IA014: Advanced Functional Programming

5. Type Classes

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Polymorphism III

Polymorphism – recap

Types of polymorphism

- *parametric* polymorphism
  - "all types"
  - Allows single piece of code to be typed parametrically, i.e. using type variables, and instantiated when needed.
  - All instances behave the same.
  - *HM type system*
- *ad-hoc* polymorphism
  - "some types"
  - *overloading:* one function has many implementations (differing by the types of the arguments)
  - May behave differently for different types of arguments.

Goal: To *extend* System HM with overloading

Motivation

- parametric functions work for *any type*
- but not all similar functions are parametric:
  - `member :: [a] -> a -> Bool`
    makes sense only if `a` can be tested for equality
  - `sort :: [a] -> [a]`
    makes sense only if `a` can be tested for ordering
  - `sumOfSquares :: [a] -> a`
    makes sense only if `a` supports arithmetic operations

Overloading arithmetic

Two approaches:

1. operations are overloaded, but not user defined functions
   - `3*3 :: Int, 3.14*3.14 :: Float`
   - however `square x = x*x :: Int -> Int`
     `square 3.14*3.14` is *illegal*
   - used by StandardML
2. different function for each input type
   - `square x = x*x` defines *two versions*
     one `Int -> Int` and one `Float -> Float`
   - but consider:
     `squares(x,y,z) = (square x, square y, square z)`
   - *8 different versions!*

Overloading equality

Three approaches:

1. equality can be overloaded on any *monotype* that *admits equality* (i.e. not function type!)
   - problem: cannot define

     ```
     member [] x = False;
     member (y:ys) x | x==y  = True
                     | otherwise = member ys x
     ```

   - original STANDARDML
2. make equality *fully polymorphic*
   - `(==) ::  a -> a -> Bool`
   - problem: *run-time error* if applied to functions
3. polymorphic in limited way
   - `(==) ::  ''a -> ''a -> Bool`
   - `''a` – type that admits equality (eqtype)
   - STANDARDML

Solution: Type Classes

- allow users to define overloaded functions `square`, `squares`, `member`
- generalize eqtypes of SML to arbitrary types
- no exponential blowup in the number of versions
- there is nothing special about equality and arithmetic
  user can define new collections of overloaded functions
- type inference works
- can be translated to System HM

Type Classes

Type classes by example: Num

```
class Num a where
   (+), (*) :: a -> a -> a
   negate:: a -> a

instance Num Int where
   (+) = addInt
   (*) = mulInt
   negate = negInt

instance Num Float where
   (+) = addFloat
   (*) = mulFloat
   negate = negFloat

square:: Num a => a -> a
square x = x * x
```

Type classes lingo

type class declaration

```
class Num a where
   (+), (*) :: a -> a -> a
   negate:: a -> a
```

- defines a new class `Num` with three operations

instance declaration

```
instance Num Int where
   (+) = addInt
   (*) = mulInt
   negate = negInt
```

- starts by assertion "`Int` is an instance of `Num`"
- justifies the assertion by giving the function definitions

Implementing a type class

```
data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)
add (NumDict a m n) = a
mul (NumDict a m n) = m
neg (NumDict a m n) = n

numDInt :: NumD Int
numDInt = NumDict addInt mulInt negInt

numDFloat :: NumD Float
numDFloat = NumDict addFloat mulFloat negFloat

square’ :: NumD a -> a -> a
square’ numDa x = mul numDa x x
```

Translation described

- for each *class* we introduce
  - new type ("method dictionary")
    ```
    data NumD a = NumDict (a -> a -> a) (a -> a -> a) (a -> a)
    ```
    - `NumD` is a *type constructor*
    - `NumDict` is a *value constructor*
  - methods to access this dictionary
    ```
    add (NumDict a m n) = a
    ```
- each *class instance*
  - is translated to a value of the "dictionary type"
    ```
    numDInt = NumDict addInt mulInt negInt
    ```
- each *term*
  - is replaced by the corresponding "access method" term
    `3.14 + 3.14` $\longrightarrow$ `add numDFloat 3.14 3.14`
- similarly for defined *functions*

  `square 3` $\longrightarrow$ `square' numDInt 3`

Functions with multiple dictionaries

definition

```
squares :: (Num a, Num b, Num c) => (a,b,c) -> (a,b,c)
squares (x, y, z) = (square  x, square y, square z)
```

- parameters of class `Num`
- completely natural syntax

translation

```
squares' :: (NumD a, NumD b, NumD c) -> (a,b,c) -> (a,b,c)
squares' (numDa, numDb, numDc) (x, y, z) =
          (square' numDa x, square' numDb y, square' numDc z)
```

- *succinct*: we need just *one version* of the function, not eight

Type Classes by example: Eq

```
class Eq a where
   (==) :: a -> a -> Bool
instance Eq Int where
   (==) = eqInt
instance Eq Char where
   (==) = eqChar
member          :: Eq a => [a] -> a -> Bool
member [] y     = False
member (x:xs) y = (x == y) || member xs y
```

translation

```
data EqD a      = EqDict (a -> a -> Bool)
eq (EqDict e)   = e
eqDInt          :: EqD Int
eqDInt          = EqDict eqInt
eqDChar         :: EqD Char
eqDChar         = EqDict eqChar
member'         :: EqD a -> [a] -> a -> Bool
member' eqDa [] y     = False
member' eqDa (x:xs) y = eq eqDa x y || member' eqDa xs y
```

Subclasses

For testing membership of a square, we need both *equality* and *arithmetic*:

```
memsq :: (Eq a, Num a) => [a] -> a -> Bool
memsq xs x = member xs (square x)
```

- natural assumption: every datatype having `(+)`, `(*)` and `negate` defined has also `(==)` defined
- i.e. `Num` is a *subclass* of `Eq`

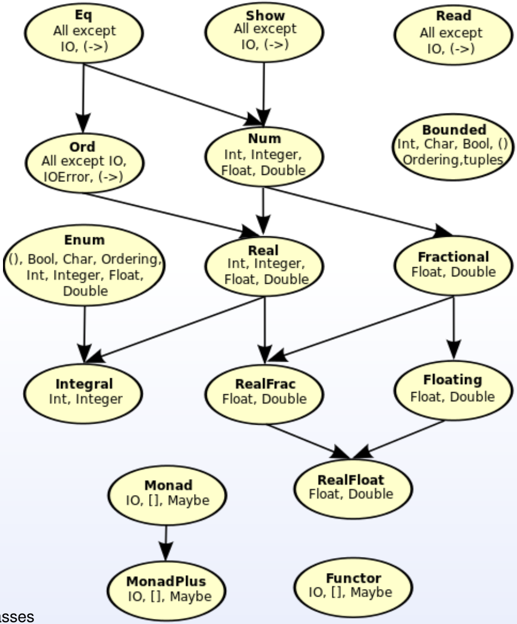
```
class Eq a => Num a where
   (+)     :: a -> a -> a
   (*)     :: a -> a -> a
   negate  :: a -> a
```

We then can write just

```
memsq :: Num a => [a] -> a -> Bool
```

Restriction: no cyclic dependency

Haskell Class Hierarchy



Numeric literals

What is the type of `3`?

- can be e.g. `Integer` or `Float`
- ML: `Integer`
- HASKELL: `Num`!

Literals as type classes

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

Even literals are overloaded:

```
Prelude> :t 1
1 :: Num a => a

inc :: Num a => a -> a
inc x = x + 1
```

Numeric classes in Haskell

```
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
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a
```

Extending Eq

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

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

instance  Eq Integer  where
    (==) = eqInteger

instance  (Eq a, Eq b) => Eq(a,b)  where
    (u,v) == (x,y) = (u == x) && (v == y)
```

Default implementation

In the class declaration we can define *default implementation* for methods:
From GHC.Classes

```
-- | The 'Eq' class defines equality ('==') and inequality ('/=').
-- All the basic datatypes exported by the "Prelude" are instances of
-- and 'Eq' may be derived for any datatype whose constituents are al
-- instances of 'Eq'.
--
-- Minimal complete definition: either '==' or '/='.
--
class  Eq a  where
    (==), (/=)           :: a -> a -> Bool

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

Instances can redefine the behaviour.

Deriving

For Eq, Ord, Enum, Bounded, Show, or Read, the compiler can generate instance declarations automatically:

```
data  Tree a =  Leaf a | Branch (Tree a) (Tree a)
      deriving (Eq, Ord)

instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)     == (Leaf y)          =  x == y
    (Branch l r) == (Branch l’ r’)    =  l == l’ && r == r’
    _            == _                 =  False

instance  (Ord a) => Ord (Tree a)  where
    (Leaf _)     <= (Branch _)        =  True
    (Leaf x)     <= (Leaf y)          =  x <= y
    (Branch _)   <= (Leaf _)          =  False
    (Branch l r) <= (Branch l’ r’)    =  l == l’ && r <= r’ || l <= l’
```

Qualified types

Qualified types

polymorphic types:
$\forall\alpha.f(\alpha)$ can be treated as having any of the types in the set

$$\{f(\mathrm{T}) \mid \mathrm{T} \text{ is a type}\}$$

Restricting polymorphism:

- allow only *some of the types*
- e.g. those satisfying a predicate π
- we write $\forall\alpha.\pi(\alpha) \Rightarrow f(\alpha)$ for the set

$$\{f(\mathrm{T}) \mid \mathrm{T} \text{ is a type} \land \pi(\mathrm{T}) \text{ holds}\}$$

qualified types

- types of the form $\pi \Rightarrow \mathrm{S}$

Entailing relation

- each type system is given by the choice of *predicates* π
- properties are described by the *entailment relation* $\Vdash$ between finite sets of predicates P and Q
- predicates are of the form $\pi = p\mathrm{T}_1 \ldots \mathrm{T}_n$, where p is a n-ary predicate symbol and T_i types
- the relation $\Vdash$ must satisfy the following properties:
    - **monotonicity:** $P \Vdash Q'$ whenerver $P \supseteq Q$
    - **transitivity:** if $P \Vdash Q$ and $Q \Vdash R$, then $P \Vdash R$
    - **closure:** if $P \Vdash Q$ then also $\theta P \Vdash \theta Q$ for any substitution θ
- we write $P \Vdash \pi$ for $P \Vdash \{\pi\}$, and P, π for $P \cup \{\pi\}$

Type classes as qualified types

- *predicates:* $C\ \mathrm{T}$
- *meaning:* T is an instance of the class named C
- additional axioms (examples of):
  - $\emptyset \Vdash Eq\ Int$
  - $Eq\ a \Vdash Eq\ [a]$
- typing rules

$$\frac{P \Vdash \pi \qquad \text{class } Q \Rightarrow \pi}{P \Vdash Q}\ \text{(super)} \qquad \frac{P \Vdash Q \qquad \text{instance } Q \Rightarrow \pi}{P \Vdash \pi}\ \text{(inst)}$$

- example

$$\frac{\texttt{Ord a} \Vdash \texttt{Ord a} \qquad \texttt{class Eq a} \Rightarrow \texttt{Ord a}}{\texttt{Ord a} \Vdash \texttt{Eq a}}\ \text{(super)}$$

Validity of class hierarchy

$$\frac{\text{class } Q \Rightarrow \pi \qquad P \Vdash \theta Q}{\text{instance } P \Rightarrow \theta\pi \text{ is valid}}\ \text{(valid)}$$

example

```
class (Eq a, Show a) => Num a
class Foo a => Bar a
class Foo a
instance (Eq a, Show a) => Foo [a]
instance Num a => Bar [a]
```

$$\frac{\mathbf{c}\ \texttt{Foo a => Bar a} \qquad \dfrac{\dots}{\texttt{Num a} \Vdash \texttt{Foo [a]}}}{\mathbf{i}\ \texttt{Num a => Bar [a]} \text{ is valid}}\ \text{(valid)}$$

$$\frac{\dfrac{\mathbf{c}\ \texttt{(Eq a, Show a) => Num a}}{\texttt{Num a} \Vdash \texttt{(Eq a, Show a)}}\ \text{(class)} \qquad \mathbf{i}\ \texttt{(Eq a, Show a) => Foo [a]}}{\texttt{Num a} \Vdash \texttt{Foo [a]}}$$

Extending HM with qualified types

type syntax

$$\begin{aligned} \mathrm{T} &::= \alpha \mid \mathrm{T} \to \mathrm{T} && \text{monotypes} \\ \mathrm{R} &::= P \Rightarrow \mathrm{T} && \text{qualified types} \\ \mathrm{S} &::= \forall \vec{\alpha}.\mathrm{R} && \text{type schemes} \end{aligned}$$

typing rules

- of the form $P \mid \Gamma \vdash \mathrm{t} : \mathrm{T}$
- meaning: assuming predicates in P and context Γ, t is of type T

$$\frac{P \mid \Gamma \vdash \mathrm{t} : \pi \Rightarrow \mathrm{R} \qquad P \Vdash \pi}{P \mid \Gamma \vdash \mathrm{t} : \mathrm{R}} \text{ (T-PRed)}$$

$$\frac{P, \pi \mid \Gamma \vdash \mathrm{t} : \mathrm{R}}{P \mid \Gamma \vdash \mathrm{t} : \pi \Rightarrow \mathrm{R}} \text{ (T-PInt)}$$

Modified HM Typing rules

$$\frac{x : \mathrm{S} \in \Gamma}{P \mid \Gamma \vdash x : \mathrm{S}} \text{ (T-Var)}$$

$$\frac{P \mid \Gamma, x : \mathrm{T}_1 \vdash \mathrm{t} : \mathrm{T}_2}{P \mid \Gamma \vdash \lambda x.\mathrm{t} : \mathrm{T}_1 \to \mathrm{T}_2} \text{ (T-Abs)}$$

$$\frac{P \mid \Gamma \vdash \mathrm{t}_1 : \mathrm{T}_1 \to \mathrm{T}_2 \qquad P \mid \Gamma \vdash \mathrm{t}_2 : \mathrm{T}_1}{P \mid \Gamma \vdash \mathrm{t}_1\ \mathrm{t}_2 : \mathrm{T}_2} \text{ (T-App)}$$

$$\frac{P \mid \Gamma \vdash \mathrm{t}_1 : \mathrm{S} \qquad Q \mid \Gamma, x : \mathrm{S} \vdash \mathrm{t}_2 : \mathrm{T}}{P \cup Q \mid \Gamma \vdash \mathrm{let}\ \ x = \mathrm{t}_1\ \ \mathrm{in}\ \ \mathrm{t}_2 : \mathrm{T}} \text{ (T-Let)}$$

$$\frac{P \mid \Gamma \vdash \mathrm{t} : \mathrm{S}' \qquad \mathrm{S}' \sqsubseteq \mathrm{S}}{P \mid \Gamma \vdash \mathrm{t} : \mathrm{S}} \text{ (T-Inst)}$$

$$\frac{P \mid \Gamma \vdash \mathrm{t} : \mathrm{S} \qquad \alpha \notin FV(\Gamma) \cup FV(P)}{P \mid \Gamma \vdash \mathrm{t} : \forall \alpha.\mathrm{S}} \text{ (T-Gen)}$$

Type inference

- by combining the rules we can again produce syntax-directed type system (in the same way as for HM)
- we extend Algorithm W in the same fashion

example

```
example x [] = False;
example x (y:ys) | y > x = True
                 | otherwise = (y == x && ys == [x])
```

- the type is `T = a -> [a] -> Bool`
- constraints: `Q = {Ord a, Eq a, Eq [a]}`
  - `Ord a`: from `y > x`
  - `Eq a`: from `y == x`
  - `Eq [a]`: from `ys == [x]`

Type inference II

The set `Q = {Ord a, Eq a, Eq [a]}` can be *simplified*:

- using *instance declaration* `instance Eq a => Eq [a]`
  `{Eq a, Eq [a]}` simplifies to `{Eq a}`
- using *class declaration* `class Eq a => Ord a`
  `{Eq a, Ord a}` simplifies to `{Ord a}`
- therefore `{Ord a, Eq a, Eq [a]}` simplifies to `{Ord a}`

The resulting type is $Q \Rightarrow P$, where

- `T = a -> [a] -> Bool`
- `Q = {Ord a}`

Therefore

```
example :: Ord a => a -> [a] -> Bool
example x [] = False;
example x (y:ys) | y > x = True
                 | otherwise = (y == x && ys == [x])
```

Detecting errors

```
Prelude> 'a' + 1

<interactive>:18:5:
    No instance for (Num Char) arising from a use of ‘+’
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 'a' + 1
    In an equation for ‘it’: it = 'a' + 1
```

Type class extensions

- *constructor classes*
  - parametrising class over a type constructor (instead of a type)
  - higher-kinded polymorphism
  - directly supports monads
- *multi-parameter type classes*
- *functional dependencies*

Constructor classes

Overloading map

map on lists

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

map on Maybe

```
data Maybe a                 = Nothing | Just a

mapMay                       :: (a->b) -> Maybe a -> Maybe b
mapMay _ Nothing             = Nothing
mapMay f (Just x)            = Just (f x)
```

map on trees

```
data Tree a                  = Leaf a | Branch (Tree a) (Tree a)

mapTree                      :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf x)           = Leaf (f x)
mapTree f (Branch xl xr) = Branch (mapTree f xl) (mapTree f xr)
```

Comparing `maps`

- `[]`, `Tree` and `Maybe` are *type constructors*
  (functions from types to types)
- `map`, `mapTree` and `mapMay` have the "same" type
  `(a->b) -> t a -> t b`
  where `t` can be `[]`, `Tree` or `Maybe`
- the correct `map` to be applied can be easily determined from the context
  e.g. `map (1+) [1,2,3]` vs. `map (1+) (Just 1)`

however

- such universal `map` is not typeable in HM
- there is no way of extending `map` to other similar structures

type classes do not help

- remember: `class Name a where ...`
  (`a` is a type here)

Functor class

```
class  Functor f  where
    fmap        :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Tree where
    fmap f (Leaf x)      = Leaf (f x)
    fmap f (Branch xl xr) = Branch (fmap f xl) (fmap f xr)

instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)
```

constructor classes

- `Functor` is an example of a *constructor class*
- parameter of `Functor` is a type constructor, not a type

Kinding

Can `Functor` be applied to any type?

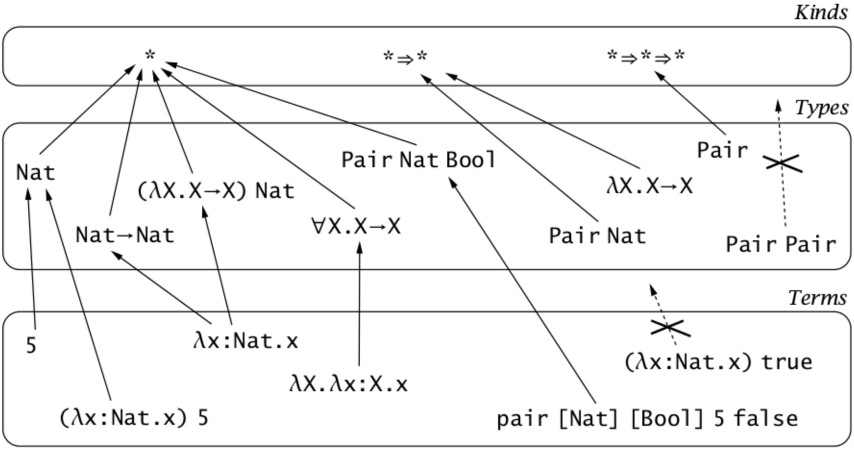
- for `Functor Int` we would get
  `fmap ::  (a->b) -> Int a -> Int b`
- obviously ill-formed, does not "typecheck"

Kinds ("types of types")

- monomorphic types have kind $*$
- unary type constructor, which takes a type of kind κ_1 and returns a type of kind κ_2 has kind κ_1 `->` κ_2.
- examples:

```
Int, Float          :: *
List, Maybe         :: * -> *
(->), (,)           :: * -> * -> *
```

Kinds: examples



(Taken from [Pierce].)

Implementing constructor classes

For each kind κ we have a collection of constructors C^κ

$$
\begin{array}{rlll}
C^\kappa ::= & \chi^\kappa & & \text{constants} \\
\mid & \alpha^\kappa & & \text{variables} \\
\mid & C^{\kappa_1 \to \kappa_2} C^{\kappa_1} & & \text{applications}
\end{array}
$$

Extending HM

Surprisingly straightforward:

$$
\frac{P \mid \Gamma \vdash \mathrm{t} : \forall \alpha^\kappa . \mathrm{S} \qquad C \in C^\kappa}{P \mid \Gamma \vdash \mathrm{t} : \{C/\alpha^\kappa\}\mathrm{S}} \text{ (T-Inst')}
$$

$$
\frac{P \mid \Gamma \vdash \mathrm{t} : \mathrm{S} \qquad \alpha^\kappa \notin FV(\Gamma) \cup FV(P)}{P \mid \Gamma \vdash \mathrm{t} : \forall \alpha^\kappa . \mathrm{S}} \text{ (T-Gen)}
$$

- kinded unification
- effective type inference

Kind inference

Kind annotations are actually not needed – can be *inferred*:

- for class definitions:

```
class  Functor f  where
    fmap        :: (a -> b) -> f a -> f b
```

  - `->` has kind `* -> * -> *`
  - therefore both `a` and `b` must have kind `*`, and
  - `f` must have kind `* -> *`, therefore
  - the type of `fmap` must be

$$\forall f^{*\to *}.\forall a^{*}.\forall b^{*}.Functor\,f \Rightarrow (a \to b) \to (f\ a \to f\ b)$$

- for datatype definitions:

```
data tConst a1 ... am = vConst1 | ... | vConstn
```

  - `tConst` has kind $\kappa_1 \to \ldots \kappa_m \to *$, where
  - $\kappa_1, \ldots, \kappa_m$ are the inferred kinds for `a1, ..., am`

- inference is easy thanks to having no kind abstraction

Multiparameter type classes

Motivation

- in HASKELL98, a class can only qualify a single type:

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

- however multiple types may be useful

Example: uniform interface to collection types

```
class Collects e s where
   empty   :: s
   insert  :: e -> s -> s
   member  :: e -> s -> Bool
```

some instances

```
instance Eq e => Collects e [e] where ...
instance Eq e => Collects e (e -> Bool) where ...
instance Collects Char BitSet where ...
instance (Hashable e, Collects e s)
    => Collects e (Array Int s) where ...
```

Collections

Type `s` is a collection of elements of type `e`:

```
class Collects e s where
   empty   :: s
   insert  :: e -> s -> s
   member  :: e -> s -> Bool
```

Problems

1. ambiguity:

   ```
   empty :: Collects e s => s
   ```

2. dropping `empty` does not help either:

   ```
   f x y col = insert x (insert y col)
   f :: (Collects a c, Collects b c) => a -> b -> c -> c

   g c = f True 'a' col
   g :: (Collects Bool c, Collects Char c) => c -> c
   ```

Constructor class approach

Abstract over the type constructor `c`:

```
class Collects e c where
   empty   :: c e
   insert  :: e -> c e -> c e
   member  :: e -> c e -> Bool
```

- `f ::  (Collects e c) => e -> e -> c e -> c e`
- `g` is rejected
- works well for lists
  `c e` instantiates to `[] e` in this case
- for others either impossible (`BitSet`) or requires tricks:

```
newtype CharFun e = MkCharFun (e -> Bool)
instance Eq e => Collects e CharFun where ...
```

Functional dependencies

Key idea: `s` **uniquely determines** `e`

```
class Collects e s | s -> e where
   empty    :: s
   insert   :: e -> s -> s
   member   :: e -> s -> Bool
```

- `s -> e` above is a *functional dependency*
- some examples:

```
class C a b where ...
class D a b | a->b where ...
class E a b | a->b, b->a ...
```

- either of these declarations is fine on its own:

```
instance D Bool Int where ...
instance D Bool Char where ...
```

- together they are rejected
- following is not allowed at all:

```
instance D [a] b where ...
```

Another example

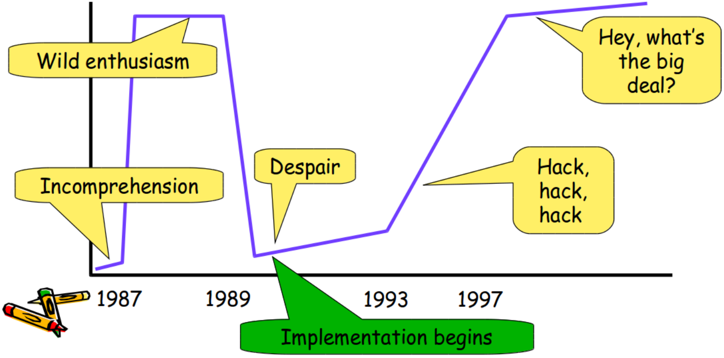
```
class Mult a b c where
  (*) :: a -> b -> c

instance Mult Matrix Matrix Matrix where ...
instance Mult Matrix Vector Vector where ...

m1, m2, m3 :: Matrix
(m1 * m2) * m3              -- type error; type of (m1*m2) is ambiguo
(m1 * m2) :: Matrix * m3    -- this is ok
```

Solution:

```
class Mult a b c | (a,b) -> c where
  (*) :: a -> b -> c
```

Type classes over time

- Type classes are the most unusual feature of Haskell's type system



S. Peyton Jones: Wearing the Hair Shirt. A retrospective on Haskell.

Reading list

primary papers

- P. Wadler, S. Blott: *How to make ad-hoc polymorphism less ad hoc*. POPL'89.
- M. Jones: *A theory of qualified types*. ESOP'92.
- M. Jones: *A system of cnstructor classes*. FPCA'93.
- M. Jones: *Type Classes with Functional Dependencies*. ESOP'00.

further reading

- M. Jones: *Functional Programming with Overloading and Higher-Order Polymorphism*. AFPT Spring School 1995.
- *A History of Haskell: Being Lazy With Class*. 2007. (Section 6)
- J. Peterson and M. Jones: *Implementing Type Classes*. PLDI'93.
- C. Hall, K. Hammond, S. Peyton Jones, P. Wadler: *Type classes in Haskell*. ESOP'94.