CS 115
Functional Programming

∃ ∀

Lecture 18: May 13, 2016

Existential Types

∀ ∃ ∀ ∃ ∀ ∃ ∀ ∃ ∀ ∃

Today

- Existential types
- The `Typeable` type class
- The `cast` function
- Applications:
  - "Interfaces" (like in Java)
  - Simulating dynamic typing
  - Extensible exceptions and the `Control.Exception` module

Motivation

- Haskell type classes give Haskell programmers great power to express features that many otherwise unrelated types may contain
- Classic example: every instance of **`Num`** must be something that can be added, multiplied, negated, *etc.*
- Often people say that Haskell's type classes are "sort of like Java interfaces" in this respect
- However, they really aren't like Java interfaces in one critical aspect

Motivation

- Java programmers can express the notion of "a collection of items, each of which implements a particular interface but which may otherwise be of different types"
- This is not possible in basic Haskell!
- You *can* have a collection all of whose elements are instances of a type class *e.g.*

```
addList :: Num a => [a] -> a
```

- Here, all elements of type `a` are instances of `Num`
- But there is an additional constraint here – what?

Motivation

```
addList :: Num a => [a] -> a
```

- All elements of the list must have the *same* type (type `a`)!
- This is a *very* severe restriction
- It prevents us from using collections in many situations where we would like to use them
- Any ideas of how we could try to overcome this limitation?

List of arbitrary types

- We could define a new datatype where every instance of the datatype is an instance of *e.g.* the **`Num`** type class:

```
data Value =
    IntVal Integer
  | FloatVal Float
  | Rational   -- from Data.Ratio
  | Complex    -- from Data.Complex
  | ...
```

- Problems with this approach?

List of arbitrary types

```
data Value =
    IntVal Integer
  | FloatVal Float
  | Rational  -- from Data.Ratio
  | Complex   -- from Data.Complex
  | ...
```

- If we want to extend the type, we have to add more constructors and rewrite all code using the datatype!

List of arbitrary types

```
data Value =
    IntVal Integer
  | FloatVal Float
  | Rational   -- from Data.Ratio
  | Complex    -- from Data.Complex
  | ...
```

- So this approach is not very flexible
- Existential types provide a much better alternative when you want to write code that will not have to be rewritten when new types are added to the datatype

List of showable types

- Let's start with a simple example: how do we define a datatype which can hold any value as long as it's an instance of type class **Show**?
- Before we try this, we will need to enable the GHC extension **ExistentialQuantification**
- In a source code file, write this at the top:

```
{-# LANGUAGE ExistentialQuantification #-}
```

- In **ghci** you can type

```
:set -XExistentialQuantification
```

Language pragmas

- Lines of this form

```
{-# LANGUAGE ExistentialQuantification #-}
```

- are called "language pragmas"
- What they do is switch on various GHC extensions to the Haskell language
- There are many such extensions, some of which are used heavily, so this is a highly-used feature!
- Here, we only need to switch on the **`ExistentialQuantification`** feature
  - Can have more features, separated by commas

List of showable types

- Now we can define our datatype:

```
data Showable = forall a . Show a => Showable a
```

- Some points to make about this:
- 1) The name of a constructor of a type can be the same as the name of the type (different namespaces)
- 2) The type variable `a` found in the datatype is not found outside the constructor (`Showable` is a type, not a type constructor)

List of showable types

```
data Showable = forall a . Show a => Showable a
```

- The `forall` word is confusing, since it's usually considered to mean *universal* quantification, not existential quantification
- Intuition: can have any value as an argument to the `Showable` constructor, as long as its type is an instance of type class `Show` (it works "for all" types that are instances of type class `Show`)

List of showable types

```
data Showable = forall a . Show a => Showable a
```

- You can also think of it as "a `Showable` value asserts that *there exists* a value which is an instance of `Show`"
- However you think about it, what it does is clear: it allows any value whose type is an instance of `Show` to be an argument to the `Showable` constructor

List of showable types

```
data Showable = forall a . Show a => Showable a
```

- Note also the overloading of the dot (`.`) syntax to mean something other than what we've seen so far:
  - the function composition operator
  - module qualification (*e.g.* `Data.List.init`)
- In fact, normal polymorphic types have an implicit `forall` before the type signature:

```
length :: forall a . [a] -> Int
```

  - meaning *universal* quantification in this case

List of showable types

```
data Showable = forall a . Show a => Showable a
```

- Here are valid elements of this type:

```
Showable (10 :: Int)
Showable "foo"
Showable '\n'
Showable True
```

- We can form lists of `Showable`s:

```
[Showable 10, Showable "foo", Showable '\n']
```

- This has the type `[Showable]`

List of showable types

- We might want to make this type an instance of **Show**, but this won't work:

```
data Showable = forall a . Show a => Showable a
  deriving (Show)
```

- GHC is not able to automatically derive a **Show** instance here, so you have to define one manually:

```
instance Show Showable where
  show (Showable s) = "Showable " ++ show s
```

List of showable types

- Now you can type **Showables** into **ghci**:

```
ghci> Showable 10
Showable 10
ghci> Showable "foo"
Showable "foo"
ghci> [Showable 10, Showable "foo"]
[Showable 10, Showable "foo"]
```

List of showable types

- OK, so now we can put values of different types into a list as long as they are instances of the same type class
- Question: what can we do with this list?
- Answer: all we can do is to `show` the elements of the list (convert them to strings) because that is all we know about the list elements!

Java-like "interfaces"

- This use of existential types is precisely what you get with Java interfaces!
- Java:
  - An interface represents a set of operations that a datatype (class) may implement
  - Can have collections of objects implementing a particular interface
  - All you can do with members of these collections is apply the methods of the interface

Java-like "interfaces"

- This use of existential types is precisely what you get with Java interfaces!
- Haskell with existential types:
  - A type class represents a set of operations that a type may implement
  - Can have collections of objects implementing a particular type class (with existentials)
  - All you can do with members of these collections is apply the methods of the type class

Significance

- Although Haskell is not an object-oriented language...
- ...with existentials, you get the most important feature of OO languages (interfaces)
- Note, though, that many uses of interfaces in Java can be handled through normal use of type classes in Haskell (resolved at compile time, not run time)
- But "collections of arbitrary types implementing an interface" requires existential types in Haskell
- Only use this when ordinary uses of type classes are insufficient

Beyond interfaces

- In addition to interfaces, Java also has the notion of a base `Object` class that is a superclass of every other class (*i.e.* every instance of any class is also an instance of the `Object` class)
- Java can automatically "cast" an instance of a class to any of its superclasses (guaranteed to succeed)
- Java programmers can also *manually* cast an instance of a class to any of its subclasses
  - not guaranteed to succeed; get a `ClassCastException` if not successful
- Can we do these things in Haskell?

The `Typeable` type class

- Haskell is not an OO language, so if we can do these things, the form of the solution will necessarily be different
- The key ingredient is a special type class called **`Typeable`**
- This is the most general type class of all; it imposes almost no restrictions on its instances!
- It will be possible to cast Haskell values which are **`Typeable`** from one type to another (possibly failing)

The `Typeable` type class

- The `Typeable` type class is defined in the module `Data.Typeable` as follows:

```
class Typeable a where
   typeOf :: a -> TypeRep
```

- `TypeRep` is a concrete representation of a type that can be compared for equality efficiently (details aren't important)
- This gives Haskell the ability to take two objects, compute their `TypeRep`s, and compare them to see if two objects are of the same type

The `Typeable` type class

- Example:

```
ghci> import Data.Typeable
ghci> typeOf (10 :: Int) == typeOf (10 :: Integer)
False
ghci> typeOf (10 :: Int) == typeOf (43 :: Int)
True
```

- GHC can also automatically derive instances of the `Typeable` type class for any new datatype
  - have to add `deriving (Typeable)` to the `data` definition
  - need to use the `DeriveDataTypeable` language pragma to do this

Casting

- Inside the `Data.Typeable` module there is this amazing function:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

- Meaning: if `a` and `b` are two types that are instances of `Typeable`, then you can try to cast an object of type `a` to an object of type `b`, and it may or may not succeed
- The implementation of this uses an extremely unsafe function called `unsafeCoerce` that we don't have to worry about
  - `cast` itself is safe (phew!)

Application: dynamic typing

- Haskell is a statically-typed language
- Dynamically-typed languages like Scheme or Python can sometimes be more convenient than statically-typed languages
- Good news! With the `Typeable` class and casting, we can simulate dynamic typing in Haskell!
- Let's define a multiplication function that can work on arbitrary types
  - will be able to multiply `Int`s and `Float`s, `Int`s and `String`s, etc.

The `Dyn` existential type

- First, define a dynamic type `Dyn`:

```
data Dyn = forall a . Typeable a => Dyn a
```

- This works, but it's convenient to restrict it to types which also implement `Show`:

```
data Dyn = forall a . (Show a, Typeable a) => Dyn a
```

- These are existential types, so we need this pragma at the beginning of the file:

```
{-# LANGUAGE ExistentialQuantification #-}
```

The `Dyn` existential type

- We would like our `Dyn` type to be an instance of `Show`, but again, GHC can't automatically derive this for existential types, so we write our own instance:

```
instance Show Dyn where
  show (Dyn x) = "Dyn " ++ show x
```

- Now we can enter `Dyn` values into `ghci` and see them printed out:

```
ghci> Dyn "foo"
Dyn "foo"
```

Type representations

- Let's define some type representations for convenience:

```
int :: TypeRep
int = typeOf (0 :: Integer)

double :: TypeRep
double = typeOf (0 :: Double)

string :: TypeRep
string = typeOf ("a" :: String)
```

Dynamic functions

- With what we know now, we can define dynamic functions:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  let ti = typeOf i
      tj = typeOf j
  in if ti == int && tj == int
        then Dyn (fromJust (cast i)
                 * fromJust (cast j) :: Integer)
        else ... (other cases)
```

- `cast` returns a `Maybe` type, which we convert to a regular type using `fromJust` (will succeed here)

Aside: `fromJust`

- Here is the definition of `fromJust` (from the module `Data.Maybe`):

```
-- The 'fromJust' function extracts the
-- element out of a 'Just' and throws an error
-- if its argument is 'Nothing'.
fromJust :: Maybe a -> a
fromJust Nothing  = error "Maybe.fromJust: Nothing"
fromJust (Just x) = x
```

Simplifying (1)

- This approach works, but it's way too tedious to use in practice
- Let's define some helper functions to make this easier to use

```
get :: (Typeable a, Typeable b) => a -> b
get = fromJust . cast
```

- `get` is used like `cast`, except that you only use it when you know that types `a` and `b` are the same
  - because their `TypeRep`s are the same!
  - the `cast` will always succeed, so `cast` will always return a `Just` value

Simplifying (1)

- Our previous example now becomes:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  let ti = typeOf i
      tj = typeOf j
  in if ti == int && tj == int
        then Dyn (get i * get j :: Integer)
        else ... (other cases)
```

Simplifying (2)

- We still need to get rid of the nested `if` statements for multiple cases
- Can't use a `case` statement with different types (can't pattern match on a `TypeRep` because its data representation is not exported (it's an abstract type))
- Instead, we will define a function called `cond` that behaves like nested `if`/`else` forms (like `cond` in Scheme)

Simplifying (2)

- We will represent cases of the `cond` form as pairs of type representations, along with the code to execute if the input types match
- Signature:

```
cond :: (TypeRep, TypeRep) ->
  [((TypeRep, TypeRep), Dyn)] -> Dyn
```

- Given a pair of input `TypeRep`s and a list of things to do given matching `TypeRep` pairs, find the one to execute and return the corresponding `Dyn` value

Simplifying (2)

- We can take advantage of the `lookup` function in the `Data.List` module, which looks for a key in a list of key/value pairs
- Given that, `cond` is easy to define:

```
cond :: (TypeRep, TypeRep) ->
  [((TypeRep, TypeRep), Dyn)] -> Dyn
cond ts assoc =
  case lookup ts assoc of
    Just d  -> d
    Nothing -> Dyn ()   -- default case
```

Simplifying (2)

- Now our original function can easily be given different cases:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  cond (typeOf i, typeOf j) $
    [((int,    int),    Dyn (get i * get j :: Integer)),
     ((double, double), Dyn (get i * get j :: Double)),
     ((int,    double),
       Dyn (fromInteger (get i) * get j :: Double)),
     ((double, int),
       Dyn (get i * fromInteger(get j) :: Double)),
     ((int,    string), Dyn (mulString (get i) (get j))),
     ((string, int),    Dyn (mulString (get j) (get i)))]
```

Simplifying (2)

- Multiply integer * integer:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  cond (typeOf i, typeOf j) $
    [((int,    int),    Dyn (get i * get j :: Integer)),
     ((double, double), Dyn (get i * get j :: Double)),
     ((int,    double),
       Dyn (fromInteger (get i) * get j :: Double)),
     ((double, int),
       Dyn (get i * fromInteger(get j) :: Double)),
     ((int,    string), Dyn (mulString (get i) (get j))),
     ((string, int),    Dyn (mulString (get j) (get i)))]
```

Simplifying (2)

- Multiply double * double:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  cond (typeOf i, typeOf j) $
    [((int,    int),    Dyn (get i * get j :: Integer)),
     ((double, double), Dyn (get i * get j :: Double)),
     ((int,    double),
       Dyn (fromInteger (get i) * get j :: Double)),
     ((double, int),
       Dyn (get i * fromInteger(get j) :: Double)),
     ((int,    string), Dyn (mulString (get i) (get j))),
     ((string, int),    Dyn (mulString (get j) (get i)))]
```

Simplifying (2)

- Multiply int * double and double * int:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  cond (typeOf i, typeOf j) $
    [((int,    int),    Dyn (get i * get j :: Integer)),
     ((double, double), Dyn (get i * get j :: Double)),
     ((int,    double),
      Dyn (fromInteger (get i) * get j :: Double)),
     ((double, int),
      Dyn (get i * fromInteger(get j) :: Double)),
     ((int,    string), Dyn (mulString (get i) (get j))),
     ((string, int),    Dyn (mulString (get j) (get i)))]
```

Simplifying (2)

- "Multiply" int * string and string * int:

```
mul :: Dyn -> Dyn -> Dyn
mul (Dyn i) (Dyn j) =
  cond (typeOf i, typeOf j) $
    [((int,    int),    Dyn (get i * get j :: Integer)),
     ((double, double), Dyn (get i * get j :: Double)),
     ((int,    double),
       Dyn (fromInteger (get i) * get j :: Double)),
     ((double, int),
       Dyn (get i * fromInteger(get j) :: Double)),
     ((int,    string), Dyn (mulString (get i) (get j))),
     ((string, int),    Dyn (mulString (get j) (get i)))]
```

Simplifying (2)

- "Multiply" int * string and string * int
- These use the **`mulString`** function defined as:

```
mulString :: Integer -> String -> String
mulString n s = concat (replicate (fromInteger n) s)
```

- This concatenates a string with itself n times:

```
ghci> mulString 3 "foo"
"foofoofoo"
```

- This is the Python notion of string "multiplication"

Testing

- Some tests:

```
ghci> mul (Dyn (5 :: Integer)) (Dyn (13 :: Integer))
Dyn 65
ghci> mul (Dyn (5 :: Integer)) (Dyn (13 :: Double))
Dyn 65.0
ghci> mul (Dyn (5 :: Double)) (Dyn (13 :: Double))
Dyn 65.0
ghci> mul (Dyn (5 :: Integer)) (Dyn "foo")
Dyn "foofoofoofoofoo"
ghci> mul (Dyn (5 :: Int)) (Dyn "foo")
Dyn ()
```

Exceptions revisited

- One nice use of existential types in Haskell is to define standard Haskell exceptions
- These can be "thrown" from any code, but can only be caught in the `IO` monad
  - If you need exceptions that can be caught in functional code, you need error-handling monads (last two lectures)
- These exceptions are called "extensible exceptions" and the code is in the `Control.Exception` module

Extensible exceptions

- These exceptions are called "extensible" because you can take nearly any type and make it into an exception type
- The only requirements on the type are that it be an instance of the `Typeable` and `Show` type classes
- The system will also use the `cast` function from `Data.Typeable` to extract exceptions from a generic exception type

Extensible exceptions

- The basic exception type is called `SomeException`
- It has this definition:

```
data SomeException =
  forall e . Exception e => SomeException e
  deriving Typeable
```

- We can see that it's an existential type
- Any datatype that is an instance of the `Exception` type class can be made into a `SomeException` value
- So what is the `Exception` type class?

Extensible exceptions

- Here it is:

```
class (Typeable e, Show e) => Exception e where
    toException :: e -> SomeException
    fromException :: SomeException -> Maybe e
    -- default definitions:
    toException = SomeException
    fromException (SomeException e) = cast e
```

- Key points:

Extensible exceptions

- Here it is:

```
class (Typeable e, Show e) => Exception e where
    toException :: e -> SomeException
    fromException :: SomeException -> Maybe e
    -- default definitions:
    toException = SomeException
    fromException (SomeException e) = cast e
```

- Key points:
  - Exception instances must be instances of `Typeable` and `Show`

Extensible exceptions

- Here it is:

```
class (Typeable e, Show e) => Exception e where
    toException :: e -> SomeException
    fromException :: SomeException -> Maybe e
    -- default definitions:
    toException = SomeException
    fromException (SomeException e) = cast e
```

- Key points:
  - To turn a suitable value into an exception, wrap it with the `SomeException` constructor

Extensible exceptions

- Here it is:

```
class (Typeable e, Show e) => Exception e where
    toException :: e -> SomeException
    fromException :: SomeException -> Maybe e
    -- default definitions:
    toException = SomeException
    fromException (SomeException e) = cast e
```

- Key points:
  - To turn an exception back into its constituent value, use the `cast` function (from `Data.Typeable`)

Exception example

- We'll define a new exception type called `MyException` that can hold a `String` value:

```
data MyException = MyException String
  deriving (Show, Typeable)
```

- The default definitions of the `Exception` type class work fine for this type, so to make it an instance of `Exception` we only have to do this:

```
instance Exception MyException
  -- no method definitions!
```

Exception example

- Now we can throw and catch exceptions of type **`MyException`**:

```
test :: IO ()
test = throw (MyException "foo")
          `catch`
            (\(e :: MyException) -> print e)
```

- The type signature in **`(e :: MyException)`** has a very important role: it will cause the function to only catch exceptions of **`MyException`** type, but propagate all other exceptions
  - N.B. need the **`ScopedTypeVariables`** pragma for this

Exception example

- Typical case: want to catch more than one kind of exception
- Solution: use a function called **`catches`**, with this type signature:

```
catches :: IO a -> [Handler a] -> IO a
```

- Intuition: given an **`IO`** computation, and a list of exception handlers, return an **`IO`** computation that handles the exceptions corresponding to the handlers

Exception example

- The **`Handler`** type itself is an existential type with this definition:

```
data Handler a =
  forall e . Exception e => Handler (e -> IO a)
```

- When an exception is thrown, **`catches`** will try each **`Handler`** in turn until it finds one whose exception matches the provided type signature, then it will execute the corresponding **`IO`** action

Exception example

- Example use of **`catches`**:

```
f = expr `catches`
  [Handler (\(ex :: ArithException) -> handleArith ex),
   Handler (\(ex :: IOException) -> handleIO ex)]
```

- If **`expr`** throws an **`ArithException`** exception, then **`handleArith`** handler will be called
- If **`expr`** throws an **`IOException`** exception, then **`handleIO`** handler will be called

Notice...

- The exception handling system is implemented purely as a Haskell module, not hard-wired into the language
- Haskell's primitive features (type classes, `Typeable`, etc.) are powerful enough to allow users to implement many features that would have to be written into most languages as primitives

Summary

- Although Haskell is a statically-typed language, sometimes we might want more dynamism than such languages usually provide
- Existential types provide a big "escape hatch" into more dynamic forms of programming just like the IO monad provides a big "escape hatch" into imperative forms of programming

Summary

- I would like to claim that "Haskell is the world's best dynamic programming language", but I don't think it's quite true
- What is true is that you don't have to give up all features you enjoyed in dynamically-typed programming languages just because you're using Haskell
- Haskell's type system is very flexible ☺

Next time

- State monads