

Advances in Programming Languages

APL8: Multiparameter Type Classes, Constructor Classes

Ian Stark

School of Informatics
The University of Edinburgh

Thursday 4 February
Semester 2 Week 4



Some Types in Haskell

This is the second of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes
- Multiparameter type classes, constructor classes,
- Monads and interaction with the outside world
- Encapsulating stateful computation

Some Types in Haskell

This is the second of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes
- Multiparameter type classes, constructor classes,
- Monads and interaction with the outside world
- Encapsulating stateful computation

Outline

- 1 Type Classes
- 2 People
- 3 Multiparameter Type Classes
- 4 Constructor Classes
- 5 Others
- 6 Closing

Parametric Polymorphism

Haskell makes extensive use of parametric polymorphism

```
reverse :: [a] -> [a]
```

```
> reverse [1,2,3]
```

```
[3,2,1]
```

```
> reverse [True,False]
```

```
[False,True]
```

```
> reverse "Edinburgh"
```

```
"hgrubnidE"
```

The polymorphic function `reverse` here must use nothing at all specific about the type 'a' being handled.

Qualified Polymorphism

The introduction of type classes refine this so functions can make assumptions about the operations available on values.

```
revShow :: Show a => [a] -> [String]
revShow = reverse . map show
```

```
> revShow [1,2,3]
["3","2","1"]
```

```
> revShow [1.2,3.4,5.6]
["5.6","3.4","1.2"]
```

```
> revShow "abc"
["'c'", "'b'", "'a'"]
```

This resembles method dispatch and OO-style polymorphism, but they are not the same: although different lists passed to `revShow` may contain different types, each list must carry only elements of a single type.

Homogeneous collections, not heterogeneous

Qualified Polymorphism

The introduction of type classes refine this so functions can make assumptions about the operations available on values.

```
revShow :: Show a => [a] -> [String]
revShow = reverse . map show
```

```
> revShow [1,2,3]
["3","2","1"]
```

```
> revShow [1.2,3.4,5.6]
["5.6","3.4","1.2"]
```

```
> revShow "abc"
["'c'", "'b'", "'a'"]
```

On the other hand, this does allow bulk operations like

```
maximum, minimum :: (Ord a) => [a] -> a
```

which caused such problems for the Java type system.

Multiple Classes

Polymorphic values may use more than one class qualification:

```
showMax :: (Ord a, Show a) => [a] -> String  
showMax = show . maximum
```

```
> showMax [1,2,3]  
"3"
```

```
> showMax "Edinburgh"  
"u"
```

```
> showMax ["Advances", "Programming", "Languages"]  
"Programming"
```


Subclassing

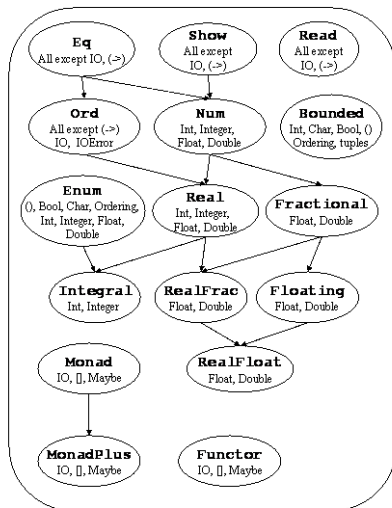
Adding qualifications to class declarations introduces subclassing:

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

So every `Ord` type is also an `Eq` type: but note that this is *subclassing* not *subtyping*.

Multiway Subclassing

Classes may depend on more than one superclass; including diamonds of related classes.



Nested Instances

```
class Reportable a where
```

```
  report :: a -> String
```

```
instance Reportable Integer where
```

```
  report i = show i
```

```
instance Reportable Char where
```

```
  report c = [c]
```

```
instance Reportable a => Reportable [a] where
```

```
  report xs = "[" ++ intercalate "," (map report xs) ++ "]"
```

```
instance (Reportable a, Reportable b) => Reportable (a,b) where
```

```
  report (x,y) = "(" ++ report x ++ "," ++ report y ++ ")"
```

```
> report [(1,'a'),(2,'b')]
```

```
"[(1,a),(2,b)]"
```

Building concrete instances like `Reportable [(a,b)]` may require some search by the compiler.

(instance declarations \approx mini logic programming)

Code Inheritance

Classes declarations may carry code that is inherited by all types of that class.

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

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

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

Instances of `Eq` may provide `==`, or `/=`, or both.

Types may draw code from multiple classes, as with OO *traits* and *mixins*.

Polymorphic qualification need not be determined by a single “primary” value.

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

left $x = \text{"Before"} ++ x$

right $y = y ++ [3,4,5]$

both $x y = (x ++ y) :: [\text{Float}]$

This answers the “binary method problem” in a similar way to OO multiple dispatch.

Typing by Result

Resolving which instance of a method to use may even be done without any arguments at all:

```
maxBound :: (Bounded a) => a
```

Instance by result is used to overload numeric constants. The definition

```
raise x = x + 5
```

is expanded by the compiler, with dictionary passing, to:

```
raise d x = (d (+)) x (d fromInteger 5)
```

Hence the user-written `raise` gets all the flexibility of built-in `5`.

Although in some cases, the slowest part of computing $(x+1)$ may be the `1`.

Instances Anywhere

Qualified polymorphic functions may even use instances defined later on:

```
import Complex  
i = sqrt (-1) :: Complex Float  
raise i
```

Instance declarations can be at class declaration; or type declaration; or anywhere else.

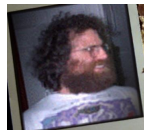
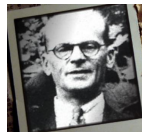
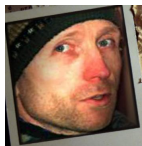
This can retrospectively hook new types up to existing libraries, or extend existing types by bringing into new classes.

In each case, a compiler can use dictionary-passing translation to a class-free lower language, which is then open to all optimisations available for general programming.

Outline

- 1 Type Classes
- 2 People**
- 3 Multiparameter Type Classes
- 4 Constructor Classes
- 5 Others
- 6 Closing

Programming Language Inventor or Serial Killer?



<http://www.malevole.com/mv/misc/killerquiz/>

Outline

- 1 Type Classes
- 2 People
- 3 Multiparameter Type Classes**
- 4 Constructor Classes
- 5 Others
- 6 Closing

Multiparameter Type Classes

In Haskell '98 a class can only qualify a single type.

```
class Reportable a where  
  report :: a -> String
```

Some implementations of Haskell extend this to *multiparameter* type classes that can relate two or more types.



M. P. Jones.

Type classes with functional dependencies.

In *Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming, ESOP 2000*, Lecture Notes in Computer Science 1782, pages 230–244. Springer-Verlag, 2000.

Multiparameter Type Class Example

For example, we might indicate that one type is a collection of elements from another:

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

We can then use different collection implementations for particular kinds of element:

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

Multiparameter Type Class Example

For example, we might indicate that one type is a collection of elements from another:

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

Unfortunately, there is a problem of ambiguity.

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

Which element type should this be collecting?

```
(\x y -> insert x . insert y) :: a -> b -> s -> s
```

Is 's' a collection of 'a' values or 'b' values? Could it be both?

Multiparameter Type Class Example

For example, we might indicate that one type is a collection of elements from another:

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

In practice, the type of elements 'e' is determined by the collection type 's'. We make this explicit with a *functional dependency*

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

This guarantees (and enforces) that for each 's' there can be at most one 'e' with `Collects s e`.

Multiparameter Type Class Example

For example, we might indicate that one type is a collection of elements from another:

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

Multiparameter type classes can give yet more overloading:

```
class Multiply a b c | a b -> c where  
  mult :: a -> b -> c
```

```
instance Num n => Multiply n n n where mult = (*)
```

```
instance Num n => Multiply n (Vector n) (Vector n) where mult = ...
```

```
instance Num n => Multiply n (Matrix n) (Matrix n) where mult = ...
```

```
instance Num n => Multiply (Matrix n) (Matrix n) (Matrix n) where ...
```

Outline

- 1 Type Classes
- 2 People
- 3 Multiparameter Type Classes
- 4 Constructor Classes**
- 5 Others
- 6 Closing

Kinds and Constructors

In Haskell `Integer` is a type, and `Maybe` is a type *constructor* — unlike types, constructors have no values.

Types and constructors are themselves classified by *kinds*. Every type has kind `*`, and constructors have kinds built using `*` and `->`.

```
Integer, Int, Float :: *           [] :: * -> *
Maybe :: * -> *                   (,) :: * -> * -> *
                                   (,,) :: * -> * -> * -> *
```

It is even possible to have higher kinds:

```
data TreeOf f a = Leaf a | Node (f (TreeOf f a))
Node [Leaf True,Leaf False] :: TreeOf [] Bool
TreeOf :: (*->*) -> * -> *
```

Classes for Constructors

Not only do constructors have kinds, they can also belong to classes within them.

```
class Functor f where           -- Type constructor  $f :: * \rightarrow *$   
  fmap :: (a  $\rightarrow$  b)  $\rightarrow$  f a  $\rightarrow$  f b
```

```
instance Functor [] where  
  fmap = map
```

```
instance Functor Maybe where  
  fmap p Nothing = Nothing  
  fmap p (Just x) = Just (p x)
```

```
instance Functor f  $\Rightarrow$  Functor (TreeOf f) where  
  fmap p (Leaf a) = Leaf (p a)  
  fmap p (Node n) = Node (fmap p n)
```

Outline

- 1 Type Classes
- 2 People
- 3 Multiparameter Type Classes
- 4 Constructor Classes
- 5 Others**
- 6 Closing

And it goes on...

Haskell has an expanding cornucopia of type-driven language features. Many are implemented in GHC, if only experimentally.

- Explicit kinds `1 :: (Int :: *)`
- Explicit for-all `f :: forall a.(a -> a -> a)`
- Rank-2 polymorphism, and higher `g :: (forall a.(a->[a])) -> Int`
- Existential types `xs :: exists a.(a,a->Bool,a->String)`
- GADT: Generalized Algebraic Datatypes
- ...

Outline

- 1 Type Classes
- 2 People
- 3 Multiparameter Type Classes
- 4 Constructor Classes
- 5 Others
- 6 Closing**

Homework

Read the following paper on multiparameter type classes.



M. P. Jones.

Type classes with functional dependencies.

In Programming Languages and Systems: Proceedings of the 9th European Symposium on Programming, ESOP 2000, Lecture Notes in Computer Science 1782, pages 230–244. Springer-Verlag, 2000.

Further Reading



Simon Peyton Jones (Editor)

Haskell 98 Language and Libraries: The Revised Report
Journal of Functional Programming 13(1):7–255.
<http://www.haskell.org/onlinereport/>



The GHC Team

The Glorious Glasgow Haskell Compilation System User's Guide
http://www.haskell.org/ghc/docs/latest/html/users_guide/index.html



Haskell' ("Haskell Prime")

<http://hackage.haskell.org/trac/haskell-prime/>



Simon Marlow

Announcing Haskell 2010
Haskell Mailing List, November 2009.

Further References



Mark Jones

A system of constructor classes: overloading and implicit higher-order polymorphism

In Functional Programming and Computer Architecture: Proceedings of FPCA '93, pages 52–61. ACM Press, 1993.



James Cheney and Ralf Hinze

First-class phantom types

Technical Report TR2003-1901, Cornell University Faculty of Computing and Information Science