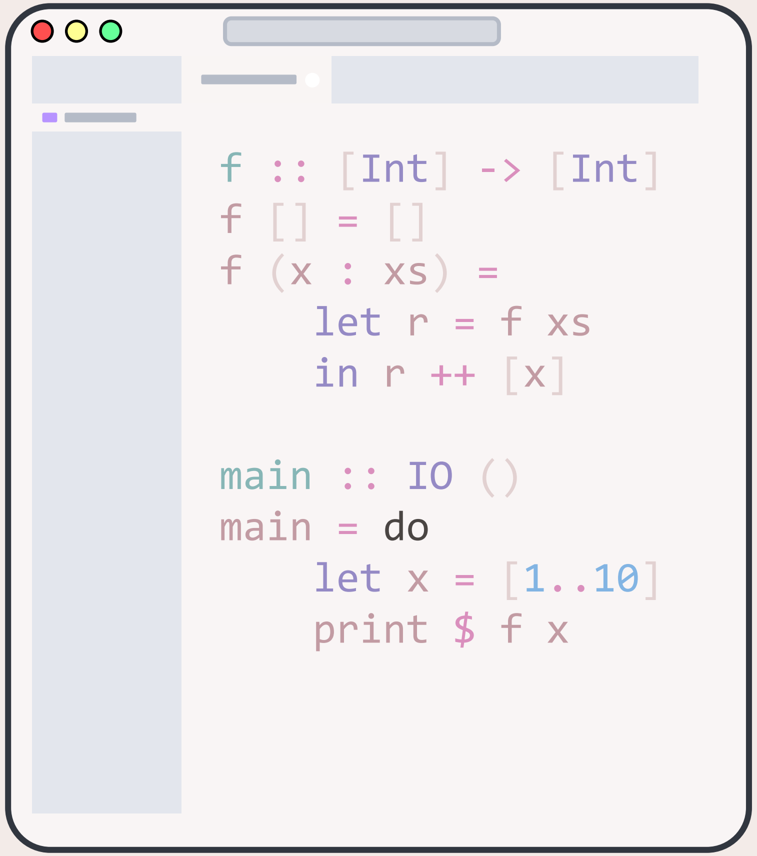
IT5100A

Industry Readiness:
Typed Functional Programming

Typeclasses

Foo Yong Qi
yongqi@nus.edu.sg



Ad-Hoc Polymorphism

Algebraic Data Types don't give much information about its component fields (different constructors usually have different fields)

```haskell
data Shape = Circle Double
           | Rectangle Double Double
```

Most importantly, we write **functions** over these types

```haskell
area :: Shape -> Double
area (Circle r) = pi * r ^ 2
area (Rectangle w h) = w * h
```

Those functions might/should also work on other types

```haskell
data House = H [Room]
type Room = Rectangle
area' :: House -> Double
area' (H ls) = foldr ((+) . area) 0 ls
```

How do we allow these types to have the **same functionality** but **type-dependent implementations**?

Python approach: if it walks like a duck, quacks like a duck, it is probably a duck

```python
@dataclass
class Rectangle:
    w: float
    h: float
    def area(self) -> float:
        return self.w * self.h
@dataclass
class House:
    ls: list[Rectangle]
    def area(self) -> float:
        return sum(x.area() for x in self.ls)

def total_area(ls):
    return sum(x.area() for x in ls)
ls = [Rectangle(1, 2), House([Rectangle(3, 4)])]
total_area(ls) # 14
```

Two classes declaring the same method with different implementations is known as **method overloading**, a form of **ad-hoc polymorphism**!

```python
@dataclass
class Fraction:
    num: int
    den: int
    def __add__(self, f: 'Fraction') -> 'Fraction':
        num = self.num * f.den + f.num * self.den
        den = self.den * f.den
        return Fraction(num, den)
print(1 + 2) # 3
print(Fraction(1, 2) + Fraction(3, 4)) # 10/8
```

Python and other languages like C++ support **operator overloading**

Add static duck typing on the fly with protocols in Python

```python
class HasArea(Protocol):
    @abstractmethod
    def area(self) -> float:
        pass

def total_area(ls: list[HasArea]) -> float:
    return sum(x.area() for x in ls)
ls: list[HasArea] = [Rectangle(1, 2), House([Rectangle(3, 4)])]
total_area(ls) # 14
```

Alternatively, define abstract class / interface and let classes inherit them

Expression problem: adding new methods to classes is difficult in OOP

Use helpers!

```python
def rectangle_area(rect: Rectangle) -> float:
    return rect.w * rect.h
def house_area(house: House) -> float:
    return sum(x.area() for x in house.ls)

def area(x, f) -> float:
    return f(x)

r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r, rectangle_area) # 2
area(h, house_area) # 12
```

Idea: put implementing functions in dictionary, lookup by type!

```python
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

Idea: put implementing functions in dictionary, lookup by type!

Dictionary where keys are types and values are implementations for `area`

```python
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

Idea: put implementing functions in dictionary, lookup by type!

Implementations of `area` for `Rectangle`s and `House`s

```
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

Idea: put implementing functions in dictionary, lookup by type!

`area` looks up implementation by type

```python
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

Idea: put implementing functions in dictionary, lookup by type!

```python
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
r = Rectangle(1, 2)
h = House([Rectangle(3, 4)])
area(r) # 2
area(h) # 12
```

`area` works on `Rectangle`s and `House`s without changing class definitions

Defining a new class/new overloaded function is a simple extension

```python
@dataclass
class Triangle(Shape):
    w: float
    h: float
HasArea[Triangle] = lambda t: t.w * t.h
area(Triangle(5, 2)) # 5
```

This form of ad-hoc polymorphism supports:

- Otherwise disparate types adhering to a common interface
- Decoupling types and behaviour

How do we do this in Haskell?

Typeclasses

Typeclass

A type system construct that enables **ad-hoc polymorphism**

Typeclass

A **nominal classification** of types that **support specified behaviour** by providing its **type-specific implementation**

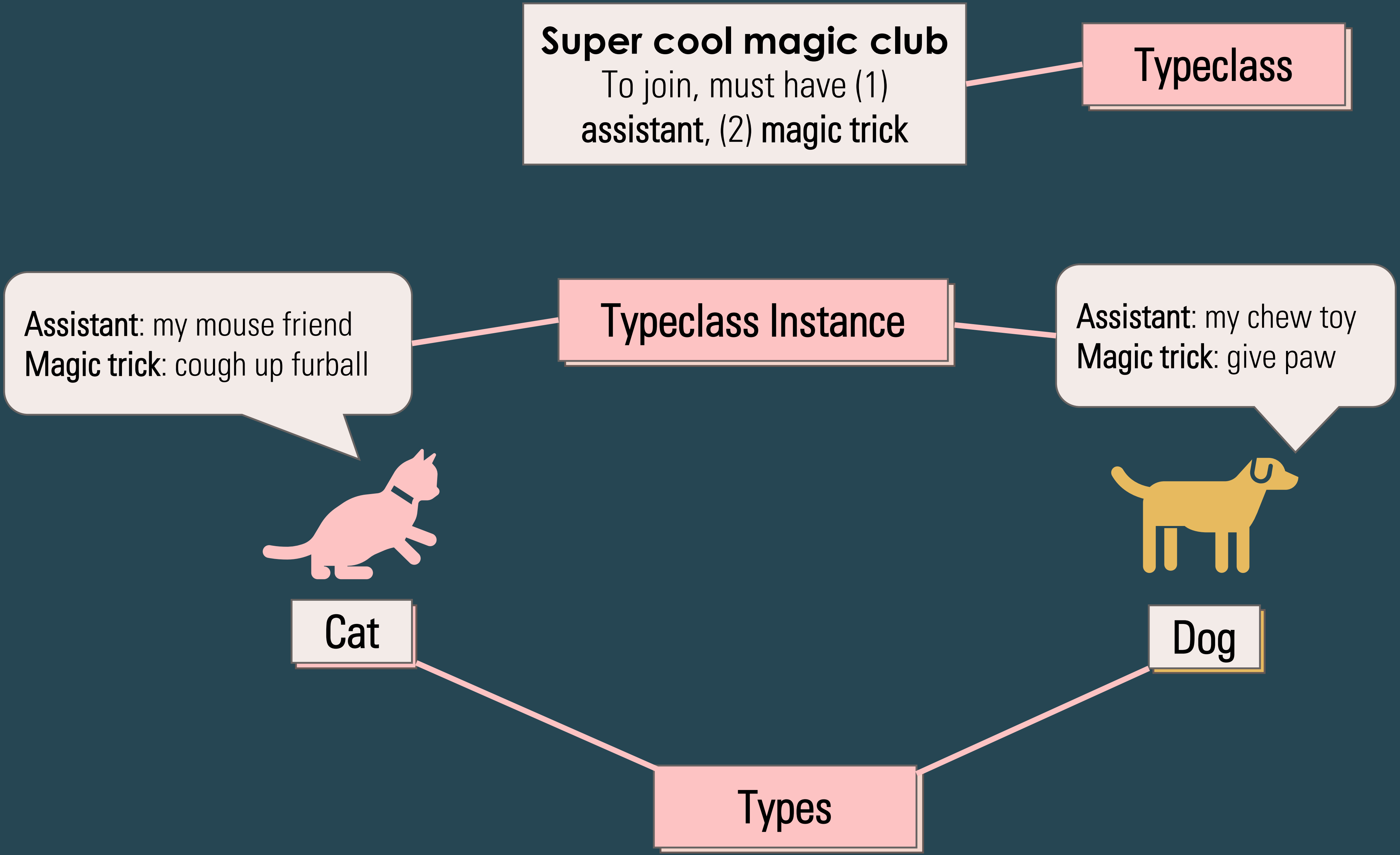
Typeclass

A **constraint** or **witness** for a type to **support specified behaviours**

Typeclass System

Typeclass
Gives interface/specification/contract for members of typeclass to follow

Typeclass Instance
Provides actual type-specific implementations of the functions specified in the typeclass


Super cool magic club
To join, must have (1) assistant, (2) magic trick
Typeclass
Assistant: my mouse friend
Magic trick: cough up furball
Typeclass Instance
Assistant: my chew toy
Magic trick: give paw
Cat
Dog
Types

Example of different types that have an area

```haskell
data Shape = Circle Double
           | Rectangle Double Double
           | Triangle Double Double

data House = H [Room]

data Room  = R { roomName :: String, shape :: Shape }
```

```haskell
data Shape = Circle Double
           | Rectangle Double Double
           | Triangle Double Double

data House = H [Room]

data Room  = R { roomName :: String, shape :: Shape }
```

Define contract for all types that have area

```haskell
class HasArea a where
    area :: a -> Double
```

Why is `HasArea` polymorphic?

Recall: we are looking up implementation by type

```
HasArea = {}
HasArea[Rectangle] = lambda rect: rect.w * rect.h
HasArea[House] = lambda house: sum(x.area() for x in house.ls)
def area(x):
    t = type(x)
    return HasArea[t](x)
```

Dictionary receives type and returns new function…

Polymorphic!

```haskell
class HasArea a where
    area :: a -> Double
```

Now we provide type-specific implementations of `area`

```haskell
instance HasArea Shape where
  area (Circle r) = pi * r ^ 2
  area (Rectangle w h) = w * h
  area (Triangle w h) = w * h / 2
instance HasArea Room where
  area x = area $ shape x
instance HasArea House where
  area (H rooms) = sum $ map area rooms
```

The `area` function now works on all those types!

```haskell
x :: Shape = Triangle 2 3
y :: Room = R "bedroom" (Rectangle 3 4)
z :: House = H [y]
ax = area x -- 3
ay = area y -- 12
az = area z -- 12
```

```haskell
x :: Shape = Triangle 2 3
y :: Room = R "bedroom" (Rectangle 3 4)
z :: House = H [y]
ax = area x -- 3
ay = area y -- 12
az = area z -- 12
```

```
ghci> :t area
area :: forall a. HasArea a => a -> double
```

Read: **`area`** is a function for all **`a`** where **`a`** is constrained by **`HasArea`**, and receives an **`a`**, and returns a **`Double`**

Another way of looking at this: `HasArea` is an ADT

```haskell
data HasArea a = HA { area :: a -> Double }
```

Another way of looking at this: `HasArea` is an ADT

```
data HasArea a = HA { area :: a -> Double }
```

Typeclass instances are just normal terms

```
hasAreaShape :: HasArea Shape
hasAreaShape = HA $ \x -> case x of
        Circle r -> pi * r ^ 2
        Rectangle w h -> w * h
        Triangle w h -> w * h / 2
```

Another way of looking at this: `HasArea` is an ADT

```haskell
data HasArea a = HA { area :: a -> Double }
```

Typeclass instances are just normal terms

```haskell
hasAreaShape :: HasArea Shape
hasAreaShape = HA $ \x -> case x of
        Circle r -> pi * r ^ 2
        Rectangle w h -> w * h
        Triangle w h -> w * h / 2
```

To compute the area of something that has an area, call the **`area`** function passing in the typeclass instance

```haskell
x :: Shape = Triangle 2 3
ax = area hasAreaShape x -- 3
```

Typeclass system

```
class HasArea a where
    area :: a -> Double
```

```
x :: Shape = Triangle 2 3
ax = area x -- 3
```

```
ghci> :t area
area :: forall a. HasArea a => a -> double
```

"Helper system"

```
data HasArea a = HA { area :: a -> Double }
```

```
x :: Shape = Triangle 2 3
ax = area hasAreaShape x -- 3
```

```
ghci> :t area
area :: forall a. HasArea a -> a -> double
```

Typeclass system

"Helper system"

Our system is almost identical to how Haskell implements typeclasses—in actual typeclass system, we let Haskell **infer the supporting term (term inference)**

```haskell
class HasArea a where
    area :: a -> Double
```

```haskell
x :: Shape = Triangle 2 3
ax = area x -- 3
```

```
ghci> :t area
area :: forall a. HasArea a => a -> double
```

```haskell
data HasArea a = HA { area :: a -> Double }
```

```haskell
x :: Shape = Triangle 2 3
ax = area hasAreaShape x -- 3
```

```
ghci> :t area
area :: forall a. HasArea a -> a -> double
```

Polymorphism & Typeclasses

Let's define a function that uses our new **area** function

```
totalArea :: [Shape] -> Double
totalArea [] = 0
totalArea (x : xs) = area x + totalArea xs
-- Point free style:
totalArea = sum . map area
```

Define another one over a list of **Rooms**

```
totalArea :: [Room] -> Double
totalArea = sum . map area
```

Polymorphism & Typeclasses

Same implementation, different types—make it **polymorphic**!

```haskell
totalArea :: [Shape] -> Double
totalArea [] = 0
totalArea (x : xs) = area x + totalArea xs
-- Point free style:
totalArea = sum . map area
```

```haskell
totalArea :: [Room] -> Double
totalArea = sum . map area
```

Don't forget to constrain **a** to have an instance of **HasArea**!

```haskell
totalArea :: forall a. HasArea a => [a] -> Double
totalArea = sum . map area
```

Now, type of list elements doesn't matter as long as they are members of **HasArea** typeclass!

```haskell
ghci> xs :: [Shape] = [Rectangle 1 2, Triangle 3 4]
ghci> ys :: [House] = [H [R "bedroom" (Rectangle 1 2)]]
ghci> axz = totalArea xs -- 8
ghci> ays = totalArea ys -- 2
```

Commonly-Used Typeclasses

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

```haskell
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Example:

```
instance Eq Fraction where
  (F a b) == (F c d) = a == c && b == d
  (F a b) /= (F c d) = a /= c || b /= d
```

Notice that usually by definition `a /= b = not (a == b)`, having to define both `==` and `/=` is cumbersome

Let's inspect the definition of Eq

```
ghci> :i Eq
type Eq :: * -> Constraint
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
    -- Defined in 'GHC.Classes'
```

Notice the `MINIMAL` pragma: we only need to define either (==) or (/=) for a complete definition

Let’s try only defining (==)

```haskell
data Fraction = F Int Int
instance Eq Fraction where
  (F a b) == (F c d) = a == c && b == d

x, y :: Fraction
x = F 1 2
y = F 3 4

xey, xney :: Bool
xey = x == y -- False
xney = x /= y -- True
```

Everything works!

Another example:

```haskell
data Tree a = Empty
            | Node (Tree a) a (Tree a)

instance Eq a => Eq (Tree a) where
    Empty == Empty = True

    (Node l1 c1 r1) == (Node l2 c2 r2) =
        l1 == l2 && c1 == c2 && r1 == r2

    x == y = False
```

Equality of trees depends on equality of their elements!

Some classes also require instances of another typeclass, e.g. `Ord`

```
class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  -- etc..
```

Cannot define `Ord` instance for type without accompanying `Eq` instance

Defining `Eq` instances simple but tedious—derive automatically!

```haskell
data Tree a = Empty
            | Node (Tree a) a (Tree a)
    deriving Eq

x = Empty /= Node Empty 1 Empty -- True
```

Recall: **`map`** maps function over items in list

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x : xs) = f x : map f xs
```

Can we try mapping **`Tree`**s?

```haskell
map' :: (a -> b) -> Tree a -> Tree b
map' _ Empty = Empty
map' f (Node l c r) = Node (map' f l) (f c) (map' f r)

x = map' (+1) (Node Empty 1 Empty) -- Node Empty 2 Empty
```

Clearly possible! Let's look at their type signatures

```haskell
map  :: (a -> b) -> [a]    -> [b]
map' :: (a -> b) -> Tree a -> Tree b
```

```haskell
map  :: (a -> b) -> [a]    -> [b]
map' :: (a -> b) -> Tree a -> Tree b
```

Any mappable **type constructor** `f` can perform `fmap` like so:

```haskell
class Mappable f where
  fmap :: (a -> b) -> f a -> f b
instance Mappable [] where
  fmap = map
instance Mappable Tree where
  fmap = map'
```

This is actually the `Functor` typeclass:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
instance Functor [] where
  fmap = map
instance Functor Tree where
  fmap = map'
```

`Functor` is an example of a higher-kinded type

Functional Dependencies

Let's look at the type of **(+)**:

```
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

This is very different from how **+** behaves in Python:

```
>>> type(1 + 1)
class <'int'>
>>> type(1 + 1.0)
class <'float'>
>>> type(1.0 + 1)
class <'float'>
>>> type(1.0 + 1.0)
class <'float'>
```

Create another typeclass defining heterogenous addition `(+#)`:

```
class (Num a, Num b, Num c) => HAdd a b c where
  (+#) :: a -> b -> c
```

Typeclass instances for `Int` and `Double`:

```
instance Num a => HAdd a a a where
  (+#) :: a -> a -> a
  (+#) = (+)

instance HAdd Int Double Double where
  (+#) :: Int -> Double -> Double
  x +# y = fromIntegral x + y

instance HAdd Double Int Double where
  (+#) :: Double -> Int -> Double
  x +# y = x + fromIntegral y
```

```
ghci> x :: Int = 1
ghci> y :: Double = 2.0
ghci> x +# y
<interactive>:3:1: error:
    - No instance for (HAdd Int Double ()) arising from a use of 'it'
    - In the first argument of 'print', namely 'it'
      In a stmt of an interactive GHCi command: print it
ghci> x +# y :: Double
3.0
```

Compiler does not know what the return type should be! Nothing stopping us from having two instances like so:

```
instance HAdd Int Double Double where
  (+#) :: Int -> Double -> Double
  -- ...
instance HAdd Int Double String where
  (+#) :: Int -> Double -> String
  -- ...
```

```
(+#) :: a -> b -> c
```

c depends solely on what **a** and **b** are
a and **b** should **uniquely characterize c**

State this as a **functional dependency**

```haskell
{-# LANGUAGE FunctionalDependencies #-}
class (Num a, Num b, Num c)
      => HAdd a b c | a b -> c where
  (+#) :: a -> b -> c
```

```
ghci> x :: Int = 1
ghci> y :: Double = 2.0
ghci> x +# y
3.0
```

Existential Typeclass "Antipattern"

Python: if class abides by protocol, can be put in a list of that protocol, okay because protocol is a class (therefore a type)

```python
class HasArea(Protocol):
    # ...
# following is ok and well-typed
ls: list[HasArea] = [Rectangle(1, 2), House([...])]
```

Not okay in Haskell since **`HasArea`** is not a type

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]
ls = [x, y, z] -- error!
```

How do we create a type that represents all types that implement `HasArea` in Haskell?

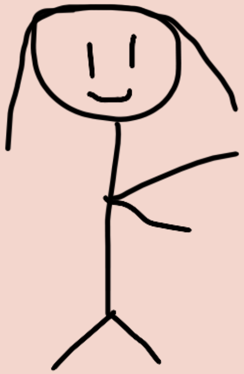
Polymorphic types are known as for-all types (implementation is independent of input)

$$\forall\alpha.\tau$$

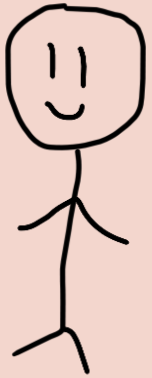
Idea behind for-all: can **substitute** α with any other type to give a new type

$$\texttt{id}\colon \forall\alpha.\,\alpha \to \alpha$$
$$\texttt{id} = \Lambda\alpha.\,\lambda x\colon\alpha.\,x$$
$$\texttt{id Int} = (\lambda x\colon\alpha.\,x)[\alpha \coloneqq \texttt{Int}]$$
$$= \lambda x\colon\texttt{Int}.\,x$$

Alice



Bob



$$\mathtt{id} : \forall \alpha. \alpha \to \alpha$$
$$\mathtt{id} = \Lambda \alpha. \lambda x : \alpha. x$$

Alice: “Here’s a polymorphic type; I don’t know what α is. I can only refer to it opaquely as α. You can replace it with whatever type you want.”

Are there "there-exists" types?
Yes! These are called **existential** types!

$$\exists\alpha.\tau$$

Idea behind exists: there is **some** type which inhabits α to give a new type

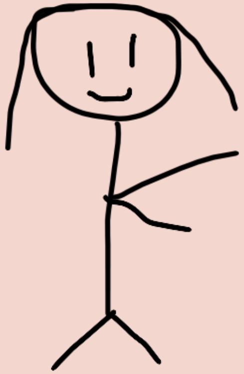
$\exists\alpha.[\alpha]$ means "some" list

`[1,2] ::` $\exists\alpha.[\alpha]$ is valid since because we can let α be `Int`

`"abc" ::` $\exists\alpha.[\alpha]$ is also valid because we can let α be `Char`

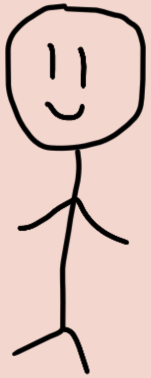
`[1,'a'] ::` $\exists\alpha.[\alpha]$ is invalid

Alice



$x : \exists \alpha. [\alpha]$
`x = [1,2]`

Bob



Alice: "Here's an existential type. I know what α is but I won't tell you. **You** can only refer to it opaquely as α."

Polymorphism: implementer **does not know the type**, **must ignore it**. User **chooses the type**.

Existential types: implementer **chooses the type**. User **does not know the type**, **must ignore it**.

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]
ls = [x, y, z] -- error!
```

Ideally, `ls` has type $[\exists \alpha . \texttt{HasArea}\ \alpha \Rightarrow \alpha]$

Haskell doesn't have existential types, what now?

```
data HasAreaType = HAT (∃α.HasArea α => α)
instance HasArea HasAreaType where
    area (HAT x) = area x
```

We create a new type `HasAreaType` that wraps any type that has area. But, still have the same problem!

Mental model for polymorphism: function that receives type and produces type/term

Mental model for existential types: a pair containing a witness type and the object itself

Object of type $\exists\alpha.\tau$ is pair (β, x) such that x has type $\tau[\alpha \coloneqq \beta]$

`(Int, [1,2])` inhabits $\exists\alpha.[\alpha]$ Because `[1,2] :: [Int]`
`(Char, “abc”)` inhabits $\exists\alpha.[\alpha]$ Because `“abc” :: [Char]`

```
HAT :: (∃a. HasArea a => a)        -> HasAreaType
```

Function that receives existential type can be thought of as a function receiving a pair consisting of type and object

```
HAT :: (a :: *, HasArea a => a)   -> HasAreaType
```

Function that receives existential type can be thought of as a function receiving a pair consisting of type and object

```
HAT :: (a :: *, HasArea a => a)   -> HasAreaType
```

Currying: function receiving more than one parameter can be curried into function that receives one parameter

```python
def add(x, y):
    return x + y

def add(x):
    return lambda y: x + y
```

First parameter is a type, so...

```haskell
HAT :: forall a. HasArea a => a   -> HasAreaType
```

Make it polymorphic!

Polymorphic functions simulate functions over existential types

Other examples:

```
area :: (∃a. HasArea a => a) -> Double
area :: forall a. HasArea a => a -> Double

EqExpr :: (∃a. Eq a => (Expr a, Expr a)) -> Expr Bool
EqExpr :: forall a. Eq a => Expr a -> Expr a -> Expr Bool
```

```haskell
data HasAreaType where
    HAT :: forall a. HasArea a => a -> HasAreaType
instance HasArea HasAreaType where
    area (HAT x) = area x
```

Now we can put terms of different types that implement `HasArea` in a list!

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]

ls :: [HasAreaType]
ls = [HAT x, HAT y, HAT z]
d = totalArea ls -- 27
```

However, in this case, not particularly useful

```haskell
x = Triangle 2 3
y = R "bedroom" (Rectangle 3 4)
z = H [y]

ls :: [Double]
ls = [area x, area y, area z]
d = sum ls -- 27
```

Existential Types

- **Not commonly used**, usually to **abstract over types that have common behaviour**
- Not knowing existential types should **not** affect understanding of typeclasses/polymorphic types
- Existential types as pairs is **very handwave-y**
- Demonstration only serves as **mental model** for why we write polymorphic functions where **return type does not depend on type parameter**

Key Point

- We should **not** replicate OO design patterns in FP just because they are familiar
- Trying to skirt around the restrictions of the type system is, generally, not a good idea—work **with** the type system

Thank you

Foo Yong Qi

yongqi@nus.edu.sg

https://yongqi.foo/