Types and strings

PATRICK D. ELLIOTT

APRIL 20, 2023

Basic typeclasses

Recall our basic type for individuals.

data E = John | Mary | Bill | Sue

We haven't given ghc any further information about this type, so there's not much we can do with it. See what happens if you evaluate the following:

John == John

What about the following:

True == True

Basic typeclasses cont.

The reason for the contrast here is that **Bool** by default is an instance of the type class **Eq**, which is the class of types that contain things that can be compared and determined to be equal in value.

Since we didn't explicitly say that E is an instance of Eq, ghc doesn't assume that it is.

Likewise, try evaluating the following in ghci:

Deriving typeclasses

We'll learn later how to declare typeclass instances, but in the mean time ghc has convenient mechanisms for automatically generating sensible typeclass instances for simple types.

data E = John | Mary | Bill | Sue deriving (Eq,Show)

Constrained polymorphism

· Inspect the type of id.
 Now inspect the type of (==), which is a function that tests for equality.
· Polymorphism is used to constrain typeclasses.
 The fewer typeclass constraints on a polymorphic type signature, the fewer assumptions the polymorphic function can make about its arguments.

Using typeclasses

What do you think will happen if you declare the in a source file?

same :: Eq a \Rightarrow a \Rightarrow b \Rightarrow Bool same a b \Rightarrow a \Rightarrow b

Using typeclasses cont.

Remember that free type variables are implicitly universally quantified.

Informally, this means that the type of id is $a \rightarrow a$, for all a in the set of types.

Type class constraints restrict the universal quantification to just types which belong to particular classes:

This means that the type of (==) is $a \rightarrow a \rightarrow Bool$, for all a that

belong to the ${\bf Eq}$ class.

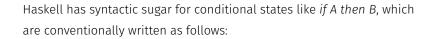
Combining typeclass restrictions

Typeclass restrictions can be combined. We've alluded to this before, but the typeclass **Show** is used to classify types whose inhabitants can be converted into strings (via the **show**) function.

What does the following function do?

```
func :: (Eq a, Show a) => a -> a -> String
func a b = if
  a == b
  then (show a) ++ " is equal to " ++ (show b)
  else "try again!"
```

Sidenote: conditionals



if _condition then _expressionA else _expressionB

You can use conditionals anywhere where you could use **_expressionA** or **_expressionB** (the expressions must be of the same type).

What does the following function do?

toyFunc n = if even n then n + 1 else n - 1

Conditionals and syntactic sugar

It's important to remember that anything that isn't function-argument application in haskell is *syntactic sugar*.

To illustrate, we could implement conditionals as a standard function:

toyFunc2 n = cond (even n) (n + 1) (n - 1)

Tuples

Tuples are a ubiquitous syntactic construct, defined in haskell as a special kind of type known as a *product type*.

Let's look at the data declaration for tuples:

$$(,)$$
 a b = $(,)$ a b

- This is quite different from what we've seen so far.
 - The datatype declaration involves a function (called a type constructor) that takes two type arguments a, b.
 - · Type constructors create types from types.
 - For example, (,) Int String is a distinct type from (,) String Int.
 - · (a,b) is syntactic sugar for (,) a b.

Working with tuples

Consider some tuples:

```
("haskell", "rocks")
("haskell", 1)
```

We can write functions fst and snd using pattern matching to extract the elements of a tuple (these are provided already in the prelude).

```
fst :: (a,b) -> a
fst (a,b) = a
snd :: (a,b) -> b
snd (a,b) = b
```

Exercise

· Write a function **swap** that takes a tuple, and swaps the elements around.

write a function condTup that takes a bool t, two tuples, (a,b),
 (c,d), and gives back a tuple of tuples (a,c) if t is true, and
 (b,d) otherwise (tip: think carefully about the type signature!).

Solution

swap :: (a,b) -> (b,a)
swap (a,b) = (b,a)

condTup :: Bool -> (a,a) -> (b,b) -> (a,b)
condTup True (a,b) (c,d) = (a,c)
condTup False (a,b) (c,d) = (b,d)



References