Classes, Jim, but not as we know them

Simon Peyton Jones (Microsoft Research)
Most new programming languages

The quick death
Successful research languages

The slow death

Practitioners

Geeks
The complete absence of death
The committee language
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)

“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)

The second life?
Language popularity: how much language X is used.

This is a chart showing combined results from all data sets, listed individually below:

- C
- Java
- C++
- PHP
- JavaScript
- Python
- C#
- Perl
- SQL
- Ruby
- Shell
- Visual Basic
- Assembly
- ActionScript
- Objective C
- Lisp
- Delphi
- Pascal
- Scheme
- Haskell
- Tcl
- Fortran
- Ada
- Lua
- ColdFusion
- Cobol
- Erlang
- D

langpop.com Aug 2013
Language popularity
how much language X is talked about
Ideas

- Purely functional (immutable values)
- Controlling effects (monads)
- Laziness
- Concurrency and parallelism
- Domain specific embedded languages
- Crazy type laboratory
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
Haskell in one slide

Type signature

Higher order

Polymorphism (works for any type a)

filter :: (a->Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

Functions defined by pattern matching

Guards distinguish sub-cases

f x y rather than f(x,y)
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x = x : filter p xs
  | otherwise = filter p xs

data Bool = False | True
data [a] = [] | a:[a]
Test for equality

- Can this really work **FOR ANY** type \( a \)?
- E.g. what about functions?

member negate [increment, \( x.0-x \), negate]
Similar problems

- sort :: [a] -> [a]
- (+) :: a -> a -> a
- show :: a -> String
- serialise :: a -> BitString
- hash :: a -> Int
Unsatisfactory solutions

- **Local choice**
  - Write \((a + b)\) to mean \((a \ `plusFloat` b)\) or \((a \ `plusInt` b)\) depending on type of \(a, b\)
  - Loss of abstraction; eg member is monomorphic

- **Provide equality, serialisation for everything, with runtime error for (say) functions**
  - Not extensible: just a baked-in solution for certain baked-in functions
  - Run-time errors
Similarly:

\[
\begin{align*}
\text{square} & : \text{Num } a \Rightarrow a \to a \\
\text{square } x & = x \times x \\
\text{sort} & : \text{Ord } a \Rightarrow [a] \to [a] \\
\text{serialise} & : \text{Show } a \Rightarrow a \to \text{String} \\
\text{member} & : \text{Eq } a \Rightarrow a \to [a] \to \text{Bool}
\end{align*}
\]
Type classes

\[
\text{square :: Num } n \Rightarrow n \rightarrow n \\
\text{square } x = x \times x
\]

\[
\text{class } \text{Num } a \text{ where} \\
\quad (+) :: a \rightarrow a \rightarrow a \\
\quad (*) :: a \rightarrow a \rightarrow a \\
\quad \text{negate} :: a \rightarrow a \\
\quad \ldots \text{etc..}
\]

\[
\text{instance } \text{Num } \text{Int}\text{ where} \\
\quad a + b = \text{plusInt} a b \\
\quad a \times b = \text{mulInt} a b \\
\quad \text{negate } a = \text{negInt} a \\
\quad \ldots \text{etc..}
\]

FORGET all you know about OO classes!

The class declaration says what the Num operations are.

An instance declaration for a type T says how the Num operations are implemented on T's.

Works for any type 'n' that supports the Num operations.

plusInt :: Int -> Int -> Int
mulInt :: Int -> Int -> Int
e tc, defined as primitives
How type classes work

When you write this...

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

...the compiler generates this

```haskell
square :: Num n -> n -> n
square d x = (*) d x x
```

The “Num n =>” turns into an extra value argument to the function. It is a value of data type Num n.

A value of type (Num T) is a vector (vtable) of the Num operations for type T.
How type classes work

When you write this...

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

...the compiler generates this

```haskell
square :: Num n -> n -> n
square d x = (*) d x x
```

The class decl translates to:
- A data type decl for Num
- A selector function for each class operation

A value of type (Num T) is a vector of the Num operations for type T
How type classes work

When you write this...

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

...the compiler generates this

```haskell
square :: Num n -> n -> n
square d x = (*) d x x
```

```haskell
instance Num Int where
  a + b  = plusInt a b
  a * b  = mulInt a b
  negate a = negInt a
  ...etc..
```

```haskell
dNumInt :: Num Int
dNumInt = MkNum plusInt
mulInt
negInt
... 
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num T) is a vector of the Num operations for type T
How type classes work

When you write this...

\[
\begin{align*}
  f :: \text{Int} & \rightarrow \text{Int} \\
  f \ x & = \text{negate} \ (\text{square} \ x)
\end{align*}
\]

...the compiler generates this

\[
\begin{align*}
  f :: \text{Int} & \rightarrow \text{Int} \\
  f \ x & = \text{negate} \ \text{dNumInt} \\
  & \ (\text{square} \ \text{dNumInt} \ x)
\end{align*}
\]

instance Num Int where
\[
\begin{align*}
  a + b & = \text{plusInt} \ a \ b \\
  a * b & = \text{mulInt} \ a \ b \\
  \text{negate} \ a & = \text{negInt} \ a
\end{align*}
\]

...etc..

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num T) is a vector of the Num operations for type T
All this scales up nicely

- You can build big overloaded functions by calling smaller overloaded functions

```haskell
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n => n -> n -> n
sumSq d x y = (+) d (square d x) (square d y)
```

Extract addition operation from d
Pass on d to square
All this scales up nicely

- You can build big **instances** by building on smaller **instances**

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

instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False

data Eq = MkEq (a->a->Bool)
  (==) (MkEq eq) = eq

deEqList :: Eq a -> Eq [a]
deEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```
Overloaded constants

class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ....
inc :: Num a => a -> a
inc x = x + 1

Even literals are overloaded

"1" means "fromInteger 1"

inc :: Num a -> a -> a
inc d x = (+) d x (fromInteger d 1)
Type classes have proved extraordinarily convenient in practice

- Equality, ordering, serialisation
- Numerical operations. Even numeric constants are overloaded
- Monadic operations

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- And on and on....time-varying values, pretty-printing, collections, reflection, generic programming, marshalling, monad transformers....

Note the higher-kindled type variable, m
Quickcheck

```haskell
propRev :: [Int] -> Bool
propRev xs = reverse (reverse xs) == xs

propRevApp :: [Int] -> [Int] -> Bool
propRevApp xs ys = reverse (xs++ys) == reverse ys ++ reverse xs
```

Quickcheck (which is just a Haskell 98 library)
- Works out how many arguments
- Generates suitable test data
- Runs tests

```
ghci> quickCheck propRev
OK: passed 100 tests

ghci> quickCheck propRevApp
OK: passed 100 tests
```
quickCheck :: Testable a => a -> IO ()

class Testable a where
    test :: a -> RandSupply -> Bool

class Arbitrary a where
    arby :: RandSupply -> a

instance Testable Bool where
    test b r = b

instance (Arbitrary a, Testable b) => Testable (a->b) where
    test f r = test (f (arby r1)) r2
    where (r1,r2) = split r

split :: RandSupply -> (RandSupply, RandSupply)
propRev :: [Int] -> Bool

test propRev r
= test (propRev (arby r1)) r2
where (r1,r2) = split r
= propRev (arby r1)
Type classes are the most unusual feature of Haskell’s type system.
Type-class fertility

- Wadler/Blott type classes (1989)
  - Higher kinded type variables (1995)
  - Multi-parameter type classes (1991)
    - Overlapping instances
    - "newtype deriving"
    - Derivable type classes
  - Implicit parameters (2000)
    - Functional dependencies (2000)
    - Extensible records (1996)
  - Associated types (2005)

Variations

Applications
- Computation at the type level
- Generic programming
- Testing
Type classes and object-oriented programming

1. Haskell "class" ~ OO "interface"
A Haskell class is more like a Java interface than a Java class: it says what operations the type must support.

```haskell
class Show a where
    show :: a -> String

f :: Show a => a -> ...

interface Showable {
    String show();
}

class Blah {
    f( Showable x ) {
        ...x.show()...
    }
}
```
Haskell “class” ~ OO “interface”

- No problem with multiple constraints:

```haskell
f :: (Num a, Show a) => a -> ...
```

- Existing types can retroactively be made instances of new type classes (e.g. introduce new Wibble class, make existing types an instance of it)

```haskell
class Blah {
    f( ??? x ) {
        ...x.show()...
    }
}
```

```haskell
class Wibble a where
    wib :: a -> Bool

instance Wibble Int where
    wib n = n+1
```

```haskell
interface Wibble {
    bool wib()
}

...does Int support Wibble?....
```
Type classes and object-oriented programming

1. Haskell “class” ~ OO “interface”
2. Type-based dispatch, not value-based dispatch
Type-based dispatch

- A bit like OOP, except that method suite (vtable) is passed separately?

```haskell
class Show where
  show :: a -> String

f :: Show a => a => a -> ...
```

- No!! Type classes implement **type-based dispatch**, not **value-based dispatch**
The overloaded value is returned by \texttt{read2}, not passed to it.

It is the dictionaries (and type) that are passed as argument to \texttt{read2}.
So the links to **intensional polymorphism** are closer than the links to **OOP**.

The dictionary is like a proxy for the (interesting aspects of) the type argument of a polymorphic function.

\[
f :: \forall a. a \rightarrow \text{Int}
f t (x :: t) = \ldots \text{typecase } t \ldots
\]

\[
f :: \forall a. C a \Rightarrow a \rightarrow \text{Int}
f x = \ldots \text{(call method of } C) \ldots
\]
```haskell
class Typeable a where
typeRep :: a -> TypeRep

data TypeRep = TR String [TypeRep]

- e.g. typeRep "foo" = TR "List" [ TR "Char" [] ]

instance Typeable Int where
typeRep _ = TR "Int" []

instance Typeable a => Typeable [a] where
typeRep (x:xs) = TR "List" [typeRep x]  -- ???

Not really a string, of course
```
Reflection

class Typeable a where
  typeRep :: a -> TypeRep

data TypeRep = TR String [TypeRep]

- e.g. typeRep "foo" = TR "List" [ TR "Char" [] ]

instance Typeable Int where
  typeRep _ = TR "Int" []

instance Typeable a => Typeable [a] where
  typeRep _ = TR "List" [typeRep (undefined :: a)]

The value argument is never looked at; it plays the role of a type argument

Hence ⊥ is fine
Type classes and object-oriented programming

1. Haskell “class” ~ OO “interface”
2. Type-based dispatch, not value-based dispatch
3. Generics (i.e. parametric polymorphism), not subtyping
Polymorphism: same code works on a variety of different argument types

- Subtyping (= Subclassing)
- Parametric polymorphism (= Generics)

**OO culture**

- cost :: Car -> Int
cost works on Fords, Renaults...

**ML culture**

- rev :: [a] -> [a]
  rev works on [Int], [Char],...
Generics, not subtyping

- Haskell has **no sub-typing**

```
data Tree = Leaf | Branch Tree Tree

f :: Tree -> Int
f t = ...
```

- Ability to act on argument of various types achieved via type classes:

```
square :: (Num a) => a -> a
square x = x*x
```

f’s argument must be (exactly) a Tree

Works for any type supporting the Num interface
Generics, not subtyping

- Means that in Haskell you must anticipate the need to act on arguments of various types

\[ f :: \text{Tree} \rightarrow \text{Int} \]
\[ \text{vs} \]
\[ f' :: \text{Treelike}\ a \Rightarrow a \rightarrow \text{Int} \]

(in OO you can retroactively sub-class Tree)
No subtyping: inference

- Type annotations:
  - Implicit = the type of a fresh binder is inferred
    
    ```java
    f x = ...
    ```
  - Explicit = each binder is given a type at its binding site
    
    ```java
    void f( int x ) { ... }
    ```

- Cultural heritage:
  - Haskell: everything implicit
    type annotations occasionally needed
  - Java: everything explicit;
    type inference occasionally possible
No subtyping: inference

- Type annotations:
  - Implicit = the type of a fresh binder is inferred
    ```java
    f x = ...;
    ```
  - Explicit = each binder is given a type at its binding site
    ```java
    void f( int x ) { ... }
    ```

- Reason:
  - Generics alone => type engine generates equality constraints, which it can solve
  - Subtyping => type engine generates subtyping constraints, which it cannot solve (uniquely)
OOP can lose information

- **In Java (ish):**
  ```java
  INum inc( INum x )
  ```
  - Result: will support INum
  - Argument: must support INum

- **In Haskell:**
  ```haskell
  inc :: Num a => a -> a
  ```
  - Result has precisely same type as argument

- **Compare...**
  ```plaintext
  x :: Float
  ...(x.inc)...
  ```
  ```plaintext
  x :: Float
  ...(inc x)...
  ```
  - INum
  - Float
In practice, because many operations work by side effect, result contra-variance doesn’t matter too much.

In a purely-functional world, where `setColour` and `setPosition` return a new `x`, result contra-variance might be much more important.

F#'s immutable libraries don’t use subclassing (binary methods big issue here too; eg set union)
Java and C# both (now) support **constrained generics**

\[
\text{A inc}\langle\text{A}\rangle(\text{A } x) \\
\text{where A:INum} \{ \\
\text{...blah...} \\
\}
\]

- Very like

\[
\text{inc :: Num } a \Rightarrow a \rightarrow a
\]

- (but little used in practice, I believe)
Why? So that this works

```csharp
interface IEnumerator<out T> {
    T Current;
    bool MoveNext();
}

m( IEnumerator<Control> )
IEnumerator<Button> b
...m(b)...
```

- Button is a subtype of Control, so
- IEnumerator<Button> is a subtype of IEnumerator<Control>

Legal iff T is only returned by methods, but not passed to a method, nor side-effected
Variance

- OOP: must embrace variance
  - Side effects => invariance
  - Generics: type parameters are co/contra/invariant (Java wildcards, C#4.0 variance annotations)
  - Interaction with higher kinds?

```haskell
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

(Only Scala can do this, and it's very tricky!)

- Variance simply does not arise in Haskell.
- And we need constrained polymorphism anyway!
Two approaches to polymorphism

- Each approach has been elaborated considerably over the last decade

What differences remain?
Can one develop a unified story?

Add type classes, type families, existentials
Add interfaces, generics, constrained generics

Subtle
Parametric polymorphism = Generics
Parametric polymorphism and subtyping both address polymorphism

Subtyping alone definitely isn’t enough

Having both is Jolly Complicated (honourable mention for Scala).

Having all of both is infeasible (higher kinds, kind polymorphism, ...)

Parametric polymorphism alone seems pretty close to “enough”
In a language with
• Generics
• Constrained polymorphism
do you (really) need subtyping too?

James Gosling: What would you take out? What would you put in? To the first, James evoked laughter with the single word: Classes. He would like to replace classes with delegation since doing delegation right would make inheritance go away.

http://www.newt.com/wohler/articles/james-gosling-ramblings-1.html