Advances in Programming Languages

APL6: Types, Classes, Haskell

Ian Stark

School of Informatics
The University of Edinburgh

Friday 8 October
Semester 1 Week 3
Some Types in Haskell

This is the first 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

Many programming languages use **types**, in many ways. **Java** does, with types and **subtypes**. Subtypes are distinct from **inheritance**, and issues like **covariance**, **contravariance** and **invariance** can be tricky.

The **Haskell** language, named after logician **Haskell Curry**, introduced **type classes** to balance **parametric** with **ad-hoc polymorphism**. It turns out that they can do much more besides, as shown in tools like **QuickCheck**.
Outline

1 Types

2 Types in Object-Oriented Languages

3 Haskell Curry

4 Type Classes in Haskell

5 Examples of Haskell Type Classes

6 Closing
Some types

A selection of types from some languages.

C/C++

int, long, float, unsigned int, char
int [], char*, char&, int(*)(float,char)

OCaml

int, int64, bool, char, string, unit
string*string, int list, bool array
int→int, int→string→char, 'a list → 'a list

Java

Object, byte[], boolean
StringBuffer, LinkedList, TreeSet, ArrayList<String>
IllegalPathStateException, BeanContextServiceRevokedListener
What do people do with types?

- Type checking
- Static type checking
- Dynamic type checking
- Type annotation
- Type inference
- Structural typing
- Nominative typing
- Subtyping
- Duck typing
- Effect types
- Soft typing
- Gradual typing
- Dynamic types
- Blame typing
To find out more...

Benjamin C. Pierce.  
*Types and Programming Languages.*  
... and lots more

Outline

1. Types
2. Types in Object-Oriented Languages
3. Haskell Curry
4. Type Classes in Haskell
5. Examples of Haskell Type Classes
6. Closing
Java is serious about abstraction

Java works almost entirely through class-based object-oriented programming; it encourages the use of abstract classes through inheritance and interfaces; and it does not expose the private workings of classes and packages.

Java is serious about typing

Java has strong static typing: all programs are checked for type-correctness at compile-time. Bytecode is checked again when classes are loaded, by the bytecode verifier, before execution. Even the new invokedynamic bytecode of Java 7 checks its dynamically created code.

Still, things do not always go as well as one might hope...
What is subtyping?

Subtyping is a well-established part of the object-oriented paradigm: an object in a subclass can stand in for an object in a superclass.

Sometimes known as Liskov’s *principle of substitutivity*:

\[
\text{properties that can be proved using the specification of an object’s presumed type should hold even though the object is actually a subtype of that type}
\]

We shall see this again later in the context of program specification and verification.

---

B. Liskov and J. Wing.
A behavioral notion of subtyping.

*ACM Transactions on Programming Languages and Systems, 16(6):1811–1841, November 1994.*
Java has subtyping: a value of one type may be used at any more general type. So `String ⩽ Object`, and every `String` is an `Object`.

Not all is well with Java types

```java
String[] a = { "Hello", "world" }; // A small string array
Object[] b = a; // Now a and b are the same array
b[0] = Boolean.FALSE; // Drop in a Boolean object
String s = a[0]; // Oh, dear
System.out.println(s.toUpperCase()); // This isn’t going to be pretty
```

This compiles without error or warning: in Java, if $S ⩽ T$ then $S[] ⩽ T[]$. Except that it isn’t. So every array assignment gets a runtime check.
Subtype variance

The issue here is with parameterized types like `String[]` and `List<Object>;` or in Haskell `Maybe a` and `(a,b)→(b,a).

Suppose some type `A⟨X⟩` depends on type `X`, and types `S` and `T` have `S ⩽ T`. Then the dependency of `A` on `X` is:

- **Covariant** if `A⟨S⟩ ⩽ A⟨T⟩`
- **Contravariant** if `A⟨S⟩ ⩾ A⟨T⟩`
- **Invariant** if neither of these holds.

For example, in the Scala language, type parameters can be annotated with variance information: `List[+T]`, `Function[−S,+T].`

In Java, arrays are typed as if they were covariant. But they aren’t.

see also parameter covariance in Eiffel
Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is
Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and

(b) correct.
Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
(b) correct.

Building such type systems is a continuing challenge.
Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:
Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance;

(really, it’s not)
Really, it’s not


Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

- subtyping is not inheritance;
Typing in OO languages

Ideally, an statically-checked object-oriented language should have a type system that is

(a) usable, and
(b) correct.

Building such type systems is a continuing challenge.

One problem is that subtyping is crucial to OO programming, but unfortunately:

• subtyping is not inheritance;
• it’s also extremely hard to get right.
Fixing object subtyping has been a busy research topic for several years.

You can see this by observing that the type declared for the max method in the Java collections class has gone from:

(public static Object max(Collection coll)

which always returns an Object, whatever is stored in the collection,

to:

(public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)

and it might still throw a ClassCastException.

(Java 1.2, 1998)
Fixing object subtyping has been a busy research topic for several years. You can see this by observing that the type declared for the `max` method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

which always returns an `Object`, whatever is stored in the collection, to:

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Fixing object subtyping has been a busy research topic for several years. You can see this by observing that the type declared for the `max` method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

which always returns an `Object`, whatever is stored in the collection, to:

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Fixing object subtyping has been a busy research topic for several years. You can see this by observing that the type declared for the `max` method in the Java collections class has gone from:

```java
public static Object max(Collection coll)
```

which always returns an `Object`, whatever is stored in the collection, to:

```java
public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)
```

and it might *still* throw a `ClassCastException`. (Java 7, 2011?)
Fixing object subtyping has been a busy research topic for several years. You can see this by observing that the type declared for the max method in the Java collections class has gone from:

\[
\text{public static Object max(Collection coll)}
\]

which always returns an Object, whatever is stored in the collection, to:

\[
\text{public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)}
\]

and it might still throw a ClassCastException. This is not a criticism: the new typing is more flexible, it saves on explicit downcasts, and the Java folks do know what they are doing.
Haskell Brooks Curry, 1900–1982
Logician
Curry-Howard correspondence

Propositions as Types

<table>
<thead>
<tr>
<th>A and B</th>
<th>A × B</th>
<th>∀x.A(x)</th>
<th>Πx.A(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A or B</td>
<td>A + B</td>
<td>∃y.B(y)</td>
<td>Σy.B(y)</td>
</tr>
<tr>
<td>A → B</td>
<td>A → B</td>
<td>∀X.X ⇒ X</td>
<td>ΛX.X → X</td>
</tr>
<tr>
<td>True</td>
<td>1</td>
<td>Proofs</td>
<td>Programs</td>
</tr>
<tr>
<td>False</td>
<td>0</td>
<td>Proof rewriting</td>
<td>Program execution</td>
</tr>
</tbody>
</table>

The *Coq* proof assistant is built on the correspondence between proofs and terms, leading to features like *computational reflection* and *program extraction*. Also, the first machine-verified proof of the four-colour theorem.
Currying

\[ A \times B \rightarrow C \quad \equiv \quad A \rightarrow B \rightarrow C \]

\[ (A \& B) \Rightarrow C \quad \iff \quad A \Rightarrow (B \Rightarrow C) \]

Left to right is *currying*.

Right to left is *uncurrying*.

If we had some ham, we could have ham and eggs, if we had any eggs.
Outline

1. Types
2. Types in Object-Oriented Languages
3. Haskell Curry
4. Type Classes in Haskell
5. Examples of Haskell Type Classes
6. Closing
What have classes ever done for us?

Object-oriented languages employ classes, inheritance, and class hierarchy for a range of reasons:

- Modularity
- Encapsulation
- Abstraction
- Polymorphism
- Name spaces
- Code reuse
- ...

Haskell’s *type classes* are quite different, but do provide some similar benefits.
Object-oriented code is *polymorphic* when it can be used with objects from different classes:

```java
Shape[] shapeArray;
...
for (Shape s : shapeArray) { // For every shape in the array ...
    s.draw(); // ... invoke its "draw" method.
}
```

Each `Shape s` may actually be a `Square`, `Circle` or other implementation of `Shape`, each with its own implementation of `draw`. 
Ad-hoc vs. Parametric polymorphism

These implementations may be entirely different, and possibly incompatible: consider `Picture.draw()` and `Cowboy.draw()`.

Christopher Strachey named this *ad-hoc polymorphism*. By contrast, *parametric polymorphism* allows code to have the same action across many types of data.

Parametric polymorphism arrived in Java 5 and C# 2.0 as *generics*, now extensively used in the standard libraries of both languages.

Note that C++ *templates* can achieve a similar effect (and many others), but at the cost of duplicating code during compilation. The ideal for parametric polymorphism is that because the action is the same, the executing code should be the same too.
class Eq a where
(==) :: a -> a -> Bool

instance Eq Int where
i == j = eqInt i j

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

P. Wadler and S. Blott.
How to make ad-hoc polymorphism less ad-hoc.
Proc. POPL '89, pp. 60–76.
Not such ad-hoc polymorphism

\begin{verbatim}
class Eq a where
(==) :: a -> a -> Bool

member :: Eq a => a -> [a] -> Bool
member x [] = False
member x (y:ys) = (x == y) || member x ys
\end{verbatim}

P. Wadler and S. Blott.
How to make ad-hoc polymorphism less ad-hoc.
Pass the dictionary

Type classes can be implemented by *dictionary-passing*. You write:

\[
\text{below} :: \text{Num } n \Rightarrow n \rightarrow n \rightarrow n \\
\text{below } x \ y = y - x
\]

The compiler can turn that into:

\[
\text{below} :: (\text{Num } n) \rightarrow n \rightarrow n \rightarrow n \\
\text{below } d \ x \ y = \text{use}_\text{subtract}_\text{from} \ d \ y \ x
\]

Here \((d :: \text{Num } n)\) is an additional parameter, a *dictionary* of all the operations that make type \(n\) an instance of class *Num*.

This need not be an expensive translation: subsequent optimisations may well then inline and even eliminate the dictionary if all the types can be determined in advance.
Outline

1. Types
2. Types in Object-Oriented Languages
3. Haskell Curry
4. Type Classes in Haskell
5. Examples of Haskell Type Classes
6. Closing
Natural overloaders

Some type classes immediately present themselves as opportunities for overloading:

- Equality: Eq
- Order: Ord
- Print and scan: Show and Read
- Iteration: Enum, Bounded, Ix

Not least, the numeric classes:

- Num, Fractional, Real, Integral, Fractional, Floating, RealFrac, RealFloat

Remember, those are just the classes. The types matching them are Float, Double, Int, Integer, Rational = Ratio Integer, Complex Double, ...
Standard Haskell Classes

S. Marlow, editor.

Haskell 2010 Language Report.

http://www.haskell.org/onlinereport/haskell2010
July 2010
Somewhat unexpectedly, the ingenious applications of type classes go far, far beyond this.

- Pretty-printing
- Modular arithmetic

```haskell
class Modular s a | s -> a where modulus :: s -> a
```

- Phantom types: `data T a = String`
- Arithmetic in the type system: `class Add a b ab`
- SK combinators, logic programming, Turing completeness...
It’s fun to have fun, but you have to know how
QuickCheck

prop_Insert x xs = ordered xs ==> ordered (insert x xs)

Main> quickCheck prop_Insert
OK: passed 100 tests

QuickCheck has no privileged access to the compiler: it uses type classes to obtain the right random generators, for the right number of arguments, for every test.

K. Claessen and John Hughes
QuickCheck: A lightweight tool for random testing of Haskell programs
Outline

1. Types
2. Types in Object-Oriented Languages
3. Haskell Curry
4. Type Classes in Haskell
5. Examples of Haskell Type Classes
6. Closing
Many programming languages use types, in many ways. Java does, with types and subtypes. Subtypes are distinct from inheritance, and issues like covariance, contravariance and invariance can be tricky.

The Haskell language, named after logician Haskell Curry, introduced type classes to balance parametric with ad-hoc polymorphism. It turns out that they can do much more besides, as shown in tools like QuickCheck.
Reading

Homework

For Tuesday’s lecture, read the following paper and set of slides.

- **P. Wadler and S. Blott.**  
  How to make ad-hoc polymorphism less ad-hoc.  
  *Proc. POPL ’89*, pp. 60–76.

- **S. L. Peyton Jones.**  
  Classes, Jim, but not as we know them  
  Invited talk at ECOOP 2009.  
If you are interested in type classes, and in particular how they can be efficiently implemented, read these.

- L. Augustsson
  Implementing Haskell Overloading
  *Proc. FPCA '93*

- J. Peterson and M. Jones
  Implementing Type Classes
  *Proc. PLDI '93*

- C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler
  Type classes in Haskell,
  *Proc. ESOP '94*
Further Reading

It’s fun to have fun, but you have to know how.

Dr Seuss
The Cat in the Hat
Random House, 1955