

# Higher-order Programming and Types\*

**Andreas Lochbihler**

Department of Computer Science  
ETH Zurich

---

\*Thanks to David Basin for slide material

# Overview

- Review of higher-order functions
  - ▶ Functions as arguments
  - ▶ Functions as results
- Case study: matrix operations
- Haskell's type system

# First-order versus higher-order functions

- First-order functions

```
fo1 :: Int -> Int
fo1 x = x + 3
```

```
fo2 :: Int -> Int -> Int
fo2 x y = x + y + x * y
```

- Higher-order functions

```
ho1 :: (Int -> Int) -> Int
ho1 f = f 2
```

```
ho2 :: (Int -> a) -> a
ho2 f = f 2
```

```
? ho2 (\x->x+3)
5 :: Int
```

- Which order is the function: `mystery x = x ?`

## Examples: map, filter, and fold

**map** ::  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

```
map f []      = []  
map f (x:xs) = f x : map f xs
```

**filter** ::  $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

```
filter p [] = []  
filter p (x:xs)  
  | p x      = x : filter p xs  
  | otherwise =      filter p xs
```

**foldr** ::  $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

```
foldr f e []      = e  
foldr f e (x:xs) = f x (foldr f e xs)
```

## These abstract general operations

**Map:** iteratively apply a function to each element

```
? map (2*) [1 .. 5]  
[2, 4, 6, 8, 10] :: [Int]
```

```
? map (>2) [1 .. 5]  
[False, False, True, True, True] :: [Bool]
```

**Filter:** selection

```
? filter (>2) [1 .. 5]  
[3, 4, 5] :: [Int]
```

```
? filter (2>) [1 .. 5]  
[1] :: [Int]
```

**Fold:** use function to “combine” elements

```
? foldr (+) 0 [1 .. 5]  
15 :: Int
```

## Examples with filter

- Remove elements with property  $p$  (i.e., select those with  $\neg p$ )

```
remove p = filter (not . p)
```

```
? remove (>2) [1 .. 5]  
[1, 2] :: [Int]
```

- Partition lists using  $p$

```
part p xs = (filter p xs, remove p xs)
```

```
? part (>2) [1 .. 5]  
([3, 4, 5], [1, 2]) :: ([Int], [Int])
```

- Which partitioning function is better? In what sense?

```
partition p [] = ([], [])  
partition p (x:xs)  
  | p x      = (x:yesses, nos)  
  | otherwise = (yesses, x:nos)  
  where (yesses, nos) = partition p xs
```

## Quick sort (again)

- Quick sort with partition

```
quicksort [] = []  
quicksort (x:xs) = quicksort left ++ [x] ++ quicksort right  
    where (left,right) = partition (<= x) xs
```

- Which program is better?

```
q [] = []  
q (x:xs) = q [y | y<-xs, y <= x] ++ [x] ++ q [y | y<-xs, y > x]
```

```
r [] = []  
r (x:xs) = r left ++ (x : r right)  
    where (left,right) = partition (<= x) xs
```

# Map and filter versus list comprehension

- `map` and `filter` can be implemented using list comprehension

```
map f xs      = [f x | x <- xs]
filter p xs   = [x | x <- xs, p x]
```

- Converse holds too: `[expr | p <- s]` implemented as<sup>1</sup>

```
let fun p = expr in map fun s
```

## Example

```
? [2*x | (x,_) <- [(1,2),(3,4),(5,6)]]
[2, 6, 10] :: [Int]
```

```
? let fun (x,_) = 2*x in map fun [(1,2),(3,4),(5,6)]
[2, 6, 10] :: [Int]
```

---

<sup>1</sup>Equal only when pattern matching with `p` succeeds on all elements of `s`. Exercise: generalize to allow for failure.



## Comprehension (cont.)

- Guards require filter: `[expr | p <- xs, guard]` translated as

```
let fun p  = expr
    pred p = guard
in map fun (filter pred xs)
```

- Example

```
? [2 * x | x <- [1 .. 5], x > 2]
[6, 8, 10] :: [Int]
```

becomes

```
? let fun x  = 2 * x
    pred x = (x>2)
in map fun (filter pred [1 .. 5])
[6, 8, 10] :: [Int]
```

## An example with fold

- `foldr`: right-associative fold

$$\text{foldr } (\oplus) e [l_1, l_2, \dots, l_n] = l_1 \oplus (l_2 \oplus \dots \oplus (l_n \oplus e))$$

`foldr` :: (a -> b -> b) -> b -> [a] -> b

`foldr f e []` = e

`foldr f e (x:xs)` = f x (foldr f e xs)

- `foldl`: left-associative fold

$$\text{foldl } (\oplus) e [l_1, l_2, \dots, l_n] = ((e \oplus l_1) \oplus l_2) \oplus \dots \oplus l_n$$

`foldl` :: (b -> a -> b) -> b -> [a] -> b

`foldl f e []` = e

`foldl f e (x:xs)` = foldl f (f e x) xs

## Fold (cont.)

- No difference for associative functions (and  $e$  is neutral element)

```
? foldl (+) 0 [1,2,3]          -- ((0 + 1) + 2) + 3
6 :: Int
```

```
? foldr (+) 0 [1,2,3]          -- 1 + (2 + (3 + 0))
6 :: Int
```

- But not all (binary) functions are associative

```
? foldl (-) 0 [1,2,3]          -- ((0 - 1) - 2) - 3
-6 :: Int
```

```
? foldr (-) 0 [1,2,3]          -- 1 - (2 - (3 - 0))
2 :: Int
```

- How does one implement `length` with `foldr` and with `foldl`?

## Implementing length with foldr

$$\text{foldr } (\oplus) e [l_1, l_2, l_3] = l_1 \oplus (l_2 \oplus (l_3 \oplus e))$$

Solution with  $1 + (1 + (1 + 0))$

```
length xs = foldr (\_ y -> 1+y) 0 xs
```

```
? length ['a', 'b', 'c']
```

```
3 :: Int
```

Compare with the “standard” definition

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

where

```
length ['a', 'b', 'c'] = 1 + length ['b', 'c'] = ... = 1+(1+(1+0))
```

Solution with foldl: Exercise!

# Functions as “first-class objects”

- Simple examples (ignoring complications of type classes):

```
? :type \x -> x  
a -> a
```

```
? :type \x -> x + 1  
Int -> Int
```

- Composition as example:

```
(.)      :: (b -> c) -> (a -> b) -> (a -> c)  
(f . g) x = f (g x)
```

```
twice f = f . f
```

```
? :type twice (1+)  
Int -> Int
```

```
? twice (1+) 7  
9 :: Int
```

# Partial application

```
g :: Int -> Int -> Int
g x y = x + 2 * y
```

```
h :: Int -> Int
h = g 1
```

```
? h 10
21 :: Int
```

```
? map (g 10) [1,2,3,4,5]           -- Partial application
[12, 14, 16, 18, 20] :: [Int]
```

```
? map (10 'g') [1,2,3,4,5]         -- Left section
[12, 14, 16, 18, 20] :: [Int]
```

```
? map ('g' 10) [1,2,3,4,5]         -- Right section
[21, 22, 23, 24, 25] :: [Int]
```

```
? map (\x -> g x 10) [1,2,3,4,5]
[21, 22, 23, 24, 25] :: [Int]
```

## Reminder

- Zipper function

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

- Example  $zip\ [2,3,4]\ [4,5,78] = [(2,4), (3,5), (4,78)]$   
 $zip\ [2,3]\ [1,2,3] = [(2,1), (3,2)]$

- Uncurry

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = f
  where f (x,y) = g x y
```

- Example

```
g x y = x * y + 17
```

```
? uncurry g (3,4)
29 :: Int
```

# Case study: operations on vectors and matrices

- Vectors over `Int`

```
type Vector = [Int]
```

- Vector addition

```
vecAdd :: Vector -> Vector -> Vector
vecAdd (x:xs) (y:ys) = (x + y) : vecAdd xs ys
vecAdd _      _      = []
```

```
? vecAdd [1,2,3] [2,2,4]
[3, 4, 7] :: [Int]
```

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 2 \\ 2 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 7 \end{pmatrix}$$

Replace recursion with `map` and `zip`:

```
vecAdd v1 v2 = map (uncurry (+)) (zip v1 v2)
```



## zipWith = map + zip

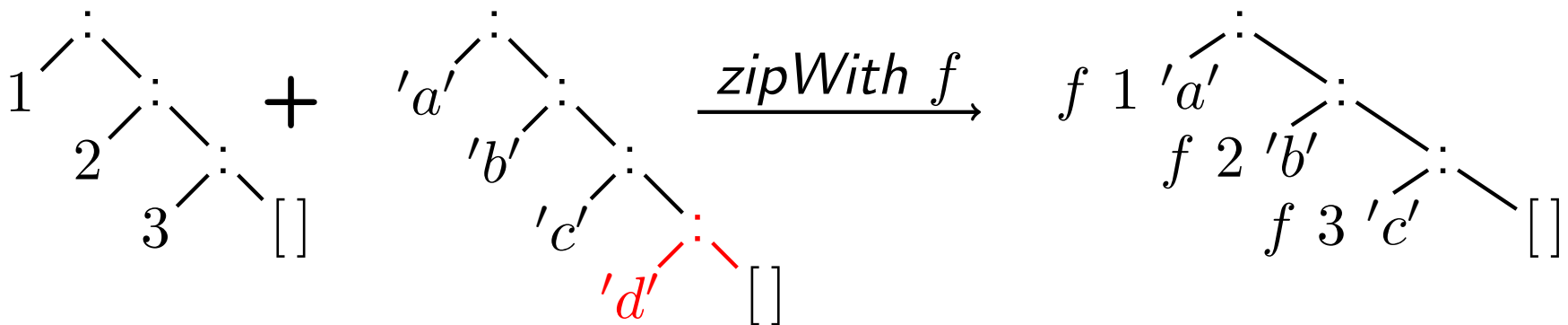
- Combination of `zip` and binary functions is common

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
```

- `map (uncurry f) (zip ... ...)` becomes `zipWith f ... ...`:

```
vecAdd :: Vector -> Vector -> Vector
vecAdd = zipWith (+)
```

- Visualisation of *zipWith*



## Matrix case study (cont.)

- $n \times m$  matrix

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{pmatrix}$$

can be represented **column-wise** using lists

$$[[a_{1,1}, a_{2,1}, \dots, a_{n,1}], [a_{1,2}, a_{2,2}, \dots, a_{n,2}], \dots, [a_{1,m}, a_{2,m}, \dots, a_{n,m}]]$$

`type Matrix = [Vector]`

- Addition of matrices

`matAdd :: Matrix -> Matrix -> Matrix`

`matAdd = zipWith vecAdd`

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} + \begin{pmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{pmatrix} = \begin{pmatrix} 8 & 14 \\ 10 & 16 \\ 12 & 18 \end{pmatrix}$$

```
? matAdd [[1,2,3],[4,5,6]] [[7,8,9],[10,11,12]]
[[8,10,12],[14,16,18]] :: [[Int]]
```

## Matrix case study (cont.)

- Constant vector of size  $n$

```
vconst :: Int -> Int -> Vector
vconst 0 _ = []
vconst n x = x : vconst (n - 1) x
```

$$\begin{pmatrix} x \\ x \\ x \\ \vdots \\ x \end{pmatrix}$$

- Unit matrix

```
unit :: Int -> Matrix
unit 0 = []
unit n =
  (1 : vconst (n - 1) 0)
  : map (0:) (unit (n - 1))
```

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

## Matrix case study (cont.)

- Constant vector of size  $n$

```
vconst :: Int -> Int -> Vector
vconst 0 _ = []
vconst n x = x : vconst (n - 1) x
```

$$\begin{pmatrix} x \\ x \\ x \\ \vdots \\ x \end{pmatrix}$$

- Unit matrix

```
unit :: Int -> Matrix
unit 0 = []
unit n =
  (1 : vconst (n - 1) 0)
  : map (0:) (unit (n - 1))
```

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

# Transposing a matrix

A list of columns is converted to a list of rows

$$\left( \begin{array}{c|c|c|c} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{2,2} & \dots & a_{n,m} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c|c} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \vdots & \vdots & & \vdots \\ a_{1,m} & a_{2,m} & \dots & a_{n,m} \end{array} \right)$$

```
tr :: Matrix -> Matrix
```

```
tr [] = []
```

```
tr [v] = map (\x -> [x]) v
```

```
tr (v:vs) = zipWith (:) v (tr vs)
```

$(\ ) \rightarrow (\ )$

$$\left( \begin{array}{c} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{array} \right) \rightarrow ( \begin{array}{c|c|c|c} a_{1,1} & a_{2,1} & \dots & a_{n,1} \end{array} )$$

```
tr [[1,2]]
```

```
[[1], [2]] :: [[Int]]
```

```
? tr [[1,2],[3,4]]
```

```
[[1, 3], [2, 4]] :: [[Int]]
```

$$\left( \begin{array}{c|c} \begin{array}{c} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{array} & vs \end{array} \right) \rightarrow \left( \begin{array}{c|c|c|c} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ \hline tr & vs & & \end{array} \right)$$

## Example of transposition

```
tr []      = []
tr [v]     = map (\x -> [x]) v
tr (v:vs)  = zipWith (:) v (tr vs)
```

$$\begin{aligned}
 tr \ [[1, 2], [3, 4]] &= zipWith \ (:) \ [1, 2] \ (tr \ [[3, 4]]) \\
 &= zipWith \ (:) \ [1, 2] \ (map \ (\lambda x. [x]) \ [3, 4]) \\
 &= zipWith \ (:) \ [1, 2] \ [[3], [4]] \\
 &= (1 : [3]) : zipWith \ (:) \ [2] \ [[4]] \\
 &= (1 : [3]) : ((2 : [4]) : zipWith \ (:) \ [] \ []) \\
 &= (1 : [3]) : ((2 : [4]) : []) \\
 &= [[1, 3], [2, 4]]
 \end{aligned}$$

## Scalar (dot) product of two vectors

- Sum of product of vectors  $v$  and  $w$ :  $v \cdot w = \sum_i v_i w_i$
- Version 1: Pedestrian loop

```
skProd :: Vector -> Vector -> Int
```

```
skProd xs ys = loop xs ys 0
```

```
  where
```

```
    loop [] [] p = p
```

```
    loop (x:xs) (y:ys) p = loop xs ys (p + x * y)
```

```
? skProd [2,3] [4,5]
```

```
23
```

## Scalar (dot) product of two vectors

- Sum of product of vectors  $v$  and  $w$ :  $v \cdot w = \sum_i v_i w_i$

- Version 1: Pedestrian loop

```
skProd :: Vector -> Vector -> Int
skProd xs ys = loop xs ys 0
  where
    loop [] [] p = p
    loop (x:xs) (y:ys) p = loop xs ys (p + x * y)
```

- Version 2: Explicit recursion, covering all cases

```
skProd :: Vector -> Vector -> Int
skProd (x:xs) (y:ys) = x*y + skProd xs ys
skProd _ _ = 0
```

? skProd [2,3] [4,5]  
23



# Scalar (dot) product of two vectors

- Sum of product of vectors  $v$  and  $w$ :  $v \cdot w = \sum_i v_i w_i$

- Version 1: Pedestrian loop

```
skProd :: Vector -> Vector -> Int
skProd xs ys = loop xs ys 0
  where
    loop [] [] p = p
    loop (x:xs) (y:ys) p = loop xs ys (p + x * y)
```

- Version 2: Explicit recursion, covering all cases

```
skProd :: Vector -> Vector -> Int
skProd (x:xs) (y:ys) = x*y + skProd xs ys
skProd _ _ = 0
```

- Version 3: Idiomatic with library functions

```
skProd :: Vector -> Vector -> Int
skProd v w = sum (zipWith (*) v w)
```

? skProd [2,3] [4,5]  
23

# Matrix multiplication

We first multiply an  $n \times m$  matrix  $A$  with a vector  $b$  of size  $m$ .

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^m a_{1,i} b_i \\ \sum_{i=1}^m a_{2,i} b_i \\ \vdots \\ \sum_{i=1}^m a_{n,i} b_i \end{pmatrix}$$

$\Rightarrow$  scalar product of  $A$ 's rows (columns of  $\text{tr } A$ ) with  $b$

```
vecMult :: Matrix -> Vector -> Vector
vecMult a b = map ('skProd' b) (tr a)
```

```
? vecMult [[1,2,3],[4,5,6]] [7,8]
[39,54,69] :: [Int]
```

# Matrix multiplication (cont.)

Matrix multiplication iterates `vecMult`  $A$  over an  $m \times k$  matrix  $B$

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,m} \\ a_{2,1} & \dots & a_{2,m} \\ \vdots & & \vdots \\ a_{n,1} & \dots & a_{n,m} \end{pmatrix} \cdot \left( \begin{array}{c|c|c} b_{1,1} & b_{1,2} & \dots \\ b_{2,1} & b_{2,2} & \dots \\ \vdots & \vdots & \\ b_{m,1} & b_{m,2} & \dots \end{array} \right) = \left( \begin{array}{c|c|c} \sum_{i=1}^m a_{1,i} b_{i,1} & \sum_{i=1}^m a_{1,i} b_{i,2} & \dots \\ \sum_{i=1}^m a_{2,i} b_{i,1} & \sum_{i=1}^m a_{2,i} b_{i,2} & \dots \\ \vdots & \vdots & \\ \sum_{i=1}^m a_{n,i} b_{i,1} & \sum_{i=1}^m a_{n,i} b_{i,2} & \dots \end{array} \right)$$

```
matMult :: Matrix -> Matrix -> Matrix
matMult a b = map (vecMult a) b
```

```
? matMult [[1,2,3],[4,5,6]] [[7,8],[9,10]]
[[39,54,69],[49,68,87]] :: [[Int]]
```

Exercise: Scalar multiplication for vectors and matrices!

# Conclusion

- First-order programming
  - ▶ Programming with elements of base types, like True or 13
  - ▶ Close to machine architecture
- Higher-order programming
  - ▶ Functions are first-class objects
- Increases abstraction and ways of constructing programs
- Other advantages like reusability and rapid prototyping



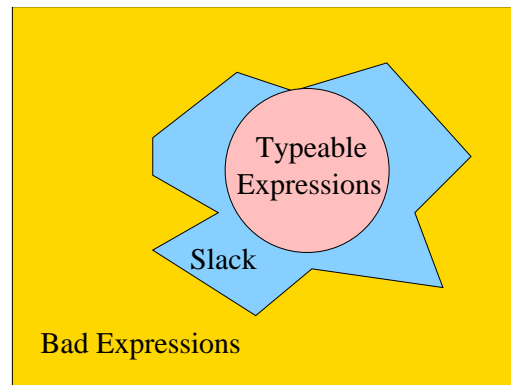
# Typing

# Type checking: an overview

- Type checking should prevent “dangerous expressions”

$2 + True$ ,  $[2] : [3]$ ,  $2 ++ [3, 4]$ ,  $fst(2)$

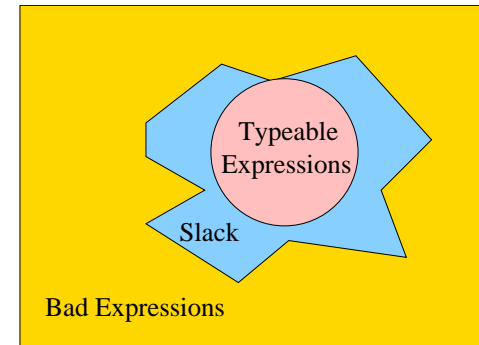
- Dangerous expressions**  $\implies$  runtime error
- Problem of which expressions are good (= non-dangerous) is undecidable



Type systems in practice are conservative: they type only a subset of good expressions (pink), rejecting some good ones (blue).

# Typing overview (cont.)

- Objectives for a type checker
  - ▶ quick, decidable, static analysis
  - ▶ permit as much generality/re-usability as possible
  - ▶ prevent runtime errors: subject reduction



If  $e \hookrightarrow e'$  and  $\vdash e :: \tau$ , then  $\vdash e' :: \tau$ .

- Topic is very rich (theory of programming)
- We examine here a simplified language: 'Mini-Haskell'



# Mini-Haskell — syntax

- Programs are terms (let variables  $\mathcal{V}$  and integers  $\mathcal{Z}$  be given)

$$\begin{aligned}
 t \quad ::= & \quad \mathcal{V} \mid (\lambda x. t) \mid (t_1 t_2) \mid \\
 & \quad \textit{True} \mid \textit{False} \mid (\mathbf{iszero} \ t) \mid \\
 & \quad \mathcal{Z} \mid (t_1 + t_2) \mid (t_1 * t_2) \mid (\mathbf{if} \ t_0 \ \mathbf{then} \ t_1 \ \mathbf{else} \ t_2) \mid \\
 & \quad (t_1, t_2) \mid (\mathbf{fst} \ t) \mid (\mathbf{snd} \ t)
 \end{aligned}$$

- Small but powerful language. Corresponds to fragment of Haskell

```
iszero :: Int -> Bool
iszero x = x == 0
```

```
? (if (iszero (2*0)) then (fst (2,3)) else (snd (2,3)))
2 :: Int
```

```
? ((\f x -> (if (iszero x) then (f 2) else (f 3)))
      ((\x y -> y + x) 2) 5)
5 :: Int
```

- Not all terms are meaningful, e.g.  $(\mathbf{iszero} (\lambda x. x))$

# Mini-Haskell — comments

- Core is  $\lambda$ -calculus: variables, abstraction, and application

$$(\lambda x. ((x\ y) (\lambda y. (x\ y))))$$

- Additional syntax and types can be easily added, e.g.,

$$\&\&, ||, \text{Strings}, \dots$$

- We employ syntactic sugar, like omitting parenthesis

$$x\ y\ z \quad \text{instead} \quad ((x\ y)\ z)$$
$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \quad \text{instead} \quad (\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3))$$

- A substantial simplification of Haskell — but the central core!

# Typing

- Types ( $\mathcal{V}_{\mathcal{T}}$  is a set of type variables:  $a, b, \dots$ )

$$\tau ::= \mathcal{V}_{\mathcal{T}} \mid Bool \mid Int \mid (\tau, \tau) \mid (\tau \rightarrow \tau)$$

- Examples:  $a, Int, (Int, Bool), ((a \rightarrow Int) \rightarrow (a, a)), \dots$
- Type system notation based on typing judgement:  $\Gamma \vdash t :: \tau$ 
  - ▶  $\Gamma$  is a set of bindings  $x_i : \tau_i$ , mapping variables to types. Intuitively  $\Gamma$  represents a kind of typing “symbol table”.
  - ▶  $t$  is a term
  - ▶  $\tau$  is a type
- Intuition: given symbol table  $\Gamma$ , then term  $t$  has type  $\tau$ .

$$x : Int \vdash x + 2 :: Int \qquad x : Int, f : Bool \rightarrow Bool \not\vdash f\ x :: Bool$$

# Typing — proof system

- Proof rules formulated in terms of type judgements  $J$

$$\frac{J_1 \quad \dots \quad J_n}{J}$$

- Example axiom ( $n \in \{\dots, -1, 0, 1, \dots\}$ ):  $\frac{}{\Gamma \vdash n :: Int}^{Int}$

- Example rule ( $op \in \{+, *\}$ ):

$$\frac{\Gamma \vdash t_1 :: Int \quad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1 \text{ op } t_2) :: Int}^{BinOp}$$

- Proofs built from rules and axioms

$$\frac{\frac{}{x : Int \vdash 2 :: Int}^{Int} \quad \frac{\frac{}{x : Int \vdash (x + 1) :: Int}^{BinOp}}{\vdots}^{BinOp}}{x : Int \vdash (2 + (x + 1)) :: Int}^{BinOp}$$

# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App}$$

Examples:

$$\vdash (\lambda x. x) :: a \rightarrow a$$

$$\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a$$

# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App}$$

Examples:

$$\frac{x : a \vdash x :: a}{\vdash (\lambda x. x) :: a \rightarrow a} \text{Abs}$$

$$\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a$$

# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{\dots, x : \tau, \dots \vdash x :: \tau}{\text{Var}}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App}$$

Examples:

$$\frac{\frac{x : a \vdash x :: a}{\text{Var}}}{\vdash (\lambda x. x) :: a \rightarrow a} \text{Abs}$$

$$\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a$$

# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App}$$

Examples:

$$\frac{\frac{}{x : a \vdash x :: a} \text{Var}}{\vdash (\lambda x. x) :: a \rightarrow a} \text{Abs}$$

$$\frac{x : a \vdash \lambda y. x :: b \rightarrow a}{\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a} \text{Abs}$$



# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}$$

Examples:

$$\frac{\frac{}{x : a \vdash x :: a} \text{Var}}{\vdash (\lambda x. x) :: a \rightarrow a} \text{Abs}$$

$$\frac{\frac{x : a, y : b \vdash x :: a}{x : a \vdash \lambda y. x :: b \rightarrow a} \text{Abs}}{\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a} \text{Abs}$$

# Rules for core $\lambda$ -calculus

**Axiom:**

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var}$$

**Abstraction** ( $x \notin \Gamma$ ):

$$\frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs}$$

**Application:**

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App}$$

Examples:

$$\frac{\frac{}{x : a \vdash x :: a} \text{Var}}{\vdash (\lambda x. x) :: a \rightarrow a} \text{Abs}$$

$$\frac{\frac{\frac{}{x : a, y : b \vdash x :: a} \text{Var}}{x : a \vdash \lambda y. x :: b \rightarrow a} \text{Abs}}{\vdash \lambda x. \lambda y. x :: a \rightarrow b \rightarrow a} \text{Abs}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \textit{App}
 \end{array}$$

$$\vdash \lambda x. \lambda y. \lambda z. (x \ z) \ (y \ z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c}{\phantom{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)}} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \textit{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{\overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a \vdash (x z) (y z) :: c}^{\Gamma=} \textit{Abs}}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \textit{Abs} \\
 \frac{\textit{Abs}}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \textit{Abs} \\
 \frac{\textit{Abs}}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \textit{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{\Gamma \vdash x z :: ? \rightarrow c \quad \Gamma \vdash y z :: ?}{\Gamma \vdash (x z) (y z) :: c} \text{App} \\
 \frac{\overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a}^{\Gamma=} \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$



# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{\Gamma \vdash x :: ?? \rightarrow (? \rightarrow c) \quad \Gamma \vdash z :: ??}{\Gamma \vdash x z :: ? \rightarrow c} \text{App} \quad \Gamma \vdash y z :: ? \\
 \hline
 \frac{\Gamma = \overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a} \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \hline
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \Gamma \vdash z :: a \\
 \hline
 \Gamma \vdash x z :: b \rightarrow c \quad \Gamma \vdash y z :: b \\
 \hline
 \Gamma \vdash (x z) (y z) :: c \quad \text{App} \\
 \hline
 \overbrace{\Gamma =}^{\text{}} \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \hline
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var} \\
 \hline
 \Gamma \vdash x z :: b \rightarrow c \quad \Gamma \vdash y z :: b \\
 \hline
 \Gamma \vdash (x z) (y z) :: c \quad \text{App} \\
 \hline
 \overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a}^{\Gamma=} \vdash (x z) (y z) :: c \\
 \hline
 x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c \quad \text{Abs} \\
 \hline
 x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c) \quad \text{Abs} \\
 \hline
 \vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \quad \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \text{App} \\
 \\
 \frac{\frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var}}{\Gamma \vdash x \ z :: b \rightarrow c} \text{App} \quad \frac{\Gamma \vdash y :: ??? \rightarrow b \quad \Gamma \vdash z :: ???}{\Gamma \vdash y \ z :: b} \text{App} \\
 \\
 \frac{\overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a}^{\Gamma=} \vdash (x \ z) \ (y \ z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x \ z) \ (y \ z) :: a \rightarrow c} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x \ z) \ (y \ z) :: a \rightarrow c}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x \ z) \ (y \ z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x \ z) \ (y \ z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x \ z) \ (y \ z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var} \quad \frac{}{\Gamma \vdash y :: a \rightarrow b} \text{Var} \\
 \frac{}{\Gamma \vdash x z :: b \rightarrow c} \text{App} \quad \frac{}{\Gamma \vdash y z :: b} \text{App} \\
 \frac{}{\Gamma =} \\
 \frac{\overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a \vdash (x z) (y z) :: c}^{\Gamma =}}{\Gamma \vdash x z :: b \rightarrow c} \text{App} \\
 \frac{}{\Gamma \vdash y z :: b} \text{App} \\
 \frac{}{\Gamma \vdash (x z) (y z) :: c} \text{Abs} \\
 \frac{}{\Gamma \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \frac{}{\Gamma \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
 \frac{}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

# A larger example

$$\begin{array}{c}
\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App} \\
\\
\frac{\frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var}}{\Gamma \vdash x z :: b \rightarrow c} \text{App} \quad \frac{\frac{}{\Gamma \vdash y :: a \rightarrow b} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var}}{\Gamma \vdash y z :: b} \text{App} \\
\frac{\Gamma \vdash x z :: b \rightarrow c \quad \Gamma \vdash y z :: b}{\Gamma \vdash (x z) (y z) :: c} \text{App} \\
\frac{\overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a}^{\Gamma=} \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
\frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
\frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
\end{array}$$

# A larger example

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

$$\begin{array}{c}
 \frac{\frac{}{\Gamma \vdash x :: a \rightarrow (b \rightarrow c)} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var}}{\Gamma \vdash x z :: b \rightarrow c} \text{App} \quad \frac{\frac{}{\Gamma \vdash y :: a \rightarrow b} \text{Var} \quad \frac{}{\Gamma \vdash z :: a} \text{Var}}{\Gamma \vdash y z :: b} \text{App} \\
 \frac{\Gamma = \overbrace{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b, z : a} \vdash (x z) (y z) :: c}{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c), y : a \rightarrow b \vdash \lambda z. (x z) (y z) :: a \rightarrow c}{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs} \\
 \frac{x : a \rightarrow (b \rightarrow c) \vdash \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow b) \rightarrow (a \rightarrow c)}{\vdash \lambda x. \lambda y. \lambda z. (x z) (y z) :: (a \rightarrow (b \rightarrow c)) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)} \text{Abs}
 \end{array}$$

Exercise:  $\vdash (\lambda z. z) (\lambda x. \lambda y. x) :: a \rightarrow b \rightarrow a$

## Examples in ghc

```
? :type \x -> x  
\x -> x :: a -> a
```

```
? :type \x y -> x  
\x y -> x :: a -> b -> a
```

```
? :type \x y z -> x z (y z)  
\x y z -> x z (y z) :: (a -> b -> c) -> (a -> b) -> a -> c
```

```
? :type (\z -> z) (\x y -> x)  
(\z -> z) (\x y -> x) :: a -> b -> a
```



## Further typing rules for mini-Haskell

- Base types  $\frac{}{\Gamma \vdash n :: Int}^{Int} \quad \frac{}{\Gamma \vdash True :: Bool}^{True} \quad \frac{}{\Gamma \vdash False :: Bool}^{False}$

- Operations (**op**  $\in \{+, *\}$ ):

$$\frac{\Gamma \vdash t :: Int}{\Gamma \vdash (\text{iszero } t) :: Bool}^{iszero} \quad \frac{\Gamma \vdash t_1 :: Int \quad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1 \text{ **op** } t_2) :: Int}^{BinOp}$$

$$\frac{\Gamma \vdash t_0 :: Bool \quad \Gamma \vdash t_1 :: \tau \quad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\text{if } t_0 \text{ then } t_1 \text{ else } t_2) :: \tau}^{if}$$

- Tuples

$$\frac{\Gamma \vdash t_1 :: \tau_1 \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}^{Tuple} \quad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\text{fst } t) :: \tau_1}^{fst} \quad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash (\text{snd } t) :: \tau_2}^{snd}$$

- Example

$$\frac{\frac{}{x : Int \vdash x :: Int}^{Var} \quad \frac{}{x : Int \vdash 2 :: Int}^{Int}}{\frac{}{x : Int \vdash x + 2 :: Int}^{BinOp}}^{Abs} \vdash \lambda x. x + 2 :: Int \rightarrow Int$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \textit{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
  2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
  3. Solve constraints (unification) to get possible types.
- Example:

$(\lambda z. z) (\lambda x. x)$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \textit{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
  2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
  3. Solve constraints (unification) to get possible types.
- Example:

$$\vdash (\lambda z. z) (\lambda x. x) :: \tau_0$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \textit{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
  2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
  3. Solve constraints (unification) to get possible types.
- Example:

$$\frac{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0 \quad \vdash \lambda x. x :: \tau_1}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \textit{App}$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \textit{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 \ t_2) :: \tau} \textit{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
  2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
  3. Solve constraints (unification) to get possible types.
- Example:

$$\frac{\frac{z : \tau_1 \vdash z :: \tau_0}{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0} \textit{Abs} \quad \vdash \lambda x. x :: \tau_1}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \textit{App}$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
  2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
  3. Solve constraints (unification) to get possible types.

- Example:

$$\tau_1 = \tau_0$$

$$\frac{\frac{\frac{}{z : \tau_1 \vdash z :: \tau_0} \text{Var}}{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0} \text{Abs} \quad \vdash \lambda x. x :: \tau_1}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \text{App}$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type

1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
3. Solve constraints (unification) to get possible types.

- Example:

$$\frac{\frac{\frac{}{z : \tau_1 \vdash z :: \tau_0} \text{Var}}{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0} \text{Abs} \quad \frac{x : \tau_2 \vdash x :: \tau_3}{\vdash \lambda x. x :: \tau_1} \text{Abs}}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \text{App}$$

$$\tau_1 = \tau_0$$

$$\tau_1 = \tau_2 \rightarrow \tau_3$$

# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type

1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
3. Solve constraints (unification) to get possible types.

- Example:

$$\frac{\frac{\frac{}{z : \tau_1 \vdash z :: \tau_0} \text{Var}}{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0} \text{Abs}}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \text{App}$$

$$\frac{\frac{\frac{}{x : \tau_2 \vdash x :: \tau_3} \text{Var}}{\vdash \lambda x. x :: \tau_1} \text{Abs}}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \text{App}$$

$$\tau_1 = \tau_0$$

$$\tau_1 = \tau_2 \rightarrow \tau_3$$

$$\tau_2 = \tau_3$$



# Type inference

$$\begin{array}{c}
 \frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \text{Var} \quad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \sigma \rightarrow \tau} \text{Abs} \quad \frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1 t_2) :: \tau} \text{App}
 \end{array}$$

- Syntax-directed typing rules specify algorithm for computing type

1. Start with judgement  $\vdash t :: \tau_0$  with type variable  $\tau_0$ .
2. Build derivation tree bottom-up by applying the matching rule.  
Introduce fresh type variables and collect constraints if needed.
3. Solve constraints (unification) to get possible types.

- Example:

$$\begin{array}{c}
 \frac{}{z : \tau_1 \vdash z :: \tau_0} \text{Var} \quad \frac{}{x : \tau_2 \vdash x :: \tau_3} \text{Var} \\
 \frac{}{\vdash \lambda z. z :: \tau_1 \rightarrow \tau_0} \text{Abs} \quad \frac{}{\vdash \lambda x. x :: \tau_1} \text{Abs} \\
 \frac{}{\vdash (\lambda z. z) (\lambda x. x) :: \tau_0} \text{App}
 \end{array}
 \quad
 \begin{array}{l}
 \tau_1 = \tau_0 \\
 \tau_1 = \tau_2 \rightarrow \tau_3 \\
 \tau_2 = \tau_3 \\
 \text{i.e., } \tau_0 = \tau_3 \rightarrow \tau_3
 \end{array}$$

E.g.,  $\tau_0 = a \rightarrow a$  and  $\tau_0 = \text{Int} \rightarrow \text{Int}$  are correct types.

# Type inference example

$\lambda x. \mathbf{iszero} \ ((\mathbf{fst} \ x) \ (2, \mathit{True}))$

# Type inference example

$$\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} \ x) \ (2, \mathit{True})) :: \tau_0$$

**Constraints:**

# Type inference example

$$\frac{\overbrace{x : \tau_1}^{\Gamma=} \vdash \mathbf{iszero} ((\mathbf{fst} \ x) \ (2, \mathit{True})) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} \ x) \ (2, \mathit{True})) :: \tau_0} \textit{Abs}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

# Type inference example

$$\begin{array}{c}
 \Gamma \vdash (\mathbf{fst} \ x) \ (2, \mathit{True}) :: \mathit{Int} \\
 \hline
 \text{App} \\
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} \ ((\mathbf{fst} \ x) \ (2, \mathit{True})) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} \ ((\mathbf{fst} \ x) \ (2, \mathit{True})) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = \mathit{Bool}$$

# Type inference example

$$\begin{array}{c}
 \Gamma \vdash \mathbf{fst} \, x :: \tau_3 \rightarrow Int \qquad \Gamma \vdash (2, True) :: \tau_3 \\
 \hline
 \Gamma \vdash (\mathbf{fst} \, x) \, (2, True) :: Int \quad App \\
 \hline
 \Gamma = \\
 \underbrace{x : \tau_1} \vdash \mathbf{iszero} \, ((\mathbf{fst} \, x) \, (2, True)) :: \tau_2 \quad App \\
 \hline
 \vdash \lambda x. \mathbf{iszero} \, ((\mathbf{fst} \, x) \, (2, True)) :: \tau_0 \quad Abs
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

# Type inference example

$$\begin{array}{c}
 \frac{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \qquad \Gamma \vdash (2, True) :: \tau_3 \\
 \hline
 \Gamma \vdash (\mathbf{fst} x) (2, True) :: Int \qquad \text{App} \\
 \hline
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{App} \quad \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \\
 \frac{}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \\
 \frac{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int \quad \Gamma \vdash (2, True) :: \tau_3}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \frac{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$



# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \quad \frac{\Gamma \vdash 2 :: \tau_5 \quad \Gamma \vdash True :: \tau_6}{\Gamma \vdash (2, True) :: \tau_3} \text{Tuple} \\
 \frac{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \quad \frac{}{\Gamma \vdash (2, True) :: \tau_3} \text{App} \\
 \frac{}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$

$$\tau_3 = (\tau_5, \tau_6)$$

# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \quad \frac{}{\Gamma \vdash 2 :: \tau_5} \text{Int} \quad \frac{}{\Gamma \vdash True :: \tau_6} \\
 \frac{}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \quad \frac{}{\Gamma \vdash (2, True) :: \tau_3} \text{Tuple} \\
 \frac{}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$

$$\tau_3 = (\tau_5, \tau_6)$$

$$\tau_5 = Int$$

# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \qquad \frac{}{\Gamma \vdash 2 :: \tau_5} \text{Int} \qquad \frac{}{\Gamma \vdash True :: \tau_6} \text{True} \\
 \hline
 \frac{}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \qquad \frac{}{\Gamma \vdash (2, True) :: \tau_3} \text{Tuple} \\
 \hline
 \frac{}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \hline
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$

$$\tau_3 = (\tau_5, \tau_6)$$

$$\tau_5 = Int$$

$$\tau_6 = Bool$$

# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \quad \frac{}{\Gamma \vdash 2 :: \tau_5} \text{Int} \quad \frac{}{\Gamma \vdash True :: \tau_6} \text{True} \\
 \hline
 \frac{}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \quad \frac{}{\Gamma \vdash (2, True) :: \tau_3} \text{Tuple} \\
 \hline
 \frac{}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \hline
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$

$$\tau_3 = (\tau_5, \tau_6)$$

$$\tau_5 = Int$$

$$\tau_6 = Bool$$

## Most general type:

$$\tau_0 = (((Int, Bool) \rightarrow Int), a) \rightarrow Bool$$

# Type inference example

$$\begin{array}{c}
 \frac{}{\Gamma \vdash x :: ((\tau_3 \rightarrow Int), \tau_4)} \text{Var} \quad \frac{}{\Gamma \vdash 2 :: \tau_5} \text{Int} \quad \frac{}{\Gamma \vdash True :: \tau_6} \text{True} \\
 \hline
 \frac{}{\Gamma \vdash \mathbf{fst} x :: \tau_3 \rightarrow Int} \text{fst} \quad \frac{}{\Gamma \vdash (2, True) :: \tau_3} \text{Tuple} \\
 \hline
 \frac{}{\Gamma \vdash (\mathbf{fst} x) (2, True) :: Int} \text{App} \\
 \hline
 \frac{\Gamma = \overbrace{x : \tau_1} \vdash \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_2}{\vdash \lambda x. \mathbf{iszero} ((\mathbf{fst} x) (2, True)) :: \tau_0} \text{Abs}
 \end{array}$$

## Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = Bool$$

$$\tau_1 = ((\tau_3 \rightarrow Int), \tau_4)$$

$$\tau_3 = (\tau_5, \tau_6)$$

$$\tau_5 = Int$$

$$\tau_6 = Bool$$

## Most general type:

$$\tau_0 = (((Int, Bool) \rightarrow Int), a) \rightarrow Bool$$

## Exercise:

Infer the type of

$$\lambda x. \lambda y. (\mathbf{iszero} (x y), x 3)$$

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\lambda x. (\mathbf{fst} \ x, \mathbf{iszero} \ x)$$

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\vdash \lambda x. (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_0$$

- Constraints:

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\frac{x : \tau_1 \vdash (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_2}{\vdash \lambda x. (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_0} \text{Abs}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$



## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\frac{
 \frac{
 x : \tau_1 \vdash \mathbf{fst} \ x :: \tau_3 \quad x : \tau_1 \vdash \mathbf{iszero} \ x :: \tau_4
 }{
 x : \tau_1 \vdash (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_2
 } \textit{Tuple}
 }{
 \vdash \lambda x. (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_0
 } \textit{Abs}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\begin{array}{c}
 \dfrac{x : \tau_1 \vdash x :: (\tau_3, \tau_5)}{x : \tau_1 \vdash \mathbf{fst} \, x :: \tau_3} \text{fst} \qquad x : \tau_1 \vdash \mathbf{iszero} \, x :: \tau_4 \\
 \hline
 \dfrac{x : \tau_1 \vdash (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_2}{\vdash \lambda x. (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_0} \text{Abs} \quad \text{Tuple}
 \end{array}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\begin{array}{c}
 \frac{}{x : \tau_1 \vdash x :: (\tau_3, \tau_5)} \text{Var} \\
 \frac{}{x : \tau_1 \vdash \mathbf{fst} \, x :: \tau_3} \text{fst} \qquad x : \tau_1 \vdash \mathbf{iszero} \, x :: \tau_4 \\
 \hline
 x : \tau_1 \vdash (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_2 \quad \text{Tuple} \\
 \hline
 \vdash \lambda x. (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_0 \quad \text{Abs}
 \end{array}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

$$\tau_1 = (\tau_3, \tau_5)$$

## When type inference fails

- Some terms are untypeable.

Type inference fails to build inference tree or solve constraints.

- Example:

$$\begin{array}{c}
 \frac{}{x : \tau_1 \vdash x :: (\tau_3, \tau_5)} \text{Var} \\
 \hline
 x : \tau_1 \vdash \mathbf{fst} \ x :: \tau_3 \quad \text{fst}
 \end{array}
 \qquad
 \frac{x : \tau_1 \vdash x :: \mathit{Int}}{x : \tau_1 \vdash \mathbf{iszero} \ x :: \tau_4} \text{iszero}$$

$$\frac{x : \tau_1 \vdash (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_2}{\vdash \lambda x. (\mathbf{fst} \ x, \mathbf{iszero} \ x) :: \tau_0} \text{Abs}$$

$\tau_4$   
*Tuple*

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

$$\tau_1 = (\tau_3, \tau_5)$$

$$\tau_4 = \mathit{Bool}$$

## When type inference fails

- Some terms are untypeable.  
Type inference fails to build inference tree or solve constraints.
- Example:

$$\begin{array}{c}
 \frac{}{x : \tau_1 \vdash x :: (\tau_3, \tau_5)} \text{Var} \quad \frac{}{x : \tau_1 \vdash x :: \text{Int}} \text{Var} \\
 \frac{}{x : \tau_1 \vdash \mathbf{fst} \, x :: \tau_3} \text{fst} \quad \frac{}{x : \tau_1 \vdash \mathbf{iszero} \, x :: \tau_4} \text{iszero} \\
 \hline
 \frac{}{x : \tau_1 \vdash (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_2} \text{Tuple} \\
 \hline
 \frac{}{\vdash \lambda x. (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_0} \text{Abs}
 \end{array}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

$$\tau_1 = (\tau_3, \tau_5)$$

$$\tau_4 = \text{Bool}$$

$$\tau_1 = \text{Int}$$

## When type inference fails

- Some terms are untypeable.

Type inference fails to build inference tree or solve constraints.

- Example:

$$\begin{array}{c}
 \frac{}{x : \tau_1 \vdash x :: (\tau_3, \tau_5)} \text{Var} \qquad \frac{}{x : \tau_1 \vdash x :: \text{Int}} \text{Var} \\
 \frac{}{x : \tau_1 \vdash \mathbf{fst} \, x :: \tau_3} \text{fst} \qquad \frac{}{x : \tau_1 \vdash \mathbf{iszero} \, x :: \tau_4} \text{iszero} \\
 \hline
 \frac{}{x : \tau_1 \vdash (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_2} \text{Tuple} \\
 \hline
 \frac{}{\vdash \lambda x. (\mathbf{fst} \, x, \mathbf{iszero} \, x) :: \tau_0} \text{Abs}
 \end{array}$$

- Constraints:

$$\tau_0 = \tau_1 \rightarrow \tau_2$$

$$\tau_2 = (\tau_3, \tau_4)$$

$$\tau_1 = (\tau_3, \tau_5)$$

$$\tau_4 = \text{Bool}$$

$$\tau_1 = \text{Int}$$

$(\tau_3, \tau_5)$  and  $\text{Int}$  do not unify.

Constraints have no solution,  
so  $\lambda x. (\mathbf{fst} \, x, \mathbf{iszero} \, x)$  untypeable.

# Self application

- **Self application** means “applying a function  $f$  to itself”.
- Self application  $\lambda f. f f$  is not typeable.

$$\begin{array}{c}
 \frac{}{f : \tau_1 \vdash f :: \tau_3 \rightarrow \tau_2} \text{Var} \quad \frac{}{f : \tau_1 \vdash f :: \tau_3} \text{Var} \\
 \hline
 \frac{}{f : \tau_1 \vdash f f :: \tau_2} \text{App} \\
 \hline
 \frac{}{\vdash \lambda f. f f :: \tau_0} \text{Abs}
 \end{array}
 \quad
 \begin{array}{l}
 \tau_0 = \tau_1 \rightarrow \tau_2 \\
 \tau_1 = \tau_3 \rightarrow \tau_2 \\
 \tau_1 = \tau_3
 \end{array}$$

- $\tau_3 = \tau_3 \rightarrow \tau_2$  requires infinite function type  
 $((\dots (\dots \rightarrow \tau_2) \rightarrow \tau_2) \rightarrow \tau_2)$ , but Haskell types are finite.

So, there is no solution to the constraints.

- Compare with natural numbers:  
 $x = x + 1$  has only solution  $\infty$ , but  $\infty \notin \mathbb{N}$ .

## GHC examples

```
? :t \x -> iszero ((fst x) (2 :: Int, True))
\x -> iszero ((fst x) (2 :: Int, True))
:: ((Int, Bool) -> Int, b) -> Bool
```

```
? :t (\n -> if iszero n then 1 else 2*n) ((\x -> x+2) (fst (2,True)))
(\n -> if iszero n then 1 else 2*n) ((\x -> x+2) (fst (2,True))) :: Int
```

```
? (\n -> if iszero n then 1 else 2*n) ((\x -> x+2) (fst (2,True)))
8 :: Int
```

```
? :t \p -> (snd p) (fst p)
\p -> snd p (fst p) :: (a,a -> b) -> b
```

```
? :t \x -> (fst x, iszero x)
ERROR: Couldn't match expected type 'Int' with actual type '(t0, b0)'
```

```
? :t \x -> x x
ERROR: Occurs check: cannot construct the infinite type: t1 = t1 -> t0
```



# Curry-Howard isomorphism

- **Propositions as types**

- ▶ Type constructor “ $\rightarrow$ ” corresponds to propositional logic connective “ $\rightarrow$ ”
- ▶ Atomic types correspond to propositional variables

- Rules correspond to those for (minimal) propositional logic

$$\frac{}{\dots, \tau, \dots \vdash \tau} \text{ axiom} \quad \frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \rightarrow \tau} \rightarrow\text{-I} \quad \frac{\Gamma \vdash \sigma \rightarrow \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau} \rightarrow\text{-E}$$

- Example

$$\frac{\frac{\frac{}{\tau, \sigma \vdash \tau} \text{ axiom}}{\tau \vdash \sigma \rightarrow \tau} \rightarrow\text{-I}}{\vdash \tau \rightarrow \sigma \rightarrow \tau} \rightarrow\text{-I}$$

- Correspondence actually quite deep

# Summary of Mini-Haskell Typing Rules

$$\frac{}{\dots, x : \tau, \dots \vdash x :: \tau} \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x. t :: \sigma \rightarrow \tau} \textit{Abs}$$

$$\frac{\Gamma \vdash t_1 :: \sigma \rightarrow \tau \quad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1 t_2 :: \tau} \textit{App} \qquad \frac{\Gamma \vdash t :: \textit{Int}}{\Gamma \vdash \mathbf{iszero} \, t :: \textit{Bool}} \textit{iszero}$$

$$\frac{}{\Gamma \vdash n :: \textit{Int}} \textit{Int} \qquad \frac{}{\Gamma \vdash \textit{True} :: \textit{Bool}} \textit{True} \qquad \frac{}{\Gamma \vdash \textit{False} :: \textit{Bool}} \textit{False}$$

$$\frac{\Gamma \vdash t_1 :: \textit{Int} \quad \Gamma \vdash t_2 :: \textit{Int}}{\Gamma \vdash (t_1 \mathbf{op} t_2) :: \textit{Int}} \textit{BinOp} \quad \text{for } \mathbf{op} \in \{+, *\}$$

$$\frac{\Gamma \vdash t_0 :: \textit{Bool} \quad \Gamma \vdash t_1 :: \tau \quad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \mathbf{if} \, t_0 \mathbf{then} \, t_1 \mathbf{else} \, t_2 :: \tau} \textit{if}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \textit{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \mathbf{fst} \, t :: \tau_1} \textit{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \mathbf{snd} \, t :: \tau_2} \textit{snd}$$

# Type Classes

# Monomorphic versus polymorphic

- Some functions are **monomorphic**

```
xor x y = (x || y) && (not (x && y))
```

```
? :type xor  
xor :: Bool -> Bool -> Bool
```

- Others are **polymorphic**

```
[]      ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

```
? :type (++)  
(++) :: [a] -> [a] -> [a]
```

- Monomorphic or polymorphic?

```
allEqual x y z = (x == y) && (y == z)
```

## Example (cont.)

- Type of `allEqual x y z = (x == y) && (y == z) ?`

```
? allEqual 4 (2 + 2) (1+3)
```

```
True :: Bool
```

```
? allEqual "hi there" ("hi " ++ "there") ('h':"i there")
```

```
True :: Bool
```

```
? allEqual (\x -> x + 1) (1+) (+1)
```

```
ERROR: ...
```

- Haskell type

```
allEqual :: Eq a => a -> a -> a -> Bool
```

# Type classes — a “middle way”

- Polymorphism restricted using class constraints

```
allEqual :: Eq a => a -> a -> a-> Bool
allEqual x y z = (x == y) && (y == z)
```

Functions for precisely those types  $a$  that belong to the **class**  $Eq$

- A class defines a set of types. E.g.,  $Eq$  is the **equality class**

►  $Int \in Eq$

```
? allEqual 3 (2+1) (1+2)
True :: Bool
```

►  $Int \rightarrow Int \notin Eq$

```
? allEqual (\x -> x + 1) (1+) (+1)
ERROR: a -> a is not an instance of class "Eq"
```

# Definition of the Eq class

- Definition (from Prelude.hs)

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x /= y = not (x==y)
```

- Definition includes

**Class name:** *Eq*

**Signature:** List of function names and types

**Default implementations (optional):** can be overwritten later

- Elements of the class are called **instances**

## Examples of Eq constrained types

- Classes allow restricted form of type generalization

```
allEqual :: Int -> Int -> Int -> Bool
allEqual n m p = (n == m) && (m == p)
```

- Most general type **with class constraint**

```
allEqual :: Eq t => t -> t -> t -> Bool
```

- Element of a list

```
elem :: Eq t => t -> [t] -> Bool
```

```
elem _ []      = False
elem a (x:xs) = (a == x) || elem a xs
```



# Instances

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x==y)
```

- instance builds instances by “interpreting” signature functions

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _      == _    = False
```

- Instances of primitive types like `Int` or `Double` use built-in (primitive) equalities

## Example: visible (and measurable) types

```
class Visible t where
  stringOf :: t -> String
  size     :: t -> Int
```

```
instance Visible Char where
  stringOf ch = [ch]
  size _ = 1
```

```
instance Visible Bool where
  stringOf True  = "Wahr"
  stringOf False = "Falsch"
  size b = 1
```

```
? (stringOf 'e') ++ "ine " ++ (stringOf True) ++ "e Aussage"
"eine Wahre Aussage" :: [Char]
```

## Example (cont.)

- If  $t$  is visible, then a list of type  $[t]$  is also visible

```
instance Visible t => Visible [t] where
  stringOf xs = concat (map stringOf xs)
  size xs     = foldr (+) 0 (map size xs)
```

```
? size [True,False]
2 :: Int
```

```
? stringOf [True,False]
"WahrFalsch" :: [Char]
```

So class **membership** can depend on membership for other types

- Equality over lists

```
instance Eq a => Eq [a] where
  []      == []      = True
  (x:xs) == (y:ys) = x==y && xs==ys
  _      == _      = False
```

## Derived classes

- Classes themselves can also depend on type conditions

```
class Eq a => Ord a where
  (<), (>), (<=), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a
                                -- implement all operators except (<=)

  x < y  = x <= y && x /= y
  x >= y = y <= x
  x > y  = y <= x && x /= y

  max x y | x <= y    = y
           | otherwise = x
  min x y | x <= y    = x
           | otherwise = y
```

- If  $a$  belongs to  $Ord$ , then  $a$  must also belong to  $Eq$
- Functions for  $Eq$  are inherited and some new ones must be given.

```
instance Ord Int where (<=) = primLeInt
```

# Class hierarchies

- Classes can be hierarchically structured

```
class Eq a where ...
```

```
class Eq a => Ord a where ...
```

```
class Ord a => Bounded a where  
  minBound, maxBound :: a
```

```
class (Eq a, Show a) => Num a where  
  (+), (-), (*) :: a -> a -> a ...
```

```
class (Num a, Ord a) => Real a where  
  toRational :: a -> Rational
```

```
class (Real a, Enum a) => Integral a where  
  quot, rem, div, mod :: a -> a -> a ...
```

- Inheritance hierarchies like in OO-programming
- Other similarities, like defaults and overriding

# Comparison: type classes vs. OO programming

## Haskell

```
class Visible t where
  stringOf :: t -> String
  size :: t -> Int
```

```
class Ord t => Bounded t
  where ...
```

```
class Foo t where
  foo :: t -> t
  bar :: t -> Bool
  bar x = True
```

```
instance Visible Int where
  stringOf x = ...
  size x = 0
```

```
sort :: Ord t => [t] -> [t]
sort xs = ...
```

## Java

```
interface Visible {
    String stringOf();
    Integer size(); }

```

```
interface Bounded extends Ord {
}

```

```
abstract class Foo {
    abstract Foo foo();
    Boolean bar() { return True; }
}

```

```
class Integer implements Visible {
    String stringOf() { ... }
    Integer size() { return 0 }
}

```

```
static <T extends Ord>
    List<T> sort(List<T> xs) { ... }

```

## Quick sort (again)

- Which type?

```
sort []      = []  
sort (a:x) = sort [y | y<-x, y<=a] ++ [a] ++ sort [y | y<-x, y>a]
```

- Operations `<=` and `>` require `Ord a => [a] -> [a]`
- *Ord* instances for many Haskell types defined in Haskell Prelude

```
? sort [5,4,7]  
[4, 5, 7] :: [Int]
```

```
? sort ["banana", "apple", "carrot"]  
["apple", "banana", "carrot"] :: [[Char]]
```

```
? sort [True, False, True]  
[False, True, True] :: [Bool]
```

## Example (cont.)

- Parameterization allows further orders (per type)

```
sort' ord [] = []  
sort' ord (a:x) = sort' ord [y | y<-x, ord y a ]  
                ++ [a] ++ sort' ord [y | y<-x, not(ord y a)]
```

```
? sort' (<) [2,5,3]  
[2, 3, 5] :: [Int]
```

```
? sort' (>) [2,5,3]  
[5, 3, 2] :: [Int]
```

```
? sort' (\x y -> x `mod` 10 < y `mod` 10) [21,55,30,8,92,15]  
[30, 21, 92, 55, 15, 8] :: [Int]
```

```
? sort' (\x y -> reverse x < reverse y) ["apple","banana","peach"]  
["banana", "apple", "peach"] :: [[Char]]
```



# Type classes and resolution of overloading

- Execution of (parametric) polymorphic functions is independent of type of arguments
- Classes implement “ad hoc” polymorphism
  - ▶ Operation depends on argument types
- Selection of the actual function:

**During compilation:** if argument types are statically known.

**Run time:** using “look-up” tables. Analogous to method look-up.

## Type classes Show and Read

```
type ShowS = String -> String           -- difference list

class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList  :: [a] -> ShowS

  showsPrec _ x s = show x ++ s           -- instances must override
  show x          = showsPrec 0 x ""      -- at least show or showsPrec
  showList xs s   = ...

? :t show
show :: Show a => a -> String

? show (-1)
"-1"
? showsPrec 10 (-1) " more text"
"(-1) more text"
? show ['H','e','l','l','o']
"\\"Hello\\""
```

## Type classes Show and Read

```

type ShowS = String -> String          -- difference list

class Show a where
  showsPrec :: Int -> a -> ShowS
  show      :: a -> String
  showList  :: [a] -> ShowS

  showsPrec _ x s = show x ++ s        -- instances must override
  show x          = showsPrec 0 x ""    -- at least show or showsPrec
  showList xs s   = ...

? :t show
show :: Show a => a -> String

? show (-1)
"-1"
? showsPrec 10 (-1) " more text"
"(-1) more text"
? show ['H','e','l','l','o']
"\\"Hello\\"

```

**Type class Read** is more complicated.  
Usually suffices to know about read.

```

? :t read
read :: Read a => String -> a

? read "\"Hello\"" :: String
"Hello"

```

## Conclusion: typing in Haskell

- Haskell features a powerful type system
  - ▶ Parametric polymorphic functions
  - ▶ Overloading of functions using type classes
- Type checking is automatic
  - ▶ No proofs, but instead type inference
- Safe type system
  - ▶ prevents runtime errors, e.g.,  $2 + \text{True}$
  - ▶ and offers considerable flexibility, e.g., quick sort