Discussion questions for
How to make ad-hoc polymorphism less ad hoc

COMP 150 - Applied Functional Programming

September 26, 2012

Background

In addition to the classes in Wadler and Blott's paper, standard Haskell defines the `Show` and `Read` type classes for converting values to and from strings. We have functions `show` and `read` with these types:

```
show :: (Show a) => a -> String
read :: (Read a) => String -> a
```

And there are *lots* of instances, some of which are shown in Figure 1 on the next page of this handout.

You might also find it useful to know that the `Num` class has been extended:

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

Try firing up the interactive interpreter `ghci` and typing

```
:info Num
```

Paper questions

(1) Both C++ and Ada enable a programmer to overload functions, procedures, methods, and operators. So what's the big deal about type classes? Are there things type classes can do that you can't do in C++ or Ada? If so, list them and give an example of each.

(2) How do you imagine the Haskell community reacted to type classes? How might the implementors have reacted?

Technical questions

(3) Look at the instance declarations in Figure 3 on page 6 (or page 65). If you remove the definitions of the witnesses, what's left? Does it remind you of anything?

If not, please write the appropriate declarations on the board and see what ideas emerge.

Type classes in practice

(4) As a simple example, suppose that a type is `Summable` if it contains `Integers`. We want an overloaded function

```
intsum :: Summable a => a -> Integer
```

Write a class declaration and a few instance declarations.

When you roll your own, you learn how to make new things. We know that we can use `not` to get the complement of a `Bool`. If we have a predicate `p` (of one argument) returning `Bool`, we can write

```
not . p
```

But what if we want to negate a two-place predicate? We can't write `not . (<)`, and stuff like

```
not2 f = \ x y -> not $ f x y
```

gets old quickly.

(5) Define a *type class* and *instance declarations* to create an overloaded function `complement` that works for predicates of any arity. In particular,

```
complement isJust = isMaybe
complement (<)    = (>=)
```

and so on for predicates of arbitrary arity.

```
Show Bool                                   Read Bool
(Show a) => Show (Maybe a)                  (Read a) => Read (Maybe a)
(Show a, Show b) => Show (Either a b)       (Read a, Read b) => Read (Either a b)
(Show a) => Show [a]                        (Read a) => Read [a]
Show Int                                    Read Int
Show Integer                                Read Integer
Show Float                                  Read Float
Show Double                                 Read Double
(Show a, Show b) => Show (a, b)             (Read a, Read b) => Read (a, b)
```

Figure 1: Instances for `Read` and `Show`

DVD Packing

(6) What were your best results packing DVDs, and how did you get them?

(7) How could you encode alternative strategies for packing DVDs?

(8) In what ways does your solution to the DVD problem exploit laziness and higher-order functions for modularity?

(9) What are the *types* of the functions you define, and what, if anything, can you learn from looking at the types?