]
```
Are
definitions

\[
\text{length} :: [\text{Int}] \rightarrow \text{Int} \\
\text{length} :: [\text{Float}] \rightarrow \text{Int}
\]

different functions?
Polymorphism

Are

\[
\text{length} :: [\text{Int}] \rightarrow \text{Int} \\
\text{length} :: [\text{Float}] \rightarrow \text{Int}
\]
different functions?

\text{length} only looks at the “structure” of the list, not “into” individual elements

For any underlying type \(t\), \text{length} :: [t] \rightarrow \text{Int}
Are

\[
\text{length} :: [\text{Int}] \to \text{Int} \\
\text{length} :: [\text{Float}] \to \text{Int}
\]
different functions?

\text{length} only looks at the “structure” of the list, not “into” individual elements

For any underlying type \( t \), \( \text{length} :: [t] \to \text{Int} \)

Use \( a, b, \ldots \) to denote generic types

So, \( \text{length} :: [a] \to \text{Int} \)
Are

\[
\text{length :: } [\text{Int}] \rightarrow \text{Int} \\
\text{length :: } [\text{Float}] \rightarrow \text{Int}
\]
different functions?

\(\text{length}\) only looks at the “structure” of the list, not “into” individual elements

For any underlying type \(t\), \(\text{length :: } [t] \rightarrow \text{Int}\)

Use \(a, b, \ldots\) to denote generic types

So, \(\text{length :: } [a] \rightarrow \text{Int}\)

Similarly, the most general type of \(\text{reverse}\) is

\(\text{reverse :: } [a] \rightarrow [a]\)
“True” polymorphism

The *same* computation is performed for different types
“True” polymorphism

The **same** computation is performed for different types

Overloading

Same symbol or function name denotes **different** computations for different types
“True” polymorphism

The same computation is performed for different types

Overloading

Same symbol or function name denotes different computations for different types

Example Arithmetic operators

At bit level, algorithms for \texttt{Int + Int} and \texttt{Float + Float} are different
Polymorphism versus overloading

- **“True” polymorphism**
  
  The *same* computation is performed for different types

- **Overloading**
  
  Same symbol or function name denotes *different* computations for different types

- **Example** Arithmetic operators
  
  At bit level, algorithms for `Int + Int` and `Float + Float` are different

- What about subclass polymorphism in OO programming?
class Shape {
}

class Circle extends Shape {
    double size {return pi*radius*radius}
}

class Square extends Shape {
    double size {return side*side}
}

Shape s1 = new Circle; print s1.size();
Shape s2 = new Square; print s2.size();
class Shape {
}
class Circle extends Shape {
    double size { return pi*radius*radius }
}
class Square extends Shape {
    double size { return side*side }
}
Shape s1 = new Circle; print s1.size();
Shape s2 = new Square; print s2.size();

Implementation of size is different!!
Conditional polymorphism

What about

```haskell
member x [] = False
member x (y:l) | (x == y) = True
| otherwise = member x l
```

Is `member :: a -> [a] -> Bool` a valid description of the type?

What is the value of `member qsort [qsort, mergesort, plus]`?

Equality of functions cannot be checked effectively.

The underlying type should support equality.
What about

\[
\text{member } x \ [\] = \text{False} \\
\text{member } x \ (y:z) \mid (x == y) = \text{True} \\
\mid \text{otherwise} = \text{member } x \ z
\]

Is \text{member} :: a \rightarrow [a] \rightarrow \text{Bool} a valid
description of the type?
Conditional polymorphism

- What about
  
  ```haskell
  member x [] = False
  member x (y:l) | (x == y) = True
  | otherwise = member x l
  ```

- Is `member :: a -> [a] -> Bool` a valid description of the type?

- What is the value of
  
  ```haskell
  member qsort [qsort, mergesort, plus]
  ```

  Equality of functions cannot be checked effectively
What about

\[
\text{member } x \ [\] = \text{False} \\
\text{member } x \ (y: l) \ \mid \ (x == y) = \text{True} \\
\ \mid \ \text{otherwise} = \text{member } x \ l
\]

Is \text{member} :: a \rightarrow [a] \rightarrow \text{Bool} \ a \ valid 
description of the type?

What is the value of

\[
\text{member qsort \ [qsort, mergesort, plus]}?
\]

Equality of functions cannot be checked effectively

The underlying type should support equality
Haskell organizes types into **classes**. A **type class** is a subset of all types.

The class `Eq` contains all types that support `==` on their elements.

The “predicate” `Eq a` tells whether or not `a` belongs to `Eq`.

Haskell would type this as

```
member :: Eq a => a -> [a] -> Bool
```

Likewise `Ord a` is the set of types that support comparison, so

```
quickSort :: Ord a => [a] -> [a]
```
Examples of declarative programming

Compute all initial segments of a list

Initial segments of \[
\]\ are empty

Initial segments of \[x:l\] — all initial segments of \[l\] with \[x\] in front, plus the empty segment

\[
\text{initial :: } [a] \rightarrow [[a]]
\]

\[
\text{initial } [] = [[]]
\]

\[
\text{initial } (x:l) = [[]] ++ [x:z \mid z \leftarrow \text{initial } l]
\]
Examples of declarative programming

Compute all initial segments of a list

- Initial segments of \([\ ]\) are empty
Examples of declarative programming

Compute all initial segments of a list

- Initial segments of `[]` are empty
- Initial segments of `x: l` — all initial segments of `l` with `x` in front, plus the empty segment
Examples of declarative programming

Compute all initial segments of a list

- Initial segments of \([\ ]\) are empty
- Initial segments of \(\mathtt{x: l}\) — all initial segments of \(\mathtt{l}\) with \(\mathtt{x}\) in front, plus the empty segment

\[
\text{initial} :: \ [\mathtt{a}] \rightarrow \ [[\mathtt{a}]] \\
\text{initial} \ [\ ] = [[\ ]]
\text{initial} \ (\mathtt{x: l}) = [[\ ]] \ ++ \\
\quad [ \mathtt{x: z} \mid \mathtt{z} \leftarrow \text{initial} \ l]
\]
Examples of declarative programming . . .

- The empty list $[]$ has no permutations

```haskell
interleave :: a -> [a] -> [[a]]
interleave x [] = [[]]
interleave x (y:ls) = [x:y:ls] ++ [y:ls2 | ls2 <- (interleave x ls)]

perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:xs) = [z | y <- perms xs, z <- interleave x y]
```
Examples of declarative programming . . .

- The empty list `[]` has no permutations
- Permutations of `x: l`  
  “Interleave” `x` through each permutation of `l`
Examples of declarative programming . . .

- The empty list `[]` has no permutations
- Permutations of `x:l`
  "Interleave" `x` through each permutation of `l`

```haskell
interleave :: a -> [a] -> [[a]]
interleave x [] = [[]]
interleave x (y:l) =
  [x:y:l] ++
  [y:l2 | l2 <- (interleave x l)]
```

```haskell
perms :: [a] -> [[a]]
perms [] = [[]]
perms (x:l) =
  [z | y <- perms l, z <- interleave x y]
```
- User defined datatypes
  Stacks, queues, trees, ...
- Hiding implementation details using modules
- "Infinite" data structures