



CS 115

Functional Programming

Lecture 16: May 9, 2016

Error-handling Monads (part 1)



Functional Programming: Spring 2016



Today

- Error-handling in Haskell
- The **error** function revisited
- Error-handling in the **IO** monad
- Extensible exceptions
- The **Either** datatype
- Using **(Either e)** as an error-handling monad





Error-handling in Haskell

- There are many ways to handle error conditions in Haskell (possibly too many)
- Conceptually, there are two things an error-handling system should be able to do:
 1. Give us a way to signal that an error has occurred and break out of the current computation
 2. Allow us to recover from errors in a controlled way
- So far, we have only seen the **error** function





The **error** function

- The **error** function is generally used in a function when the arguments to the function are invalid:

```
factorial :: Integer -> Integer
```

```
factorial n | n < 0 = error "factorial: invalid input"
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

- We use **error** to abort the computation if the input value **n** is invalid (factorial is not defined for **n < 0**)
- What would happen without this line?





The **error** function

- Without the **error** line:

```
factorial :: Integer -> Integer
```

```
factorial 0 = 1
```

```
factorial n = n * factorial (n - 1)
```

- Give factorial a negative number input:

```
Prelude> factorial (-1)
```

- Infinite loop!
- We need the **error** function to prevent nontermination





The **error** function

- Different example: tail of a list
- Actually defined like this:

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

```
tail [] = error "Prelude.tail: empty list"
```

- Let's try it on the empty list:

```
Prelude> tail []
```

```
*** Exception: Prelude.tail: empty list
```





The **error** function

- Different example: tail of a list
- Imagine if it was defined as:

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

- If we did this:

```
Prelude> tail []
```

```
*** Exception: Non-exhaustive patterns in  
function tail
```

- Here, we are using the **error** function to make the **tail** function *total* (having no undefined cases)





The **error** function

- In both cases, we are using the **error** function to signal an error, but not to recover from it
- **error** is normally used only with non-recoverable situations (situations that shouldn't be recovered from)
- Filling in missing cases to make
 - partial functions total
 - nonterminating cases terminating
- are typical applications of **error**





The `error` function

- `error` has the type:

`error :: String -> a`

- `error` takes a `String` (the error message)
- and conceptually returns a value of any type `a`
- How is this possible?
- Answer: it never returns!
- Type signature is to make compiler happy
- Allows us to use `error` with any function, since its return type will always be "valid" for that function





The `error` function

- In fact, it is possible to recover from the `error` function in the `IO` monad (as we'll see)
- However, this facility is rarely used, both because it's not what `error` is for and because better alternatives exist
- Let's look at other error-handling options





Error-handling in the IO monad

- Inside the **IO** monad, errors can be signaled ("thrown") and recovered from ("caught")
- Haskell uses an elegant mechanism called *extensible exceptions* to define exception types
- Extensible exceptions are like an algebraic datatype that has an unbounded number of constructors
- You can add your own custom constructors to provide for custom error handling
- This uses a mechanism called *existential types* (topic of an upcoming lecture)





Error-handling in the IO monad

- Error-handling requires two functions: **throw** and **catch**
- They are defined in the **Control.Exception** module
- **throw** has this type signature:
`throw :: Exception e => e -> a`
- **catch** has this type signature:
`catch :: Exception e => IO a -> (e -> IO a) -> IO a`
- Let's look at **throw**, **catch**, and **Exception** in more detail





Exception

- **Exception** is a type class which represents exception values
 - usually used with existential types
- Various instances of **Exception** exist which represent common exception cases
 - **IOException** is a record type used to represent typical exception values in the **IO** monad
 - **ErrorCall** is a datatype containing a **String** which is what the **error** function throws
 - ...and many others





throw

- Recall the type signature of **throw**:

throw :: Exception e => e -> a

- Assuming **e** is an instance of **Exception**, **throw** raises an exception and "returns" a value of an arbitrary type **a** (like **error**)
- The return type doesn't matter because **throw** doesn't actually return!
- In fact, **error** is actually defined as:
error s = throw (ErrorCall s)
- ErrorCall** just turns a string into an exception





catch

- Recall the type signature of **catch**:

catch :: **Exception** **e** => **IO a** -> (**e** -> **IO a**) -> **IO a**

- Assuming **e** is an instance of **Exception**, **catch** takes
 - a computation of type **IO a** that can (possibly) throw errors
 - an *error handler* of type (**e** -> **IO a**)
- and returns a new computation of type **IO a** that can handle errors thrown by the original computation





Example

- Trivial example using **error** and **catch**:

```
bogus :: IO ()
```

```
bogus =
```

```
  catch
```

```
    (error "This is an error!")      -- throws an error
```

```
    (\(ErrorCall s) -> putStrLn s) -- handles the error
```

- Running this gives:

```
Prelude> bogus
```

```
This is an error!
```





Extensible exceptions

- This system has one big advantage and one big disadvantage
- Big advantage: can define new **Exception** instances to model specific exceptional conditions that you want to handle
- Big disadvantage: can only catch exceptions in the **IO** monad, which is a *huge* limitation!
- Many computations of interest have exceptional situations, but do not need to be in the **IO** monad





To exception-handling monads

- All of the preceding material is a preamble/ motivation for the material I'll now present on error-handling monads
- As such, you can forget about it!
 - Read the Hoogle documentation on [Control.Exception](#) to learn more about this approach if you're interested
- Error-handling monads offer a much more elegant and specific way to deal with the same problem





"Notions of computation"

- Recall: monads are a general way of modelling different "notions of computation" so that functions that embody these notions can be composed as easily as pure functions
- We've seen
 - functions that may do I/O (the **IO** monad)
 - functions that may fail (the **Maybe** monad)
 - functions that may return multiple values (the list monad)
- Error-handling gives rise naturally to a whole *family* of monads of a particular structure





"Notions of computation"

- Consider a "notion of computation" that represents a computation that, in addition to mapping a value of type **a** to a value of type **b**, can also, in some circumstances, throw an error of some error type **e**
- We can write the type of functions embodying this notion of computation schematically as follows:

a --[possibly fail with a specified error condition]--> **b**





Error handling vs. the **Maybe** monad

- This is very reminiscent of the **Maybe** monad
- Contrast

a --[possibly fail with a specified error condition]--> **b**

- with the characteristic functions of the **Maybe** monad:

a --[possibly fail]--> **b**

- The main difference: an error handling monad is more specific about what errors can occur
- The **Maybe** monad lumps all failures together as **Nothing** values





The **Either** datatype

- Instead of **Maybe**, we'll use the **Either** datatype to represent our computations

- **Either** has this definition:

```
data Either a b =
```

```
    Left a
```

```
  | Right b
```

- **Either** is a binary type constructor (two type arguments **a** and **b**)
- **Either** has the kind $* \rightarrow * \rightarrow *$





The **Either** datatype

- **Either** can represent any data that can have one of two arbitrary types
- We will use the **Left** constructor to represent the error datatype and the **Right** constructor to represent normal (non-erroneous) results
- A simple choice is to have **String** as our error type, representing error messages
- Our characteristic computations then have the type
 $a \rightarrow \text{Either String } b$





The `Either` datatype

`a -> Either String b`

- Let's work with computations of this form to see how cumbersome it will be
- Let's start with an integer division function that checks for division by zero:

```
safe_divide :: Integer -> Integer -> Either String Integer
safe_divide _ 0 = Left "divide by zero"
safe_divide i j = Right (i `div` j)
```

- `div` is the integer (truncating) division function
 - actually works for any `Integral` type





safe_divide

- Let's try this out in `ghci`:

```
Prelude> 36 `safe_divide` 6
```

```
Right 6
```

```
Prelude> 1 `safe_divide` 0
```

```
Left "divide by zero"
```

- So far so good...
- Let's try to use it with a simple computation
- Example: `f i j k = i + j / k`
- What would this look like if we used `safe_divide` instead of just the `/` operator?





safe_divide

- Here we go:

```
f :: Integer -> Integer -> Integer -> Either String Integer
f i j k =
  case j `safe_divide` k of
    Left msg -> Left msg
    Right r -> Right (i + r)
```

- This is pretty complicated compared to the original function!
- Also, the output type is different from the input types, which will make composition difficult
- But let's keep going anyway...





safe_divide

- We can extend `safe_divide` to handle multiple error conditions
- For instance, let's say that trying to divide two integers that are not evenly divisible is an error
- Now we get:

```
safe_divide :: Integer -> Integer -> Either String Integer
safe_divide _ 0 = Left "divide by zero"
safe_divide i j | i `mod` j /= 0 = Left "not divisible"
safe_divide i j = Right (i `div` j)
```





divide

- Let's use `safe_divide` to write a function called `divide` that propagates the divide-by-zero error, but which handles the not-divisible situation by throwing out the remainder:

```
divide :: Integer -> Integer -> Either String Integer
divide i j =
  case i `safe_divide` j of
    Left "divide by zero" -> Left "divide by zero"
    Left "not divisible" -> Right (i `div` j)
    Right k -> Right k
```

- Anything bother you about this code?





divide

```
divide :: Integer -> Integer -> Either String Integer
```

```
divide i j =
```

```
  case i `safe_divide` j of
```

```
    Left "divide by zero" -> Left "divide by zero"
```

```
    Left "not divisible" -> Right (i `div` j)
```

```
    Right k -> Right k
```

- The code is certainly not pretty, but...
- We are pattern matching on literal strings!
- This is not what error messages are intended to be used for
- Easy to get wrong (one typo and it's no good!)
 - Also not exhaustive (why?)





divide

```
divide :: Integer -> Integer -> Either String Integer
```

```
divide i j =
```

```
  case i `safe_divide` j of
```

```
    Left "divide by zero" -> Left "divide by zero"
```

```
    Left "not divisible" -> Right (i `div` j)
```

```
    Right k -> Right k
```

- The right way is to define a specific datatype to represent the kinds of errors we might find in some class of computations
- Here, we are doing arithmetic computations, so let's define a datatype called **ArithmeticError** and rewrite **safe_divide** and **divide** in terms of it





ArithmeticError

```
data ArithmeticError =  
    DivideByZero  
  | NotDivisible  
  -- could add more cases here  
deriving Show
```

- Now we don't need the error messages to represent the errors
- Translating our previous definitions is straightforward





ArithmeticError

```
safe_divide :: Integer -> Integer
              -> Either ArithmeticError Integer

safe_divide _ 0 = Left DivideByZero
safe_divide i j | i `mod` j /= 0 = Left NotDivisible
safe_divide i j = Right (i `div` j)
```

```
divide :: Integer -> Integer
        -> Either ArithmeticError Integer

divide i j =
  case i `safe_divide` j of
    Left DivideByZero -> Left DivideByZero
    Left NotDivisible -> Right (i `div` j)
    Right k -> Right k
```





ArithmeticError

- Limitation of this approach:
- If we want to add another **ArithmeticError** constructor, we might have to modify a lot of code
- We can instead use existential types (like **Control.Exception** does) to get extensible exceptions, but that would make the discussion too complicated, so we'll stick with error types that are simple algebraic datatypes
- Also, many custom error-handling situations can be described in terms of a datatype with a small number of constructors





A more complicated function

- Consider this slightly more complicated function:

$g\ i\ j\ k = i / k + j / k$

- Let's write this using `safe_divide` instead of (`/`):

```
g :: Integer -> Integer -> Integer
    -> Either ArithmeticError Integer
```

```
g i j k =
  case i `safe_divide` k of
    Left err1 -> Left err1
    Right q1 ->
      case j `safe_divide` k of
        Left err2 -> Left err2
        Right q2 -> Right (q1 + q2)
```





A more complicated function

- This is extremely gross!
- One line of code became seven lines
- We will see that monads will simplify this code significantly
- On to error-handling monads...





Error-handling monads

- To work with monads, we need to know what the type signature of the monadic functions are going to be
- In our case, using **Either** to make our result type gives

`a -> Either e b`

- where **e** represents the particular error type (e.g. **String** or **ArithmeticError**)
- Regardless of which error type is used, the structure of the computations will be the same





Issue #1

- Note that we can also use functions with more arguments e.g. with this type signature:

`a -> b -> Either e c`

- **`safe_divide`** and **`divide`** have type signatures like this
- This is OK because of currying; given the first argument the rest of the function has the type

`b -> Either e c`

- which is the basic type of the computation





Issue #2

- Note that the basic form of monadic functions is

$a \rightarrow m\ b$

- for some monad m
- Here, we have functions of the form

$a \rightarrow \text{Either } e\ b$

- How can we reconcile the two?
- What is our monad m going to be in this case?





Issue #2

- Compare

$a \rightarrow m\ b$

$a \rightarrow \text{Either}\ e\ b$

- We can rewrite the latter as:

$a \rightarrow (\text{Either}\ e)\ b$

- because type constructors can be curried just like functions can be!
- $\text{Either}\ e$ has kind $* \rightarrow *$
- $\text{Either}\ e$ is a *unary* type constructor
 - which is what a monad needs to be too!





Issue #2

- Comparing

`a -> m b`

`a -> (Either e) b`

- We can see that the **Monad** instance is going to be **Either e**
- This defines a whole *family* of monads, one for each error type **e**
- (This will become very significant when we discuss state monads)





The Monad instance

- The **Monad** instance for **(Either e)** looks like:
instance Monad (Either e) where
 - definition of >>= for Either e**
 - definition of return for Either e**
- We need to fill in the definitions for **(>>=)** and **return**
- Fortunately, one definition will work for all error types **e**





The >>= operator

- As before, define the >>= operator based on what you want the monad to accomplish

- The type signature will be:

```
(>>=) :: Either e a -> (a -> Either e b) -> Either e b
```

- To define this, consider the two cases of **Either e a** in the first argument
- If the first argument is **Left x**, an error occurred previously and should be passed along
- If the first argument is **Right y**, then **y** is a non-error result and should be passed to the function which is the second argument





The >>= operator

- This gives us our definition:

```
(>>=) :: Either e a -> (a -> Either e b) -> Either e b
(Left x)  >>= _ = Left x    -- propagate the error
(Right y) >>= f = f y        -- extract the non-error value
                                -- and pass to function f
```

- This definition is very similar to the corresponding definition from the **Maybe** monad
- Working by analogy, what should the definition of **return** be?





The `return` method

- `return` has the type signature:

`return :: a -> Either e a`

- The result of `return x` for some `x` cannot be `Left y` for any `y`, since this implies that an error occurred as the result of simply putting a value into the `Either e` monad
- Instead, the probable definition would be
`return x = Right x`
- As usual, we need to check this using the monad laws





The `return` method

- Monad law 1:

Show: `return x >>= f == f x`

`return x >>= f`

→ `Right x >>= f` *-- definition of return*

→ `f x` *-- definition of >>=*

- OK, so monad law 1 checks out





The `return` method

- Monad law 2:

Show: `mv >>= return == mv`

Case 1: `mv == Left x`

`mv >>= return`

→ `Left x >>= return`

→ `Left x` -- definition of `>>=`

→ `mv`

- Case 1 checks out





The `return` method

- Monad law 2:

Show: `mv >>= return == mv`

Case 2: `mv == Right y`

`mv >>= return`

→ `Right y >>= return`

→ `return y` -- definition of `>>=`

→ `Right y` -- definition of `return`

→ `mv`

- Case 2 checks out, so monad law 2 checks out





Monad law 3

- Verifying monad law 3 is somewhat long
- Left as an exercise for the reader, as usual
- The **Monad** instance for **Either e** is thus:

```
instance Monad (Either e) where  
    return x = Right x
```

```
Left x  >>= _ = Left x  
Right y >>= f = f y
```





Recall

- Recall this function:

```
g :: Integer -> Integer -> Integer
    -> Either ArithmeticError Integer
g i j k =
  case i `safe_divide` k of
    Left err1 -> Left err1
    Right q1 ->
      case j `safe_divide` k of
        Left err2 -> Left err2
        Right q2 -> Right (q1 + q2)
```





With monads

- Using monads and the **do** notation, this becomes:

```
g :: Integer -> Integer -> Integer
    -> Either ArithmeticError Integer
```

```
g i j k =
    do q1 <- i `safe_divide` k
       q2 <- j `safe_divide` k
    return (q1 + q2)
```

- Much cleaner and easier to understand!
- But still not as nice as:

```
g i j k = i / k + j / k
```





With `liftM2`

- We might ask if we could write this function like this:

```
g :: Integer -> Integer -> Integer
    -> Either ArithmeticError Integer
```

```
g i j k = (i `safe_divide` k)
          + (j `safe_divide` k)
```

- Unfortunately, this does not type check!
- `(i `safe_divide` k)` and `(j `safe_divide` k)` have type `Either ArithmeticError Integer`
- `+` has type `Integer -> Integer -> Integer` in this context
- However, we can pull another rabbit out of our hat





With `liftM2`

- The module `Control.Monad` defines a function called `liftM2` with this definition:

```
liftM2 :: (Monad m) => (a1 -> a2 -> r)
      -> m a1 -> m a2 -> m r
```

```
liftM2 f m1 m2 =
    do x1 <- m1
       x2 <- m2
       return (f x1 x2)
```

- We can use this to simplify our definition even further





With `liftM2`

- With `liftM2`, our definition becomes:

```
g :: Integer -> Integer -> Integer
    -> Either ArithmeticError Integer
g i j k = (i `safe_divide` k) <+> (j `safe_divide` k)
  where (<+>) = liftM2 (+)
```

- `+` (in this context) expects both operands to have type `Integer`
- `<+>` expects both operands to have the type `Either ArithmeticError Integer`
- If we had one with type `Integer` and the other with type `Either ArithmeticError Integer`, what would we do?





With `liftM2`

- Rewrite the function `f` (defined previously) to use `ArithmeticError` as the error type:

```
f :: Integer -> Integer -> Integer ->
    Either ArithmeticError Integer
```

```
f i j k =  -- i + j/k
  case j `safe_divide` k of
    Left msg -> Left msg
    Right r -> Right (i + r)
```

- Now we can write this as:

```
f i j k = (return i) <+> (j `safe_divide` k)
  where (<+>) = liftM2 (+)
```

- Still a bit ugly, but it works and is concise





Limitations of monads

- Note that there is a fundamental difference between

`g i j k = i / k + j / k`

- and

`g i j k = (i `safe_divide` k) <+> (j `safe_divide` k)`
`where (<+>) = liftM2 (+)`

- The second example imposes a specific order of evaluation on the subexpressions: `(i `safe_divide` k)` will always be evaluated before `(j `safe_divide` k)`
 - Not necessarily the case for the first example





Limitations of monads

- For some monads (**IO** monad, state monads), imposing a particular sequencing is extremely important; the code won't work properly without it
- For other monads (**Maybe**, list, **Either e**), sequencing is often irrelevant, but we get it anyway
- This is the price we pay for the benefits of monads





Next time

- More error-handling monads
- Throwing and catching errors in the **Either** monad
- The **MonadError** type class
- Functional dependencies

