Advances in Programming Languages

APL7: Polymorphism from Types to Kinds and Beyond

Ian Stark

School of Informatics
The University of Edinburgh

Tuesday 12 October
Semester 1 Week 4

Foreword

Some Types in Haskell

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

- Type classes
- Polymorphism, kinds and constructor classes
- Monads and interaction with the outside world

Foreword

Some Types in Haskell

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

- Type classes
- Polymorphism, kinds and constructor classes
- Monads and interaction with the outside world

Summary

Object-oriented languages offer *polymorphism* and run different code for different objects. For class-based OO languages like Java this *ad-hoc* polymorphism can require complex resolution of method invocation. In addition, *generic* code acts on *parameterized types* to give *parametric polymorphism*.

Functional languages use parametric polymorphism extensively; Haskell *type classes* extend it with *qualified types* to select different code for different types. This gives ad-hoc polymorphism, overloading, inheritance, multiple dispatch, and more.

Types and type constructors are grouped by *kind*, and even *higher kinds*; these too can be qualified into *constructor classes* like Functor.

Outline

Flavours of Polymorphism

Ad-hoc Polymorphism

Classic object-oriented polymorphism: invoke method a.draw() and get whatever code is assigned to the target object a or its class.

Implementing this requires some attention to the *dispatch* of methods to determine the code finally executed.

Parametric Polymorphism

Operations that act similarly on arguments of all types: sorting a list, applying a function to every element of a collection.

Closely tied to *parameterized types* and in OO languages known as *generics*, as with LinkedList<String> or Map<K,V>.

What Decides Which Method in Java?

In a class-based object-oriented programming language like Java, with overloading, inheritance, interfaces and abstract classes, it can be quite complex to resolve which method implementation is actually invoked on execution.

```
Appointment booking = diary.lookup(date,time,place);
```

What contributes to the dispatch decision as to which lookup method implementation executes at runtime?

What Decides Which Method in Java?

```
Appointment booking = diary.lookup(date,time,place);
```

What determines the lookup code actually executed?

- The name of the method?
- The compile-time class of the diary variable?
- The run-time class of the object in the diary variable?
- The number of parameters listed?
- The compile-time class of the parameters date, time, place?
- The run-time class of the objects passed as arguments date, time, place?
- The class of the result booking?
- The subsequent operations on the result booking?
- Something more?

Java makes certain choices here; other languages make different ones.

Parametric Polymorphism in Haskell

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

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'"]
```

These have a *dictionary-passing* implementation, which is accessible to standard compiler optimisations.

Qualified Polymorphism

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

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

These look a little like method dispatch and OO-style ad-hoc 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

This *qualified polymorphism* deals well with issues like

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

which caused such problems for the Java type system.

Outline

Multiple Classes

Polymorphic values may use more than one 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

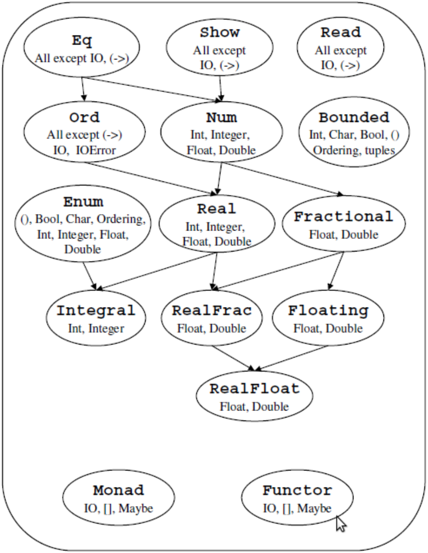
Adding qualifications to class declarations introduces subclassing:

```
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 sub*classing* not sub*typing*.

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]
```

Nested Instances

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

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,'p'),(2,'q')]
"[(1,p),(2,q)]"
```

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

Multimethods

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

```
(++) :: [a] -> [a] -> [a]

left   x = "Before" ++ x
right y = y ++ [3,4,5]
both x y = (x ++ y) :: [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

```
bump x = x + 5                    -- 5 :: (Num t) => t
```

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

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

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

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

Outline

Types and Constructors

In Haskell every value has a type.

```
42 :: Integer
pi :: Double
"Hello, world!" :: String
1/3 :: Rational
```

Some types are built from other types.

```
Just pi :: Maybe Float
Nothing :: Maybe Float
Left 4 :: Either Int Float
Right 1.2 :: Either Int Float
[True] :: [Bool]
```

Kinds and Higher Kinds

In Haskell Integer is a type, while Maybe and Either are type *constructors* — 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 :: * -> *
  fmap :: (a -> b) -> f a -> 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 => Functor (TreeOf f) where
  fmap p (Leaf a) = Leaf (p a)
  fmap p (Node n) = Node (fmap p n)
```

And it goes on...

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

- Multiparameter type classes **class** Collects s e
- 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

Summary

Object-oriented languages offer *polymorphism* and run different code for different objects. For class-based OO languages like Java this *ad-hoc* polymorphism can require complex resolution of method invocation. In addition, *generic* code acts on *parameterized types* to give *parametric polymorphism*.

Functional languages use parametric polymorphism extensively; Haskell *type classes* extend it with *qualified types* to select different code for different types. This gives ad-hoc polymorphism, overloading, inheritance, multiple dispatch, and more.

Types and type constructors are grouped by *kind*, and even *higher kinds*; these too can be qualified into *constructor classes* like Functor.

Reading

Homework

Friday's lecture will be about monads and I/O in Haskell. Read the following set of slides on types and effects.

Simon Peyton Jones
Caging the Effects Monster: The Next Big Challenge
Slides from talks at QCon 2008 and ACCU '08
Available online from
http://research.microsoft.com/en-us/people/simonpj/

Further References

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha
The Java Language Specification, Third Edition
Addison Wesley, 2005

Bryan O'Sullivan, Don Stewart, and John Goerzen
Real World Haskell
O'Reilly Media, 2008
http://www.realworldhaskell.org/

To find out about Java method invocation, see §15.12 of the language specification.

For more on type classes, see Chapter 6 of Real World Haskell, and in particular the sections on numeric types and functions.

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