Higher-order Programming and Types

David Basin

Department of Computer Science
ETH Zurich

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

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

```
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

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

```
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 matrices and vectors

- Vectors and matrices over `Int`

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

- Vector addition

```
vecAdd :: Vector -> Vector -> Vector
vecAdd v1 v2 = map (uncurry (+)) (zip v1 v2)

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

- 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 _      _      = []

vecAdd :: Vector -> Vector -> Vector
vecAdd = 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

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

- Addition of matrices

```
matAdd :: Matrix -> Matrix -> Matrix
matAdd = zipWith vecAdd

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

Transposing a matrix

- A list of columns is converted to a list of rows

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

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

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

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

Example of transposition

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

Scalar (dot) product of two vectors

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

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

? skProd [1,2,3] [4,5,6]
32 :: Int
```

Matrix multiplication

- We first multiply an $n \times m$ matrix with an $m \times 1$ column vector

```
vecMult :: Matrix -> Vector -> Vector
vecMult m v = map (`skProd` v) (tr m)

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

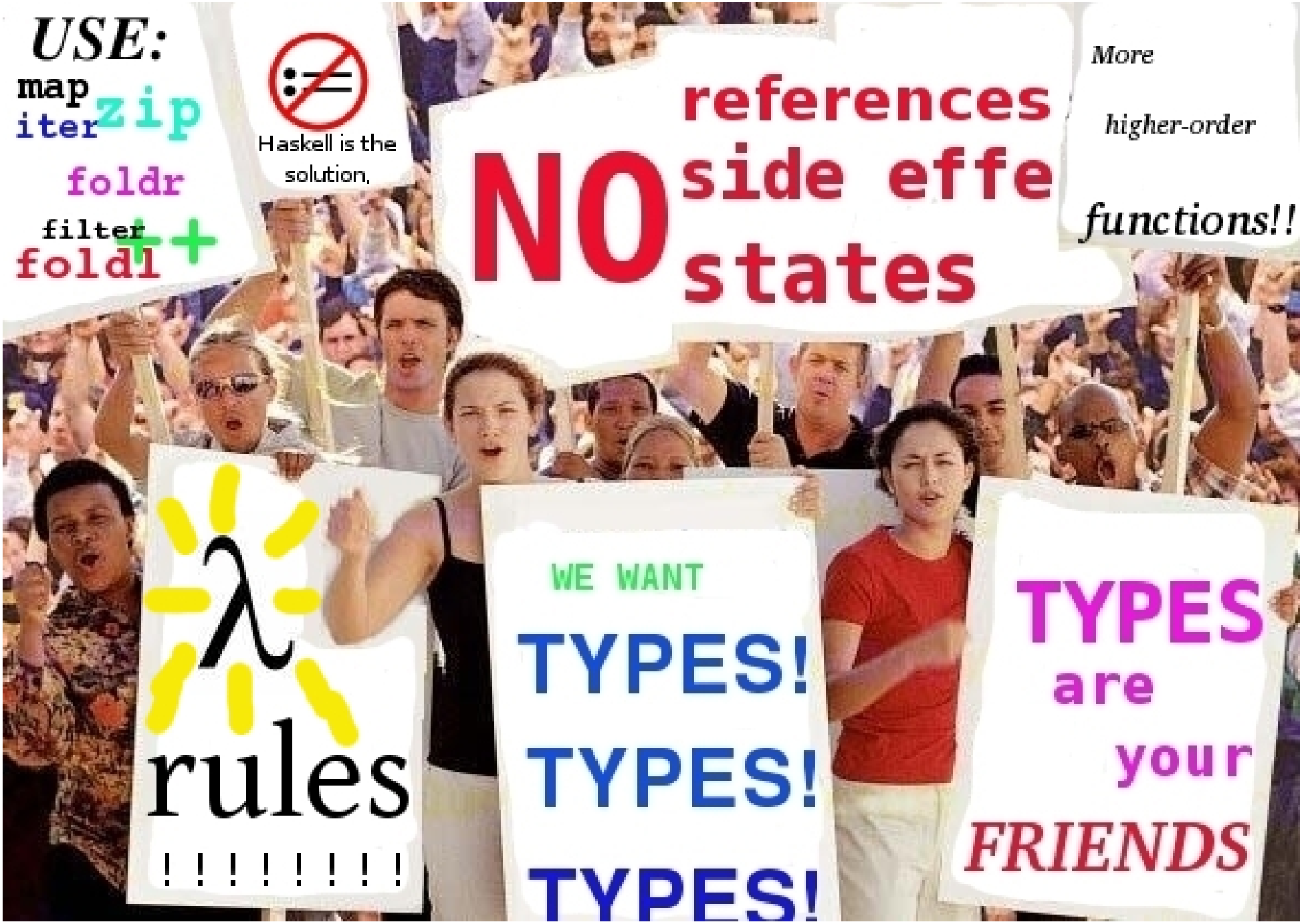
- Matrix multiplication iterates this operation over an $m \times k$ matrix

```
matMult :: Matrix -> Matrix -> Matrix
matMult m1 m2 = map (vecMult m1) m2

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

Conclusion

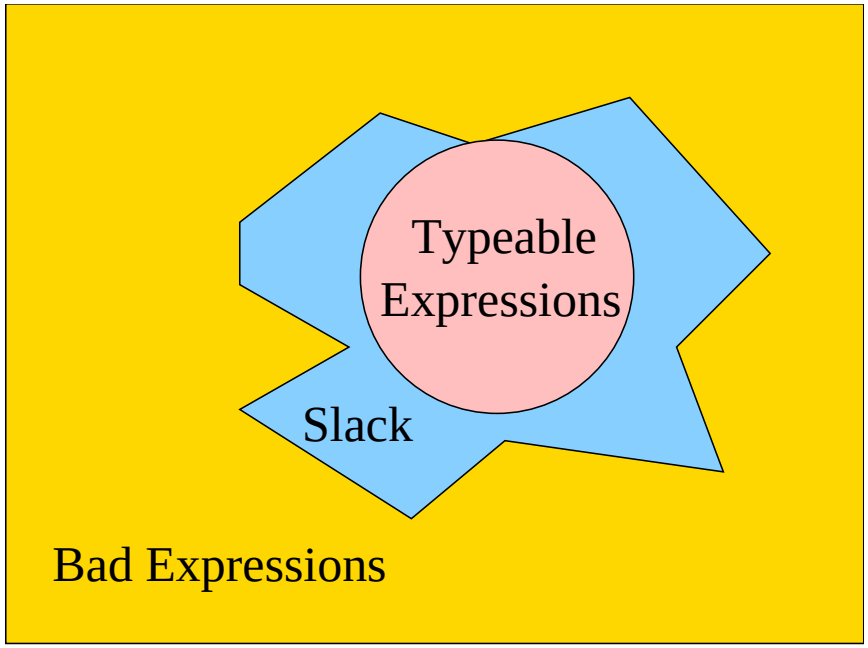
- First-order programming
  - Programming with elements of base types, like True or 13
  - Close to machine architecture
- Higher-order programming
  - Functions are first-class objects
- Increases abstraction and ways of constructing programs
- Other advantages like reusability and rapid prototyping


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

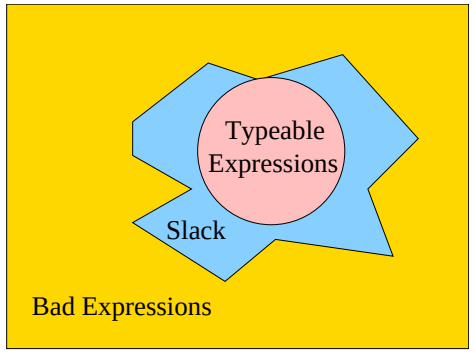
Typing

Type checking: an overview

- Type checking should prevent "dangerous expressions", e.g., $2 + True$
- **Dangerous expressions** $\Longrightarrow$ runtime error
- Undecidable problem!



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

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

- Typing is a very rich topic (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)

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

- 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 will also 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: $A \vdash t :: \tau$
  - A is a set of bindings $x_i : \tau_i$, mapping variables to types. Intuitively A represents a kind of typing "symbol table".
  - t is a term
  - τ is a type
- Intuition: given symbol table A, then 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

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

- Example axiom ($n \in \{\ldots, -1, 0, 1, \ldots\}$): $A \vdash n :: Int$
- Example rule (**op** $\in \{+, *\}$):

$$\frac{A \vdash t_1 :: Int \quad A \vdash t_2 :: Int}{A \vdash (t_1 \ \textbf{op} \ t_2) :: Int}$$

- Proofs built from rules and axioms

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

Rules for core λ-calculus

- **Axiom**: $\ldots, x : \tau, \ldots \vdash x :: \tau$
- **Abstraction** $(x \notin A)$: $\dfrac{A, x : \sigma \vdash t :: \tau}{A \vdash (\lambda x.\, t) :: \sigma \to \tau}$
- **Application**: $\dfrac{A \vdash t_1 :: \sigma \to \tau \quad A \vdash t_2 :: \sigma}{A \vdash (t_1\, t_2) :: \tau}$
- Examples:

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

- Exercise:

$$\vdash \lambda x.\, \lambda y.\, \lambda z.\, (xz)(yz) :: (a \to (b \to c)) \to (a \to b) \to (a \to c)$$

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
```

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

$$\ldots, \tau, \ldots \vdash \tau \qquad \frac{A, \sigma \vdash \tau}{A \vdash \sigma \Rightarrow \tau} \qquad \frac{A \vdash \sigma \Rightarrow \tau \quad A \vdash \sigma}{A \vdash \tau}$$

- Example

$$\frac{\dfrac{\tau, \sigma \vdash \tau}{\tau \vdash \sigma \Rightarrow \tau}}{\vdash \tau \Rightarrow \sigma \Rightarrow \tau}$$

- Correspondence actually quite deep

Further typing rules for mini-Haskell

- Base types

$$A \vdash n :: Int \qquad A \vdash True :: Bool \qquad A \vdash False :: Bool$$

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

$$\frac{A \vdash t :: Int}{A \vdash (\textbf{iszero}\, t) :: Bool} \qquad \frac{A \vdash t_1 :: Int \quad A \vdash t_2 :: Int}{A \vdash (t_1\ \textbf{op}\ t_2) :: Int} \qquad \frac{A \vdash t_0 :: Bool \quad A \vdash t_1 :: \tau \quad A \vdash t_2 :: \tau}{A \vdash (\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) :: \tau}$$

- Tuples

$$\frac{A \vdash t_1 :: \tau_1 \quad A \vdash t_2 :: \tau_2}{A \vdash (t_1, t_2) :: (\tau_1, \tau_2)} \qquad \frac{A \vdash t :: (\tau_1, \tau_2)}{A \vdash (\textbf{fst}\, t) :: \tau_1} \qquad \frac{A \vdash t :: (\tau_1, \tau_2)}{A \vdash (\textbf{snd}\, t) :: \tau_2}$$

Example

$$\frac{\dfrac{x : Int \vdash x :: Int \quad x : Int \vdash 2 :: Int}{x : Int \vdash x + 2 :: Int}}{\vdash \lambda x.\, x + 2 :: Int \to Int}$$

Examples (cont.)

- A larger example

$$\dfrac{\dfrac{x : Int \vdash x :: Int \quad x : Int \vdash 2 :: Int}{\dfrac{x : Int \vdash x + 2 :: Int}{\vdash \lambda x.\, x + 2 :: Int \to Int}} \quad \dfrac{\dfrac{\vdash 2 :: Int \quad \vdash True :: Bool}{\vdash (2, True) :: (Int, Bool)}}{\vdash \mathbf{fst}\,(2, True) :: Int}}{\vdash (\lambda x.\, x + 2)\,(\mathbf{fst}\,(2, True)) :: Int}$$

- Examples in ghc

```
? :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
```

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

- Definition includes

  Class name: Eq
  Signature: List of function names and types
  (Optional Standard-)Definitions: 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

```
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 `Float` use built-in (primitive) equalities

Example: visible (and measurable) types

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

instance Visible Char where
  toString ch = [ch]
  size _ = 1

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

? (toString 'e') ++ "ine " ++ (toString 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
  toString xs = concat (map toString xs)
  size xs     = foldr (+) 0 (map size xs)

? size [True,False]
2 :: Int

? toString [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

  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

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.

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
- Secure type system
  - prevents runtime errors, e.g., $2 + True$
  - and offers considerable flexibility, e.g., quick sort