CSEP505: Programming Languages
Lecture 9: Haskell Typeclasses and Monads;

Dan Grossman
Autumn 2016
Acknowledgments

• Slide content *liberally* appropriated with permission from Kathleen Fisher, Tufts University

• She in turn acknowledges Simon Peyton Jones, Microsoft Research, Cambridge “for some of these slides”

• And then I probably introduced errors and weaknesses as I changed them [and added the material on the Monad type-class and wrote the accompanying code file]…

• *Also note: This lecture relies heavily on lec9.hs*

• Then onto OOP as a separate topic (awks not applicable)
Generics vs. Overloading [again]

- **Parametric polymorphism:**
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - If \( f :: t \to t \) then \( f :: \text{Int} \to \text{Int}, f :: \text{Bool} \to \text{Bool}, \ldots \)

- **Overloading**
  - Single symbol may refer to *more than one* algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - \( + \) has types \( \text{Int} \to \text{Int} \to \text{Int} \) and \( \text{Float} \to \text{Float} \to \text{Float}, \text{but not} \ t \to t \to t \text{ for arbitrary } t \)
Why overloading?

Many useful functions are not parametric

- Can `member` work for any list type?
  ```haskell
  member :: [a] -> a -> Bool
  ``
  No! Only for types `a` that support equality

- Can `sort` work for any list type?
  ```haskell
  sort :: [a] -> [a]
  ``
  No! Only for types `a` that support ordering

- Can `serialize` work for any type?
  ```haskell
  serialize :: a -> String
  ``
  No! Only for types `a` that support ordering
How you do this in OCaml/SML

The general always-works approach is have callers pass function(s) to perform the operations:

```ocaml
member :: (a -> a -> Bool) -> [a] -> a -> Bool
member _ [] _ = False
member eqFun (x:xs) v = eqFun x v 
  || member eqFun xs v
```

Works fine but:

- A pain to thread the function(s) everywhere
- End up wanting a record of functions, a “dictionary”
- Now have to thread right dictionaries to right places
- Types get a little messier?
Part 1 of lec9.hs does “explicit dictionary passing”
- Works fine in Haskell and would work fine in OCaml too
- Lets us use write “generic” algorithms provided caller gives a
dictionary (e.g., `double` or `sumOfSquares`)
- Can even use dictionaries to build other dictionaries (e.g.,
  `complexDictMaker`)
- Funny dictionaries can produce funny results (e.g.,
  `fortyTwo`)

Lecture 9  CSE P505 Autumn 2016  Dan Grossman

6
Enter Type Classes

Type-classes are *built-in support* for *implicit* dictionary-passing

- Concise types to describe [records of] overloaded functions
- Sophisticated standard library of type classes for [all the] common purposes
- But nothing “privileged” in the library/language: Users can declare their own type classes (nothing special about `==`, `+`, etc.)
- Interacts well enough with type inference [won’t study the “magic”]

And/but:
- Ends up “taking over the language and standard library”
- Lots of fancy features that are super-useful, but we’ll have time for just a quick exposure beyond the basics
Type Class Design Overview

• [Step 0: Do *not* try to compare these things to OOP classes and such; they are different. Will study OOP next.]

• Step 1: Type class declarations
  – Define a set of [typed] operations and give the set a name
  – Example: The `Eq a` type-class has operations `==` and `/=` both of type `a -> a -> Bool`

• Step 2: Instance declarations
  – Specify the implementations for a particular type
  – Examples: for `Int`, `==` is integer equality, for `String`, `==` is string equality (but *could* have decided case-insensitive)

• Step 3: Qualified types
  – Use qualified types to express that a polymorphic type must be an instance of your type class
  – Example: `member' :: Eq a => [a] -> a -> Bool`
Qualified types

\[
\text{member' :: Eq a => [a] -> a -> Bool}
\]

• *Very roughly* like a bound on the type variable
  – Caller must instantiate type variable with a type that is known to be an instance of the class
  – Callee may assume the type is an instance of the class (so can use the operations)
  – So “fewer” callers type-check and “more” callees type-check

• At run-time, the right dictionary will be *implicitly* passed and used
  – Call-site “knows which dictionary”
  – Calls in callee “use the dictionary”
More Examples

sort :: Ord a => [a] -> [a]
reverse :: [a] -> [a]
square :: Num a => a -> a
squarePair :: (Num a, Num b) => (a,b) -> (a,b)
stringOfMin :: (Ord a, Show a) => [a] -> String
Our own classes and instances

- The class declaration gives names and types to operations
- An instance declaration provides the operations’ implementations

```haskell
class MyNum a where
    plus' :: a -> a -> a
    times' :: a -> a -> a
    neg'   :: a -> a
    zero'  :: a

instance MyNum Int where
    plus' = (+)
    times' = (*)
    neg'   = \x -> -1 * x
    zero'  = 0

instance MyNum Float where
    plus' = (+)
    times' = (*)
    neg'   = \x -> -1.0 * x
    zero'  = 0.0
```
Then use them

- Use qualified types to write algorithms over overloaded operations

```haskell
member' :: Eq a => [a] -> a -> Bool
member' []     v = False
member' (x:xs) v = (==) x v || member' xs v

double' :: MyNum a => a -> a
double' v = (plus' (plus' v v) zero')

sumOfSquares' :: MyNum a => [a] -> a
sumOfSquares' [] = zero'
sumOfSquares' (x:xs) = plus' (times' x x) (sumOfSquares' xs)

i8  = double' 4
f8  = double' 4.0
yes = member' [3,4,5] 4
no  = member' ["hi", "bye"] "foo"
```
Compositionality of functions

- Overloaded functions can be defined using other overloaded functions

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

quadAndFour :: Num a => a -> (a, Int)
quadAndFour x = (square x * square x, square 2)

eg = quadAndFour 3.0 -- (81.0, 4)
```

- `quadAndFour` “doesn’t know” what dictionary it was passed, but it knows which dictionary to pass to each of its calls to `square`
Compositionality of Instances

- Can use qualified instances to build compound instances in terms of simpler ones
- Simple example from standard library:

```haskell
class Eq a where
  (==) :: a -> a -> Bool
instance Eq Int where
  (==) = intEq
    -- intEq primitive equality
instance (Eq a, Eq b) => Eq (a,b) where
  (==) (u,v) (x,y) = (u == x) && (v == y)
instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

- A little more complicated example: see lec9.hs for

  ```haskell
  instance MyNum a =&gt; MyNum (Complex a) ...
  ```
Subclasses

- Can specify “any instance of class \texttt{Foo} must also be an instance of class \texttt{Bar}”
  - Example: \texttt{Ord a} subclass of \texttt{Eq}
  - Example: \texttt{Fractional a} subclass of \texttt{Num}
    - \texttt{(Fractional supports real division and reciprocals)}
- Easy to define:
  ```
  class Eq a => Ord a where -- defines Ord a
  ...
  ```
- An instance must provide everything in the superclass (too)
- Makes a qualified type “provide more”

- This still isn’t OOP classes [we are defining and passing dictionaries separately and with static type resolution]
Default methods

• A class declaration can provide default implementations
  – Including in terms of other implementations
  – Instances can override the default or not
  – Example: not-equal as not of equal
  – Example: >= as > or ==
  – Example: arbitrary result like 42

```
-- Minimal complete definition: (==) or (/=)
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

• This still isn’t OOP classes [we are defining and passing dictionaries separately and with static type resolution]
No, really, it’s not OOP

• Dictionaries and method suites (vtables) are similar

But…

• As we have said:
  – Dictionaries “travel” separately from values
  – Method resolution is static in Haskell, based on types

• Also:
  – Constrains polymorphism, does not introduce subtyping
  – Can add instance declarations for types “retroactively”
  – Dictionary selection can depend on result types:
    \[
    \text{fromInteger :: Num } a \Rightarrow \text{ Integer } \rightarrow a
    \]
Topics to skip

Very useful for practical programming but a bit off our trajectory:

• **deriving** to get automatic instances from data definitions
  – Example: Show a tree

• Support for numeric literals using the `fromInteger` operation that lets you use 0, 3, 79, etc. in any instance of `Num`

• Interaction with type inference
  – Mostly “works fine”
  – Various details, including do not reuse operation names across classes in same scope
Now constructor classes

• Recall:
  – Int, [Int], Complex Int, Bool, Int -> Int, etc. are types
  – [-], Tree, etc. are type constructors (given a type, produce a type)

• We can define type classes for type constructors
  – Nothing really “new” here
  – Harder to read at first, but “arity” of the constructor is inferred from use in class declaration

• See Part 3 of lec9.hs for instances and uses of this example:

```haskell
class HasMap g where
  map' :: (a -> b) -> g a -> g b
```
Now back to monad

- **Monad** is a constructor class just like `HasMap (!!)`
  - “Required” operations are `>>=` and `return`
  - Default operations for things like `>>`
  - `IO` is “just” one “special” instance of monad
  - There are *many* useful instances of monad
  - Any instance of monad can use do-notation since it’s just sugar for calls to `>>=`

- See Parts 4, 5, and 6 of lec9.hs to blow your mind 🤓
Summary of all that (!) 😊

- “Part 4”
  - `Monad` is a constructor typeclass
  - `Instance Monad Maybe'` gives intuitive definitions to `>>=` and `return`
  - do-notation for “maybe” can be much less painful than life without it
- “Part 5”
  - Naturally, can write code generic over “which monad instance”
  - So can reuse combinators like
    `sequence :: Monad m => [m a] -> m [a]`
- “Part 6”
  - State monad `definition` is purely functional but looks-and-feels like imperative programming when using it
Other cheats

• So type classes seem to work pretty well
  – Haskell has, over time, committed to them ever-more fully

• Without them, you can:
  – Do explicit dictionary passing
  – “Cheat” in various ways:
    • SML: special support for `Eq` and nothing else
      – Oh also +, *, etc. for `int` and `float`
    • OCaml: cheat on key functions like `hash` and `=` being allegedly fully polymorphic but can fail at runtime and/or violate abstractions

• C++: OOP or code duplication, neither of which is the same??