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 :: $(a \to b) \to [a] \to [b]$

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

filter :: $(a \to Bool) \to [a] \to [a]$

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

foldr :: $(a \to b \to b) \to b \to [a] \to b$

```
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[1]

```
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]
```

[1] 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

$$\mathit{foldr}\ (\oplus)\ e\ [l_1, l_2, \ldots, l_n] = l_1 \oplus (l_2 \oplus \cdots \oplus (l_n \oplus e))$$

```
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

$$\mathit{foldl}\ (\oplus)\ e\ [l_1, l_2, \ldots, l_n] = ((e \oplus l_1) \oplus l_2) \oplus \cdots \oplus l_n$$

```
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`

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

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`

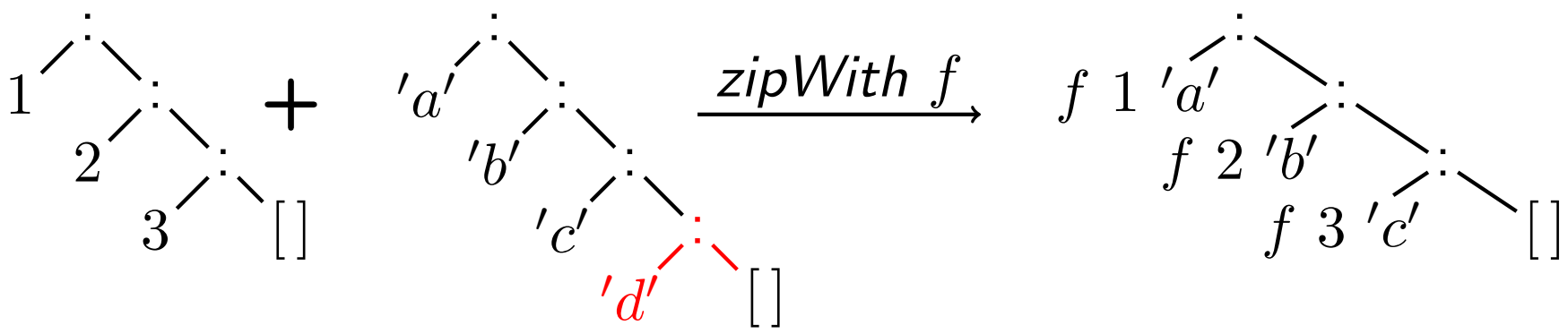
- Combination of `zip` and binary functions is common

```
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

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

  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
```

$$\left(\begin{array}{c|c} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{array}\right) + \left(\begin{array}{c|c} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{array}\right) = \left(\begin{array}{c|c} 8 & 14 \\ 10 & 16 \\ 12 & 18 \end{array}\right)$$

```
?  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))
```

$$\left(\begin{array}{c|c|c|c|c} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{array}\right)$$

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)
```

$$(\ ) \rightarrow (\ )$$

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

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

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

$$\left(\begin{array}{c|c} a_{1,1} \\ a_{2,1} \\ \vdots \\ a_{n,1} \end{array}\right.\left|\begin{array}{c} vs \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ \multicolumn{3}{c|}{\texttt{tr}} & vs \end{array}\right)$$

Example of transposition

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

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

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)
```

- 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.

$$\left(\begin{array}{c|c|c|c} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{array}\right) \cdot \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} a_{1,i} b_i \\ \sum_{i=1}^{m} a_{2,i} b_i \\ \vdots \\ \sum_{i=1}^{m} a_{n,i} b_i \end{pmatrix}$$

$\Rightarrow$ scalar product of A's rows (columns of `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

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

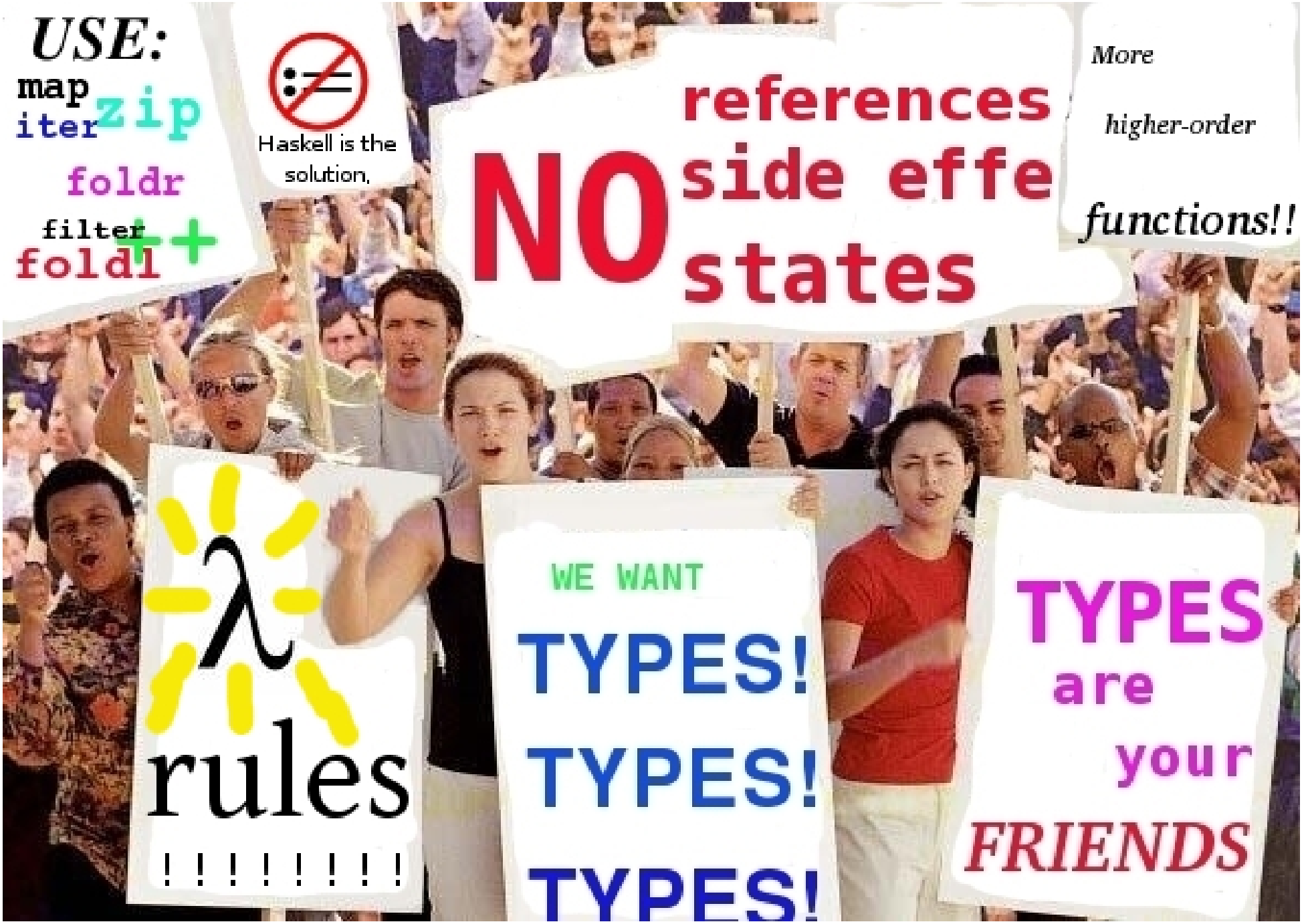
```
matMult :: Matrix -> Matrix -> Matrix
matMult a b = map (vecMult a) b

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

Exercise: Scalar multiplicaton 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


USE:
map
iter
zip
foldr
filter
++
foldl
Haskell is the solution.
NO
references
side effe
states
More
higher-order
functions!!
λ
rules
!!!!!!!!
WE WANT
TYPES!
TYPES!
TYPES!
TYPES
are
your
FRIENDS

Typing

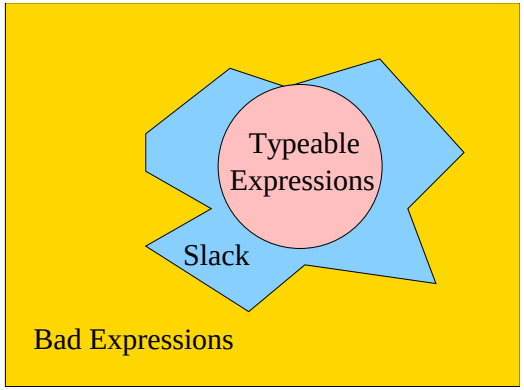
Type checking: an overview

- Type checking should prevent "dangerous expressions"

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

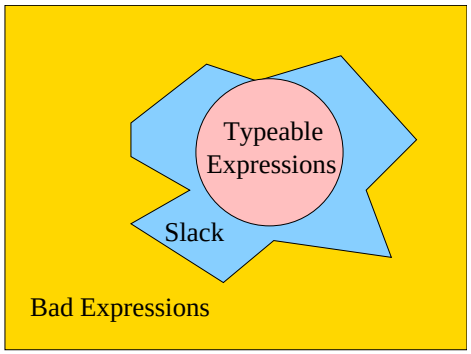
- **Dangerous expressions** $\Longrightarrow$ runtime error

- 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

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

- 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)

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

- 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. $(\textbf{iszero}\,(\lambda x.\, x))$

Mini-Haskell — comments

- Core is λ-calculus: variables, abstraction, and application

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

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

`&&`, `||`, Strings, . . .

- We employ syntactic sugar, like omitting parenthesis

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

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

Typing

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

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

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

$$x : Int \vdash x + 2 :: Int \qquad x : Int, f : Bool \to Bool \nvdash f\ x :: Bool$$

Typing — proof system

- Proof rules formulated in terms of type judgements J

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

- Example axiom ($n \in \{\dots, -1, 0, 1, \dots\}$):

$$\frac{}{\Gamma \vdash n :: Int}\ \mathit{Int}$$

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

$$\frac{\Gamma \vdash t_1 :: Int \qquad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1\ \mathbf{op}\ t_2) :: Int}\ \mathit{BinOp}$$

- Proofs built from rules and axioms

$$\dfrac{\dfrac{}{x : Int \vdash 2 :: Int}\ \mathit{Int} \qquad \dfrac{\vdots}{x : Int \vdash (x + 1) :: Int}\ \mathit{BinOp}}{x : Int \vdash (2 + (x + 1)) :: Int}\ \mathit{BinOp}$$

Rules for core λ-calculus

Axiom:

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \mathit{Var}$$

Abstraction $(x \notin \Gamma)$:

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

Application:

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1\ t_2) :: \tau}\ \mathit{App}$$

Examples:

$$\frac{\dfrac{}{x : a \vdash x :: a}\ \mathit{Var}}{\vdash (\lambda x.\, x) :: a \to a}\ \mathit{Abs}$$

$$\frac{\dfrac{\dfrac{}{x : a, y : b \vdash x :: a}\ \mathit{Var}}{x : a \vdash \lambda y.\, x :: b \to a}\ \mathit{Abs}}{\vdash \lambda x.\, \lambda y.\, x :: a \to b \to a}\ \mathit{Abs}$$

A larger example

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \mathit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x.\, t) :: \sigma \to \tau}\ \mathit{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1\ t_2) :: \tau}\ \mathit{App}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x :: a \to (b \to c)}\ \mathit{Var} \qquad \dfrac{}{\Gamma \vdash z :: a}\ \mathit{Var}}{\Gamma \vdash x\ z :: b \to c}\ \mathit{App} \qquad \dfrac{\dfrac{}{\Gamma \vdash y :: a \to b}\ \mathit{Var} \qquad \dfrac{}{\Gamma \vdash z :: a}\ \mathit{Var}}{\Gamma \vdash y\ z :: b}\ \mathit{App}}{\dfrac{\dfrac{\overbrace{x : a \to (b \to c), y : a \to b, z : a}^{\Gamma=} \vdash (x\ z)\ (y\ z) :: c}{x : a \to (b \to c), y : a \to b \vdash \lambda z.\, (x\ z)\ (y\ z) :: a \to c}\ \mathit{Abs}}{x : a \to (b \to c) \vdash \lambda y.\, \lambda z.\, (x\ z)\ (y\ z) :: (a \to b) \to (a \to c)}\ \mathit{Abs}}{\vdash \lambda x.\, \lambda y.\, \lambda z.\, (x\ z)\ (y\ z) :: (a \to (b \to c)) \to (a \to b) \to (a \to c)}\ \mathit{Abs}$$

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

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 $\dfrac{}{\Gamma \vdash n :: Int}\ \mathit{Int}$ $\dfrac{}{\Gamma \vdash True :: Bool}\ \mathit{True}$ $\dfrac{}{\Gamma \vdash False :: Bool}\ \mathit{False}$
- Operations (**op** $\in \{+, *\}$):

$$\frac{\Gamma \vdash t :: Int}{\Gamma \vdash (\textbf{iszero}\ t) :: Bool}\ \mathit{iszero} \qquad \frac{\Gamma \vdash t_1 :: Int \qquad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: Int}\ \mathit{BinOp}$$

$$\frac{\Gamma \vdash t_0 :: Bool \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash (\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) :: \tau}\ \mathit{if}$$

- Tuples

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

- Example

$$\frac{\dfrac{\dfrac{}{x : Int \vdash x :: Int}\ \mathit{Var} \qquad \dfrac{}{x : Int \vdash 2 :: Int}\ \mathit{Int}}{x : Int \vdash x + 2 :: Int}\ \mathit{BinOp}}{\vdash \lambda x.\, x + 2 :: Int \to Int}\ \mathit{Abs}$$

Type inference

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash (\lambda x.\, t) :: \sigma \to \tau}\ \textit{Abs} \qquad \frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash (t_1\ t_2) :: \tau}\ \textit{App}$$

- Syntax-directed typing rules specify algorithm for computing type
  1. Start with judgement $\vdash t :: \tau_0$ with type variable τ_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:

$$\frac{\dfrac{\dfrac{}{z : \tau_1 \vdash z :: \tau_0}\ \textit{Var}}{\vdash \lambda z.\, z :: \tau_1 \to \tau_0}\ \textit{Abs} \qquad \dfrac{\dfrac{}{x : \tau_2 \vdash x :: \tau_3}\ \textit{Var}}{\vdash \lambda x.\, x :: \tau_1}\ \textit{Abs}}{\vdash (\lambda z.\, z)\ (\lambda x.\, x) :: \tau_0}\ \textit{App}$$

$$\tau_1 = \tau_0$$
$$\tau_1 = \tau_2 \to \tau_3$$
$$\tau_2 = \tau_3$$

I.e., $\tau_0 = \tau_3 \to \tau_3$

E.g., $\tau_0 = a \to a$ and $\tau_0 = Int \to Int$ are correct types.

Type inference example

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x :: ((\tau_3 \to Int), \tau_4)}\,Var}{\Gamma \vdash \textbf{fst}\ x :: \tau_3 \to Int}\,fst \qquad \dfrac{\dfrac{}{\Gamma \vdash 2 :: \tau_5}\,Int \qquad \dfrac{}{\Gamma \vdash True :: \tau_6}\,True}{\Gamma \vdash (2, True) :: \tau_3}\,Tuple}{\Gamma \vdash (\textbf{fst}\ x)\ (2, True) :: Int}\,App}{\dfrac{\overbrace{x : \tau_1}^{\Gamma=} \vdash \textbf{iszero}\ ((\textbf{fst}\ x)\ (2, True)) :: \tau_2}{\vdash \lambda x.\, \textbf{iszero}\ ((\textbf{fst}\ x)\ (2, True)) :: \tau_0}\,Abs}\,App$$

Constraints:

$\tau_0 = \tau_1 \to \tau_2$

$\tau_2 = Bool$

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

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

$\tau_5 = Int$

$\tau_6 = Bool$

Most general type:

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

Exercise:

Infer the type of

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

When type inference fails

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

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \tau_1 \vdash x :: (\tau_3, \tau_5)}\,\mathit{Var}}{x : \tau_1 \vdash \mathbf{fst}\ x :: \tau_3}\,\mathit{fst} \qquad \dfrac{\dfrac{}{x : \tau_1 \vdash x :: Int}\,\mathit{Var}}{x : \tau_1 \vdash \mathbf{iszero}\ x :: \tau_4}\,\mathit{iszero}}{x : \tau_1 \vdash (\mathbf{fst}\ x, \mathbf{iszero}\ x) :: \tau_2}\,\mathit{Tuple}}{\vdash \lambda x.\, (\mathbf{fst}\ x, \mathbf{iszero}\ x) :: \tau_0}\,\mathit{Abs}$$

- Constraints:

$\tau_0 = \tau_1 \to \tau_2$
$\tau_2 = (\tau_3, \tau_4)$
$\tau_1 = (\tau_3, \tau_5)$
$\tau_4 = Bool$
$\tau_1 = Int$

(τ_3, τ_5) and 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.

$$\dfrac{\dfrac{}{f : \tau_1 \vdash f :: \tau_3 \to \tau_2}\ \textit{Var} \qquad \dfrac{}{f : \tau_1 \vdash f :: \tau_3}\ \textit{Var}}{\dfrac{f : \tau_1 \vdash f\ f :: \tau_2}{\vdash \lambda f.\ f\ f :: \tau_0}\ \textit{Abs}}\ \textit{App}$$

$$\tau_0 = \tau_1 \to \tau_2$$
$$\tau_1 = \tau_3 \to \tau_2$$
$$\tau_1 = \tau_3$$

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

  So, there is no solution to the constraints.

- Compare with natural numbers:
  $x = x + 1$ has only solution ∞, 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 "$\to$" corresponds to propositional logic connective "$\to$"
  - Atomic types correspond to propositional variables
- Rules correspond to those for (minimal) propositional logic

$$\frac{}{\ldots, \tau, \ldots \vdash \tau}\ \textit{axiom} \qquad \frac{\Gamma, \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}\ \to\text{-}I \qquad \frac{\Gamma \vdash \sigma \to \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\ \to\text{-}E$$

- Example

$$\dfrac{\dfrac{\dfrac{}{\tau, \sigma \vdash \tau}\ \textit{axiom}}{\tau \vdash \sigma \to \tau}\ \to\text{-}I}{\vdash \tau \to \sigma \to \tau}\ \to\text{-}I$$

- Correspondence actually quite deep

Summary of Mini-Haskell Typing Rules

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

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \textit{App} \qquad \frac{\Gamma \vdash t :: Int}{\Gamma \vdash \textbf{iszero}\ t :: Bool}\ \textit{iszero}$$

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

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

$$\frac{\Gamma \vdash t_0 :: Bool \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 :: \tau}\ \textit{if}$$

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

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
```

  - $Int \to Int \notin Eq$

```
? 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 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., $2 + True$
  - and offers considerable flexibility, e.g., quick sort