CS510AP

Type Classes

Generic Programming
What are type classes

• Type classes are unique to Haskell
• They play two (related) roles
• Overloading
  – A single name indicates many different functions.
  – E.g. (+) might mean both integer and floating point addition.
• Implicit Parameterization
  – An operation is implicitly parameterized by a set of operations that are used as if they were globally available resources.
Attributes of Haskell Type Classes

- Explicitly declared
  - class and instance declarations

- Implicit use
  - Type inference is used to decide:
    - When a type class is needed.
    - What class is meant.

- Uniqueness by type
  - The inference mechanism must decide a unique reference to use.
  - No overlapping instances
The Haskell Class System

• Think of a Qualified type as a type with a Predicate

• Types which meet those predicates have "extra" functionality.

• A class definition defines the type of the "extra" functionality.

• An instance declarations defines the "extra" functionality for a particular type.
Example Class Definition

class Eq a where

    (==), (/=) :: a -> a -> Bool
    x /= y       = not (x==y)

class (Eq a) => Ord a where

    compare :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min :: a -> a -> a
Properties of a class definition

- `class (Eq a) => Ord a where`
- `compare :: a -> a -> Ordering`
- `(<=), (>=), (>), (<) :: a -> a -> Bool`
- `max, min :: a -> a -> a`

- Class name is capitalized, think of this as the name of a type predicate that qualifies the type being described.
- Classes can depend on another class or in other words require another classes as a prerequisite.
- The methods of a class are functions whose type must depend upon the type being qualified.
- There can be more than one method.
- The methods can be ordinary (prefix) functions or infix operators.
Overloading – The Num Class

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

    x - y = x + negate y
    fromInt = fromIntegral
Extending the Num Class with Complex

- Make Complex numbers an instance of class Num.
  - data Complex = C Float Float
  - An instance of Num, must first be an instance of Eq and Show and provide methods for (+), (-), and (*) (amongst others).
  - First provide the numeric operators
    - complex_add (C x y) (C a b) = C (x+a) (y+b)
    - complex_sub (C x y) (C a b) = C (x-a) (y-b)

- complex_mult (C x y) (C a b) = C (x*a - y*b) (x*b + a*y)
Num Instance

• Then make the instance declaration
  - instance Eq(Complex) where
    - (C x y) == (C a b) = x==a && y==b

  - instance Show(Complex) where
    - showsPrec = error "No show for complex"
    - showList = error "No show for complex"

  - instance Num(Complex) where
    - x + y = complex_add x y
    - x - y = complex_sub x y
    - x * y = complex_mult x y

• Note that the Show instance is quite imprecise, but this will cause an error only if it is ever used
Overview

- Classes provide a formal and precise mechanism for introducing overloading in a strongly typed language.
- The inference mechanism always chooses exactly one of the overloaded definitions.
- The inference mechanism can be used to perform compile-time computation, by the judicious use of class and instance declarations.
  - Sort of like programming in prolog.
Generic Programming

• Generic programming is writing one algorithm that can run on many different datatypes.
• Saves effort because a function need only be written once and maintained in only one place.
• Examples:
  - equal :: a -> a -> Bool
  - display :: a -> String
  - marshall :: a -> [Int]
  - unmarshall :: [Int] -> a
Flavors

1) Universal type embedding
2) Shape based type embeddings
   1) With isomorphism based equality proofs
   2) With leibniz based equality proofs
3) Cast enabling embeddings

4) In this world
   1) equal :: Rep a -> a -> a -> a -> Bool
   2) display :: Rep a -> a -> String
   3) marshall :: Rep a -> a -> [ Int ]
   4) unmarshall :: Rep a -> [Int] -> a
Use of Overloading

• Overloading in Haskell makes generics quite easy to use.

• Instead of types like:
  - equal :: Rep a -> a -> a -> Bool
  - display :: Rep a -> a -> String
  - marshall :: Rep a -> a -> [ Int ]
  - unmarshall :: Rep a -> [Int] -> a

• We use overloading to build functions with types like
  - equal :: Generic a => a -> a -> a -> Bool
  - display :: Generic a => a -> String
  - marshall :: Generic a => a -> [ Int ]
  - unmarshall :: Generic a => [Int] -> a

Class Generic a where
  rep :: Rep a
Getting started

• We’ll start with the explicit Rep based approach where the representation type is passed as an explicit argument.

• How do we represent types as data?
• That depends in what you want to do with the types.
Universal Embedding

- Let there be a universal type that can encode all values of the language (a datatype that uses dynamic tags (constructor functions) to distinguish different kinds of values.

```haskell
data Val
    = Vint Int -- basic types
    | Vchar Char
    | Vunit
    | Vfun (Val -> Val) -- functions
    | Vdata String [Val] -- data types
    | Vtuple [Val ] -- tuples
    | Vpar Int Val
```
Interesting functions

- Note there are several interesting functions on the universal domain Value
  - Equality
  - Mapping
  - Showing
  - Numeric operations
instance Show (Val) where
    show (Vint n) = show n
    show (Vchar c) = show c
    show Vunit = "()"
    show (Vfun f) = "fn"
    show (Vdata s []) = s
    show (Vdata s xs) =
        "(" ++ s ++ plist " " xs " " ")"
    show (Vtuple xs) = plist "(" ++ xs ++ ")"
    show (Vpar n x) = show x
instance Eq Val where
  x == y = test x y
  where test (Vint n) (Vint m) = n==m
  test (Vchar n) (Vchar m) = n==m
  test Vunit Vunit = True
  test (Vdata s xs) (Vdata t ys) =
    s==t && tests xs ys
  test (Vtuple xs) (Vtuple ys) =
    tests xs ys
  test (Vpar n x) (Vpar m y) = test x y
  test _ _ = False
  tests [] [] = True
  tests (x:xs) (y:ys) =
    test x y && tests xs ys
  tests _ _ = False
Num class

instance Num Val where

    fromInt x = Vint x

    (+) (Vint x) (Vint y) = Vint (x+y)

    (*) (Vint x) (Vint y) = Vint (x*y)

    (-) (Vint x) (Vint y) = Vint (x - y)
Mapping

mapVal :: (Val -> Val) -> Val -> Val
mapVal f (Vpar n a) = Vpar n (f a)
mapVal f (Vint n) = Vint n
mapVal f (Vchar c) = Vchar c
mapVal f Vunit = Vunit
mapVal f (Vfun h) =
    error "can't mapVal Vfun"
mapVal f (Vdata s xs) =
    Vdata s (map (mapVal f) xs)
mapVal f (Vtuple xs) =
    Vtuple(map (mapVal f) xs)
Flavor 1 for Type reps

```haskell
data Rep t = Univ (t -> Val) (Val -> t)
```

We represent a type \( t \) by a pair of functions that inject and project from the universal type.

Property \((\text{Univ} \ f \ g) :: \text{Rep} \ t\)

For all \( x :: t \) . \( g(f \ x) == x \)

Functions

\[
\text{into} \ (\text{Univ} \ f \ g) = f \\
\text{out} \ (\text{Univ} \ f \ g) = g
\]
Example Reps

\[\text{intU} = \text{Univ Vint } (\lambda (\text{Vint } n) \rightarrow n)\]
\[\text{charU} = \text{Univ Vchar } (\lambda (\text{Vchar } c) \rightarrow c)\]
\[\text{unitU} = \text{Univ } (\text{const Vunit}) (\text{const } () )\]

\[\text{pairU} : : (\text{Rep a}) \rightarrow (\text{Rep b}) \rightarrow \text{Rep } (a,b)\]
\[\text{pairU } (\text{Univ } \text{to1 from1}) (\text{Univ } \text{to2 from2}) = \text{Univ } f \ g\]
  \[\text{where } f \ (x,y) = \text{Vtuple}[\text{to1 } x, \text{to2 } y]\]
  \[g \ (\text{Vtuple}[x,y]) = (\text{from1 } x, \text{from2 } y)\]

\[\text{arrowU } r1 \ r2 = \text{Univ } f \ g\]
  \[\text{where } f \ h = \text{Vfun}(\text{into } r2 \ . \ h \ . \ \text{out } r1)\]
  \[g \ (\text{Vfun } h) = \text{out } r2 \ . \ h \ . \ \text{into } r1\]
Generic Functions

equal (Univ into from) x y =
    (into x) == (into y)

display (Univ into from) x = show (into x)

Strategy:
1) Push (or pull) values into (out of) the universal domain.
2) Then manipulate the “typeless” data
3) Pull the result (if necessary) out of the universal domain.
Marshall and Unmarshall

marshall (Univ to from) x =
    reverse (flat (to x) [])

flat :: Val -> [Int] -> [Int]
flat (Vint n) xs = n : 1 : xs
flat (Vchar c) xs = ord c : 2 : xs
flat Vunit xs = 3 : xs
flat (Vfun f) xs = error "no Vfun in marshall"
flat (Vdata s zs) xs =
    flatList zs (length zs : (flatString s (5 : xs)))
flat (Vtuple zs) xs =
    flatList zs (length zs : 6 : xs)
flat (Vpar x) xs = flat x (7 : xs)

flatList [] xs = xs
flatList (z:zs) xs = flatList zs (flat z xs)
unmarshall (Univ to from) xs = from j
  where (j,ks) = (unflat xs)

unflat :: [Int] -> (Val,[Int])
unflat (1: x : xs) = (Vint x,xs)
unflat (2: x : xs) = (Vchar (chr x),xs)
unflat (3: xs) = (Vunit,xs)
unflat (5: xs) = (Vdata s ws,zs)
  where (s,n : ys) = unflatString xs
    (ws,zs) = unflatList n ys
unflat (6: n : xs) =
  (Vtuple ws,zs) where (ws,zs) = unflatList n xs
unflat (7: xs) = (Vpar x,ys) where (x,ys) = unflat xs
unflat zs =
  error ("Bad Case in unflat of unmarshall"++ show zs)

unflatList 0 xs = ([],xs)
unflatList n xs = (v:vs,zs)
  where (v,ys)= unflat xs
    (vs,zs) = unflatList (n-1) ys
Generic Map

\[
gmap ::
    \text{Rep } b \rightarrow \text{Rep } c \rightarrow
    (\forall a . \text{Rep } a \rightarrow \text{Rep}(t \ a)) \rightarrow
    (b \rightarrow c) \rightarrow t \ b \rightarrow t \ c
\]

\[
gmap \ \text{repB} \ \text{repC} \ t \ f \ x =
    \text{out } \text{repLC} \ (\text{help} \ (\text{into } \text{repLB} \ x))
\]

where \text{repLB} = t \ \text{repB}
\text{repLC} = t \ \text{repC}
\text{help} \ \text{xs} = \text{mapVal } \text{trans } \text{xs}
\text{trans} \ x =
    \text{into } \text{repC} \ (f(\text{out } \text{repB} \ x))
\]
Using Overloading

• We can use overloading to build representations automatically

```haskell
class Generic t where
  univ :: Rep t

class Generic1 t where
  univ1 :: Rep a -> Rep(t a)

class Generic2 t where
  univ2 :: Rep a -> Rep b -> Rep (t a b)
```
Instances

instance Generic Int where
    univ = intU

instance Generic Char where
    univ = charU

instance Generic () where
    univ = unitU

instance
    (Generic1 t,Generic a) => Generic(t a)
where
    univ = univ1 univ
Higher order instances

instance Generic a =>
    Generic1 ((,) a)
where
    univ1 = pairU univ

instance Generic a =>
    Generic1 ((->) a)
where
    univ1 = arrow univ
Datatype instances

list (Univ to from) = Univ h k

where

  h [] = Vdata "[]" []
  h (x:xs) = Vdata ":" [ Vpar 1 (to x),h xs]
  k (Vdata "[]" []) = []
  k (Vdata ":" [Vpar 1 x,xs]) = (from x) : k xs

instance Generic1 [] where
  univ1 = list
Either data type

eitherU (Univ to1 from1) (Univ to2 from2) = Univ h k
where
  h (Left x) = Vdata "Left" [Vpar 1 (to1 x)]
  h (Right x) = Vdata "Right" [Vpar 2 (to2 x)]
  k (Vdata "Left" [Vpar 1 x]) = Left (from1 x)
  k (Vdata "Right" [Vpar 2 x]) = Right (from2 x)

instance Generic a =>
  Generic1 (Either a) where
  univ1 = eitherU univ
Overloaded functions

hmap :: (Generic a, Generic b, Generic1 c) => (a -> b) -> c a -> c b
hmap f x = gmap univ univ univ1 f x

disp :: Generic a => a -> [Char]
disp x = display univ x

eq :: Generic a => a -> a -> Bool
eq x y = equal univ x y

marsh :: Generic a => a -> [Int]
marsh x = marshall univ x

unmarsh :: Generic a => [Int] -> a
unmarsh x = unmarshall univ x
Flavors 2 & 3 to Type reps

- Another viable approach to representing types as data is to use a “shape” based approach.
- In this approach, types are represented by a parameterized datatype, with constructors for each “shape” that a type can take on.
- Shape based approaches make use of types which represent proofs of equality between two types.
- So lets build a “shape” based approach parameterized by a proof type.
Shapes

data Shape eq t
    = Sint (eq t Int)
    | Schar (eq t Char)
    | Sunit (eq t ())
    | forall a b .
        Sfun (Shape eq a) (Shape eq b) (eq t (a->b))
    | forall a b .
        Spair (Shape eq a) (Shape eq b) (eq t (a,b))
    | forall a b .
        Splus (Shape eq a) (Shape eq b) (eq t (Either a b))
    | forall a . Sdata Name (Shape eq a) (eq t a) (Ptr a)
Names

data Name = App String [Name] deriving Eq

name :: Shape p t -> Name
name (Sint p) = App "Int" []
name (Schar p) = App "Char" []
name (Sunit p) = App "()" []
name (Sfun x y p) = App "->" [name x, name y]
name (Spair x y p) = App "(,)" [name x, name y]
name (Splus x y p) = App "Either" [name x, name y]
name (Sdata t x y p) = t
name (Scon s t) = name t
name (Spar n x) = name x
Equality Proofs

- What operations should equality proofs support?

```haskell
class EqProof proof where
    from :: proof a b -> (a -> b)
    to   :: proof a b -> (b -> a)
    self :: proof a a
    inv  :: proof a b -> proof b a
    assoc:: proof a b -> proof b c -> proof a c
```
Simple Shapes

intP :: EqProof p => Shape p Int
intP = (Sint self)

charP :: EqProof p => Shape p Char
charP = (Schar self)

unitP :: EqProof p => Shape p ()
unitP = (Sunit self)

pairP :: EqProof p =>
        Shape p a -> Shape p b -> Shape p (a,b)
pairP x y = (Spair x y self)

plusP :: EqProof p =>
        Shape p a -> Shape p b -> Shape p (Either a b)
plusP x y = (Splus x y self)
Generic Show

\texttt{rShow :: EqProof }p\texttt{ => Shape }p\texttt{ a }\rightarrow\texttt{ a }\rightarrow\texttt{ String}

\texttt{rShow (Sint }p\texttt{) }x\texttt{ = show (from }p\texttt{ x)}
\texttt{rShow (Schar }p\texttt{) }c\texttt{ = show (from }p\texttt{ c)}
\texttt{rShow (Sunit }p\texttt{) }x\texttt{ = "()"}
\texttt{rShow (Spair }a\texttt{ b }p\texttt{) }x\texttt{ = "(}++\texttt{rShow }a\texttt{ m++},"++\texttt{rShow }b\texttt{ n++})"}
\hspace{1em}\texttt{where (m,n) = from }p\texttt{ x}
\texttt{rShow (Splus }a\texttt{ b }p\texttt{) }x\texttt{ =}
\hspace{1em}\texttt{case (from }p\texttt{ x) of}
\hspace{2em}\texttt{Left }x\texttt{ -> "(Left "++\texttt{rShow }a\texttt{ x++})"}
\hspace{2em}\texttt{Right }x\texttt{ -> "(Right "++\texttt{rShow }b\texttt{ x++})"}
\texttt{rShow (Sdata trm inter }p\texttt{ ptr) }x\texttt{ = rShow inter (from }p\texttt{ x)}
\texttt{rShow (Scon }s\texttt{ (Sunit }p\texttt{)) }x\texttt{ = }s\texttt{ -- Nullary constructor}
Generic Equality

\[ \text{rEqual} :: \text{EqProof} p \Rightarrow \text{Shape} p a \rightarrow a \rightarrow a \rightarrow \text{Bool} \]

rEqual (Sint ep) x y = from ep x == from ep y
rEqual (Schar ep) x y = from ep x == from ep y
rEqual (Sunit ep) x y = from ep x == from ep y
rEqual (Spair a b ep) x y =
  case (from ep x, from ep y) of
    ((m1,n1), (m2,n2)) ->
      rEqual a m1 m2 && rEqual b n1 n2
rEqual (Splus a b ep) x y =
  case (from ep x, from ep y) of
    (Left m, Left n) -> rEqual a m n
    (Right m, Right n) -> rEqual b m n
    (_, _) -> False
rEqual (Sdata trm inter p ptr) x y =
  rEqual inter (from p x) (from p y)
Isomorphism proofs

data Iso a b = Ep (a->b) (b->a)

instance EqProof Iso where
  from (Ep f g) = f
  to   (Ep f g) = g
  self = Ep id id
  inv f = Ep (to f) (from f)
  assoc f g =
    Ep (from g . from f) (to f . to g)
Iso based Representations

data Rep t = Iso (Shape Iso t)

liftI f (Iso x) (Iso y) = Iso(f x y)

intI :: Rep Int
intI = Iso intP
charI :: Rep Char
charI = Iso charP
unitI :: Rep ()
unitI = Iso unitP

pairI :: Rep a -> Rep b -> Rep (a,b)
pairI = liftI pairP
plusI :: Rep a -> Rep b -> Rep (Either a b)
plusI = liftI plusP
funI :: Rep a -> Rep b -> Rep (a -> b)
funI = liftI funP
Iso based datatype Rep's

```haskell
listI :: Rep a -> Rep [a]
listI (Iso a) = Iso shape
  where t = App "[]" [name a]
    intermediateShape =
      plusP unitP (pairP a shape)
    shape = Sdata t intermediateShape
    proof listInterPtr

proof = Ep f g
f [] = Left ()
f (x:xs) = Right(x,xs)
g (Left ()) = []
g (Right(x,xs)) = x:xs
```
Leibniz Proofs

data Leibniz a b =
    Eq { eq1 :: forall f. f a -> f b }

instance EqProof Leibniz where
    to e = unId . eq1 (inv e) . Id
    from e = unId . eq1 e . Id
    self = Eq id
    inv = flip eq2 self
    assoc = flip eq1

newtype Id a = Id { unId :: a}
newtype Flip f a b = Flip { unFlip :: f b a }
eq2 :: Leibniz a b -> f a c -> f b c
eq2 e = unFlip . eq1 e . Flip
Leibniz based Representations

data Rep t = Leibniz (Shape Leibniz t)

liftL f (Leibniz x) (Leibniz y) = Leibniz(f x y)

intL :: Rep Int
charL :: Rep Char
unitL :: Rep ()

intL = Leibniz intP
charL = Leibniz charP
unitL = Leibniz unitP

pairL :: Rep a -> Rep b -> Rep (a,b)
plusL :: Rep a -> Rep b -> Rep (Either a b)
funL :: Rep a -> Rep b -> Rep (a -> b)

pairL = liftL pairP
plusL = liftL plusP
funL = liftL funP
Combining the Approaches

```
data Rep t = Univ (t -> Val) (Val -> t)  
           | Iso (Shape Iso t)  
           | Leibniz (Shape Leibniz t)

class Generic t where
  univ :: Rep t
  iso :: Rep t
  leib :: Rep t

class Generic1 t where
  univ1 :: Rep a -> Rep(t a)
  iso1 :: Rep a -> Rep(t a)
  leib1 :: Rep a -> Rep(t a)

class Generic2 t where
  univ2 :: Rep a -> Rep b -> Rep (t a b)
  iso2 :: Rep a -> Rep b -> Rep (t a b)
  leib2 :: Rep a -> Rep b -> Rep (t a b)
```
Simple Instances

instance Generic Int where
  univ = intU
  iso = intI
  leib = intL

instance Generic Char where
  univ = charU
  iso = charI
  leib = charL

instance Generic () where
  univ = unitU
  iso = unitI
  leib = unitL
Higher Order Instances

instance (Generic1 t, Generic a) =>
    Generic(t a) where
    univ = univ1 univ
    iso = univ1 iso
    leib = leib1 leib

instance Generic a => Generic1 ((,) a) where
    univ1 = pairU univ
    isol = pairI iso
    leib1 = pairL leib

instance Generic a => Generic1 ((->) a) where
    univ1 = arrow univ
    isol = funI iso
    leib1 = funL leib