Functional Programming

2021-2022

Sjaak Smetsers

Type classes revisited

Lecture 6

Outline

- Type classes
- Overloading vs. higher-order functions
- Case study: map-reduce
- Summary

Overloading

- sometimes we wish to use the same name for semantically different, but related functions
  - +, * etc: arithmetic operations (Int, Integer, Float, Double . . .)
  - (==), (/=) : equality and inequality (almost any type)
  - show, read: converting to and from strings (almost any type)
- we want to overload these identifiers
- Haskell's type classes: a systematic approach to overloading
  - (ad-hoc polymorphism vs universal polymorphism)

Class declarations

- new classes can be declared using the `class` mechanism.
- eg the class `Eq` of equality types is declared in the standard prelude as follows:

```
class Eq a where
    (==), (/=) :: a → a → Bool
```

- this declaration states that for a type `a` to be an instance of the class `Eq`, it must support equality and inequality operators of the specified types.
- `(==),(/=)` are member functions of the type class Eq (also called methods)
- types of the member functions:

```
(==),(/=):: (Eq a) ⇒ a → a → Bool
```

- `(Eq a)` ⇒ is a class context; it constrains the type variable `a`

Overloaded functions

- since == is overloaded, `x == y` can be ambiguous (i.e we don't know which instance is used here)
- what happens if the compiler can’t resolve overloading?
- eg list membership uses equality:

```
elem :: (Eq a) ⟹ a ⟶ [a] ⟶ Bool
elem x [ ]        = False
elem x (y : ys) = x == y || elem x ys
```

- elem becomes overloaded
- in general: a (polymorphic) function is called *overloaded* if its type contains one or more class contexts (aka *class constraints*)

Default definitions

- inequality is typically defined in terms of equality (or vice versa)

```
class Eq a where
  (==),(/=):: a ⟶ a ⟶ Bool
  x /= y = not (x == y)
  x == y = not (x /= y)
```

- *default declarations* avoid having to give both definitions every time we introduce a new instance
  - in an instance declaration of `Eq` it suffices now to provide *either* the code for `==` *or* the code for `/=`

Subclasses

- classes can be extended

```
data Ordering = LT | EQ | GT
class (Eq a) ⇒ Ord a where
    compare                 :: a ⟶ a ⟶ Ordering
    (<), (<=), (>), (>=)    :: a ⟶ a⟶ Bool
    max, min                :: a ⟶ a ⟶ a
```

- Ord is a subclass of Eq; conversely, Eq is a superclass of Ord
- subclasses keep class contexts manageable
- necessary if method of superclass is used in one of the default methods
  - eg the default implementation of compare is

```
compare x y
    | x == y    = EQ
    | x <= y    = LT
    | otherwise = GT
```

- Ord includes several default implementations
  - defining either compare or ≤ is sufficient

Bounded

- instances of `Ord` have to implement a *total* order
- occasionally, a type has a *least* and a *greatest* element with respect to that ordering

```
class Bounded a where
  minBound :: a
  maxBound :: a
```

- the type `Int` of machine integers is bounded, the type `Integer` of mathematical integers isn't

```
>>> maxBound :: Int
9223372036854775807
>>> maxBound :: Integer
No instance for Bounded Integer
```

- (it's a *compile-time* error to use `maxBound` at `Integer`)

Enum

- the dot-dot notation is overloaded

```
class Enum a where
  succ, pred :: a → a
  toEnum :: Int → a
  fromEnum :: a → Int
  enumFrom :: a → [a]                         -- [n ..]
  enumFromThen :: a → a → [a]                 -- [n,n' ..]
  enumFromTo :: a → a → [a]                   -- [n .. m]
  enumFromThenTo :: a → a → a → [a] -- [n, n' .. m]
```

- useful for generating test data

```
⋙ [Mon .. Sun]
[Mon, Tue, Wed, Thu, Fri, Sat, Sun]
```

Instance declarations

- the type

```
data Blood = A | B | AB | O
```

- can be made into an equality type as follows:

```
instance Eq Blood where
  A  == A  = True
  B  == B  = True
  AB == AB = True
  O  == O  = True
  _  == _  = False
```

Class instances of parametric types

- to define equality on a parametric type, say, `Tree a` we require equality on the element type `a`
- an instance declaration can have a context too

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
instance (Eq a) ⇒ Eq (Tree a) where
   Leaf x1     == Leaf x2     = x1 == x2
   Leaf _      == Fork _ _    = False
   Fork _ _    == Leaf _      = False
   Fork l1 r1  == Fork l2 r2  = l1 == l2 && r1 == r2
```

- read: if `a` supports equality, then `Tree a` supports equality too

Deriving instances

- defining equality (or instances of some other classes) is tedious, can be derived automatically:

```
data Gender = Female | Male
  deriving (Eq, Ord, Enum, Show, Read)
```

- the compiler generates the ‘obvious’ code (using a technique similar to generic programming; lecture 7)
- deriving works for parametric types too

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
  deriving (Eq, Ord, Show, Read)
```

Pretty printing

- converting data into textual representation: pretty printing

```
type ShowS = String ⟶ String

class Show a where
  show      :: a ⟶ String
  showsPrec :: Int ⟶ a ⟶ ShowS
  showList  :: [a] ⟶ ShowS

  show x     = showsPrec 0 ""
```

- operator precedences can be taken into account
- for each type we can also decide how to format lists of elements of that type
- you almost always want to say **`deriving (Show)`**

Parsing

- converting textual representation into data

```
type ReadS a = String ⟶ [(a,String)]
class Read a where
  readsPrec :: Int ⟶ ReadS a
  readList  :: ReadS [a]
```

- Read uses "list of successes" technique (more in lecture 13: Parsing)
- Additionally we have

```
read :: Read a ⇒ String ⟶ a
```

- `read:` input string must be completely consumed
- `read.show` should be the identity

Overloading vs. hio-functions (I)

- instead of overloading we can use functions as arguments
- eg

```
elem :: (Eq a) ⟹ a ⟶ [a] ⟶ Bool
elem x [ ]       = False
elem x (y : ys) = x == y || elem x ys
```

- abstract away from Eq

```
elemBy :: (a ⟶ a ⟶ Bool) ⟶ a ⟶ [a] ⟶ Bool
elemBy eq x [ ]       = False
elemBy eq x (y : ys) = x `eq` y || elemBy eq x ys
```

Overloading vs. hio-functions (II)

- instance of Eq for [ ]:

```
instance (Eq a) ⟹ Eq [a] where
    [] == []          = True
    [] == _l          = False
    _l == []          = False
    (x:xs) == (y:ys)  = x == y && xs == ys
```

- eliminating/abstracting away from Eq

```
eqList :: (a ⟶ a ⟶ Bool) ⟶ [a] ⟶ [a] ⟶ Bool
eqList eq []     []     = True
eqList eq []     _l     = False
eqList eq _l     []     = False
eqList eq (x:xs) (y:ys) = x `eq` y && eqList eq xs ys
```

Overloading vs. hio-functions (III)

- consider type

```
data Gtree a = Branch a [Gtree a]
```

- instance of Eq:

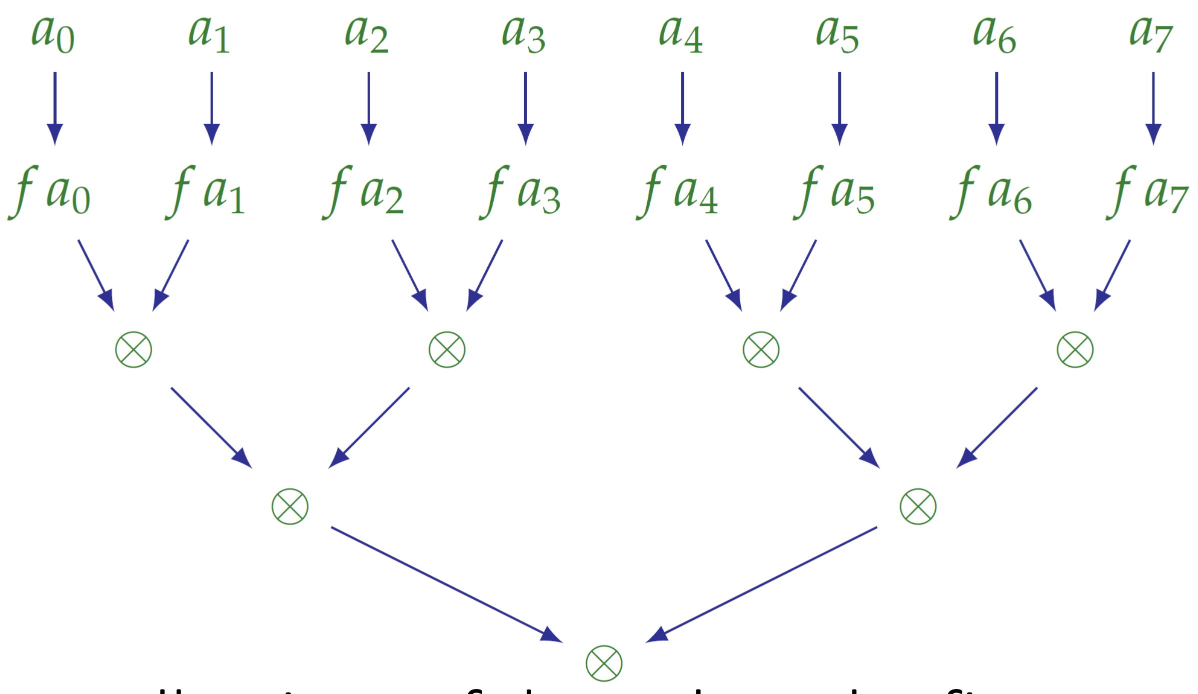
```
instance (Eq a) ⇒ Eq (Gtree a) where
   Branch e1 trs1 == Branch e2 trs2 = e1 == e2 && trs1 == trs2
```

- eliminating overloading

```
eqGtree :: (a ⟶ a ⟶ Bool) ⟶ Gtree a ⟶ Gtree a ⟶ Bool
eqGtree eq (Branch e1 trs1) (Branch e2 trs2)
   = e1 `eq` e2 && eqList (eqGtree eq) trs1 trs2
```

Case study: Google's map-reduce (thanks to Hinze)

- let's explore Google's map-reduce API
- *idea:* do something uniform across a huge collection of data (in parallel) and then
- combine the results



- if we use lists to model huge collections of data, then the first step is simply an application of `map`

Monoids

- it remains to define a reduction: collapsing a list of values into a single value
- *minimal assumptions:* the operation $\otimes$ is associative and has a unit element
- thus map-reduce builds on so-called *monoids.*

Monoids (from Wikipedia)

- *Suppose that S is a set and $\otimes$ is some binary operation $S \times S \to S$, then S with $\otimes$ is a monoid if it satisfies the following two axioms:*
  - *Associativity*
    - *For all a, b and c in S, the equation $(a \otimes b) \otimes c = a \otimes (b \otimes c)$ holds.*
  - *Identity element*
    - *There exists an element e in S such that for every element a in S, the equations $e \otimes a = a \otimes e = a$ hold.*

detour: Operator associativity and precedence in Haskell

- Operator associativity:
  property of an operator describing how multiple usages of the same operator are grouped in the absence of parentheses.
  - eg `1/2/3 = (1/2)/3` (and not `1/(2/3)`)
- Operator Precedence:
  property of an operator describing how usages of different operators are grouped in the absence of parentheses.
  - eg `1+2*3 = 1+(2*3)` (and not `(1+2)*3`)
- When you introduce a new operator, you can add a *fixity declaration* to it, in which you indicate the associativity and the precedence.
  - eg `infixl 7 *, /, `quot`, `rem`, `div`, `mod``

Monoids in Haskell

- why not define a class for monoids?

```
class Monoid m where
  mempty   :: m
  (<>)     :: m ⟶ m ⟶ m

  mconcat :: [m] ⟶ m
  mconcat :: foldr (<>) mempty
```

- (in Haskell the monoid operation `<>` is defined in a separate class `Semigroup` extended by `Monoid`)
- Monoid laws: the operation `<>` should be associative with **`mempty`** as its unit element

```
x <> mempty = x = mempty <> x
(x <> y) <>  z = x <> (y <> z)
```

Reduce

- collapsing a list of values into a single value:

```
reduce :: (Monoid m) ⟹ [m] ⟶ m
reduce = mconcat
```

- other possibility

```
reduce = foldl (<>) mempty
```

- the art of map-reduce is to find a suitable monoid

Examples of monoids: lists

- lists form a monoid

```
instance Monoid [a] where
  mempty = [ ]
  (<>)   = (++)
```

- (proof obligation: ++ is associative with [ ] as its unit)
- for lists, `reduce` amounts to `concat`

```
reduce [[4,7],[ ],[1],[1]] = [4,7,1,1]
```

Examples of monoids: integers

- problem: `Int` gives rise to several monoids—which one to pick?
  - Remember: only one instance per type possible.
- solution: we introduce a new type for each instance eg

```
newtype Additive = Sum { fromSum :: Int }
    deriving (Show)
instance Monoid Additive where
    mempty = Sum 0
    x <> y = Sum (fromSum x + fromSum y)
```

- the underlying `Int` value is extracted using `fromSum`
- **`newtype`** is like **`type`** in that a new type is defined in terms of an old one (no run-time overhead)
- **`newtype`** is like **`data`** in that the type defined is unequal to all other types
- we cannot say `4711 + Sum 0815`

Examples of monoids: integers— continued

- another instance

```
newtype Multiplicative = Product {  fromProduct :: Int  }
    deriving (Show)
instance Monoid Multiplicative where
    mempty = Product 1
    x <> y = Product (fromProduct x * fromProduct y)
```

- example applications

```
>>> reduce [Sum i | i←[1..100]]
Sum { fromSum = 5050 }
>>> reduce [Product i | i←[1..10]]
Product { fromProduct = 3628800 }
>>> fromProduct (reduce [Product i | i←[1..10]])
3628800
```

Examples of monoids: bounded orders

- bounded orders also form a monoid

```
newtype Maximum a = Max { fromMax :: a }
    deriving (Show)
instance (Ord a,Bounded a) ⇒ Monoid (Maximum a) where
    mempty          = Max minBound
    Max x <> Max y = Max (x `max` y)
```

- application: ranking web pages

```
type Rank = Int
rank :: String ⟶ String ⟶ Rank -- Google’s secret
best :: String ⟶ [String] ⟶ Maximum Rank
best s = reduce . map (\x ⟶ Max (rank s x))
```

search string

page

Threading information around

- of course, we usually want to see the highest-ranked web page (best only returns the maximum rank)
- idea: pair the web pages with their rank

```
data WithString key = With key String
```

- ranking web pages

```
best' :: String → [String] → Maximum (WithString Rank)
best' s = reduce . map (\x → Max (With (rank s x) x))
```

- get this working, the following instances are needed

```
instance Eq key ⇒ Eq (WithString key) where
  With k1 _ == With k2 _ = k1 == k2
instance Ord key ⇒ Ord (WithString key) where
  With k1 _ ≤ With k2 _ = k1 ≤ k2
instance Bounded key ⇒ Bounded (WithString key) where
  minBound = With minBound "<<404 Error>>"
  maxBound = With maxBound "<<404 Error>>"
```

Sequential evaluation of polynomials

- a polynomial can be represented by a list of coefficients eg

```
p :: Integer ⟶ Integer
p (x) = 4 + 7*x + 1*x² + 1*x³
```

- is represented by `[4,7,1,1]`
- sequential evaluation of polynomials. Horner's rule :

```
p (x) = 4 + x*(7 + x*(1 + x*(1 + x*0)))
```

- can be captured as a fold:

```
evaluate :: Integer ⟶ [Integer] ⟶ Integer
evaluate x = foldr (\c v ⟶ c + x*v) 0
```

- eg `evaluate 2 [4,7,1,1]` yields `p(2) = 30`

parallel evaluation of polynomials

- Idea: `p (x) = (4 + 7*x) +` x^2`*(1 + 1*x)`
- eg `p (2) = (4 + 7*2) +` 2^2`*(1 + 1*2) = 18 + 4*3 = 30`
- we have to maintain two pieces of information: x^{d+1} (d: the degree of the polynomial) and the actual value of the polynomial

```
data Poly = Poly Integer Integer

instance Monoid Poly where

    mempty                  = Poly 1 0

    Poly x u <> Poly y v = Poly (x*y) (u + x*v)

evaluate :: Integer ⟶ [Integer] ⟶ Poly

evaluate x = reduce.map (\a ⟶ Poly x a)
```

- eg `evaluate 2 [4,7,1,1]` yields `Poly (`2^4`) (p (2)) = Poly 16 30`

Probability distributions

- discrete probability distribution (probability mass function)

```
type Prob = Rational
newtype Dist event = D { fromD :: [(event,Prob)]}
```

- invariant: probabilities of a distribution dist sum up to 1

```
sum [p | (e,p) ← fromD dist] == 1
```

- (ideally, each event occurs exactly once; exercise: define

```
norm :: (Ord event) ⇒ Dist event → Dist event)
```

- uniform distribution

```
uniform :: [event] → Dist event
uniform es = D [(e, 1 % n) | e ← es] where
   n = genericLength es
```

Combining distributions: The probability monoid

- Monoid instance

```
instance (Monoid event) ⟹ Monoid (Dist event) where
  mempty        = D [(mempty,1)]
  D d1 <> D d2 = D [(e1 <> e2,p1*p2) | (e1,p1)⟵d1, (e2,p2)⟵d2]
```

- combining event-probability pairs:
  - probabilities are multiplied
  - events are ‘mappended’
- (is the invariant satisfied?)

Probability distributions: examples

- a fair die

```
die :: Dist Integer
die = uniform [1 .. 6]
```

- changing event types

```
mapDist :: (a → b) → Dist a → Dist b
mapDist f (D d) = D [(f e,p) | (e,p) ← d ]
```

- playing monopoly: two dice, pips of the dice are added
  - we use de additive monoid to combine events

```
dieM :: Dist Additive
dieM = mapDist Sum die
```

- rolling two (Monopoly) dice

```
>>> reduce [dieM,dieM]
[(2,1 % 36),(3,1 % 36),(4,1 % 36),(5,1 % 36),(6,1 % 36),
 (7,1 % 36),…]
>>> norm it
[(2,1 % 36),(3,1 % 18),(4,1 % 12),(5,1 % 9),(6,5 % 36),(7,1 % 6),
 (8,5 % 36),(9,1 % 9),(10,1 % 12),(11,1 % 18),(12,1 % 36)]
```

Playing Yahtzee

Since there are 300 ways to roll a **full house** in a single roll and there are 7776 rolls of five dice possible, the **probability of rolling** a **full house** is 300/7776, which is close to 1/26 and 3.85%. ... This is 50 times more likely than **rolling** a **Yahtzee** in a single roll. Jan 31, 2019

www.thoughtco.com › Statistics › Probability & Games

The Probability of Rolling a Full House in Yahtzee? - ThoughtCo

Playing Yahtzee -- representation

- Now we use the list monoid to combine events

```
dieY :: Dist [Integer]
dieY = mapDist (:[]) die
```

- multiple dice

```
dice :: Int ⟶ [Dist [Integer]]
dice n = replicate n dieY
```

- rolling dice

```
rollY n = reduce (dice n)
```

- sum of probabilities

```
(??) :: (a ⟶ Bool) ⟶ Dist a ⟶ Prob
ev ?? dist = sum [ p | (v,p) ⟵ fromD dist, ev v]
```

- probability of getting a Yahtzee in a single roll

```
yahtzee :: [Integer] ⟶ Bool
yahtzee roll = length (group roll) == 1
⋙ yahtzee ?? rollY 5
1 % 1296
```

Abstraction, abstraction, abstraction

- question: what are the three most important concepts in programming?
- answer: abstraction, abstraction, abstraction!
- type classes allow you to capture commonalities across datatypes
- classes are most useful if the type uniquely determines the instance (example: functor (introduced later), counterexample: monoid)