# CS 115
# Functional Programming

*Lecture 17:*

# Error-handling Monads (part 2)

# Previously

- 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

# Today

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

# Recall

- We were doing computations involving **`Integer`**s that could fail in particular ways
- We defined an algebraic datatype called **`ArithmeticError`** to represent our error values:

```
data ArithmeticError =
    DivideByZero
  | NotDivisible
  -- could define more error values here
  deriving Show
```

# Recall

- We used the **`Either`** datatype to define a monad **`Either e`** (for any error type **`e`**), with this definition:

```
instance Monad (Either e) where
  Left e >>= _ = Left e
  Right v >>= f = f v

  return v = Right v
```

# Recall

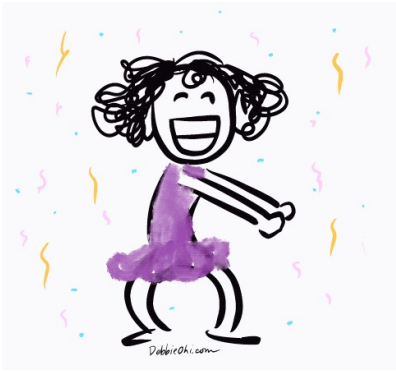
- We started with functions like this:

```
g :: Integer -> Integer -> Integer
       -> Either ArithmeticError Integer
g i j k =  -- i `div` k + j `div` 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)
```

# Recall

- We used the **`Either ArithmeticError`** monad to simplifiy **`g`** to:

```
g :: Integer -> Integer -> Integer
       -> Either ArithmeticError Integer
g i j k =  -- i `div` k + j `div` k
  do q1 <- i `safe_divide` k
     q2 <- j `safe_divide` k
     return (q1 + q2)
```

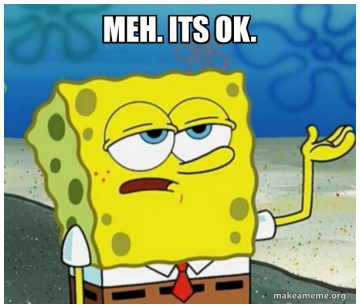


- This is much cleaner than the previous version of **`g`**

# Recall

- We used the **`liftM2`** function to simplify **`g`** still further to:

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




- This is almost as simple as the definition

```
g i j k = i `div` k + j `div` k
```

- that does no error-handling at all!

# So far

- We know how to represent computations that can result in exceptional values using the `Either` datatype and an error type `e`
- We know how to combine such computations using the (`Either e`) monad
- But some critical parts of the error-handling process haven't been dealt with yet

# What's missing

- We need to know how to signal an error situation ("throw" an error)
- We need to know how to recover from an error situation ("catch" an error)
- This will take up the entire lecture
- However, don't worry: it's not hard!

# MonadError

- Haskell defines a type class called **MonadError** which defines the **throw**ing and **catch**ing functions we need
- Definition:

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

- Defined in the module **Control.Monad.Except**
- There's a lot to talk about here!

# MonadError

```
class Monad m => MonadError e m | m -> e ...
```

- **`MonadError`** is a type class with *two* type parameters **`m`** and **`e`**
- **`m`** must be an instance of the **`Monad`** type class
  - hence the constraint **`Monad m =>`**
  - **`m`** is therefore a (unary) type constructor
- **`e`** is the error type used
- The **`| m -> e`** stuff is a *functional dependency*
  - We'll discuss this later in the lecture

# throwError

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

- `throwError` is used to signal an error
- It takes an error value of type `e` and creates a monadic computation of type `m a`
- The type `a` is not relevant, since `throwError` will not "return" in the normal sense
  - much like the `error` function has the type `String -> a`

# catchError

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

- `catchError` is used to recover from an error
- First argument is a monadic computation that may throw an error
- Second argument is an *error handler*
  - takes an error thrown by the first argument
  - "handles" it to give the final result

# Example

- As an example, let's look at our **`safe_divide`** function from last time:

```
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)
```

- We will use this function to derive the **`MonadError`** instance for the **`Either ArithmeticError`** monad

# Example

- Structure of the **`instance`** declaration:

```
instance MonadError ArithmeticError (Either ArithmeticError)
  where
    throwError err = ...
    catchError val handler = ...
```

- Note that **`MonadError`** takes two type parameters:
  - the **`Either ArithmeticError`** monad
  - the **`ArithmeticError`** error type
- You might think that these need to be related
  - And they do! (We'll get back to this)

# throwError

- **throwError** for this instance will have the type:

```
throwError :: ArithmeticError -> Either ArithmeticError a
```

- Therefore, whatever the result type of **throwError**, it cannot depend on the monadic result type **a**
- Therefore, how can we fill in this definition:

```
throwError err = ???
```

- ?

# throwError

```
throwError err = ???
```

- Recall that to signal *e.g.* a divide-by-zero error in `safe_divide`, we have

```
safe_divide _ 0 = Left DivideByZero
```

- We could rewrite this as:

```
safe_divide _ 0 = throwError DivideByZero
```

- from which we get the definition:

```
throwError err = Left err
```

- Or just:

```
throwError = Left
```

# throwError

- Let's rewrite **safe_divide** using **throwError**:

```
safe_divide :: Integer -> Integer
                    -> Either ArithmeticError Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 = throwError NotDivisible
safe_divide i j = Right (i `div` j)
```

# throwError

- This is OK, but the **Right** constructor seems out of place
- Recall that **return** in this monad is just **Right**
- This leads to the next version:

```
safe_divide :: Integer -> Integer
                    -> Either ArithmeticError Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 = throwError NotDivisible
safe_divide i j = return (i `div` j)
```

# throwError

```
safe_divide :: Integer -> Integer
                    -> Either ArithmeticError Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 = throwError NotDivisible
safe_divide i j = return (i `div` j)
```

- This is extremely generic code!
- In fact, (without the type signature) it would work for any instance of **`MonadError`** where the error type is **`ArithmeticError`**
- Knowing this, we can rewrite the code again

# throwError

```
safe_divide :: MonadError ArithmeticError m =>
  Integer -> Integer -> m Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 = throwError NotDivisible
safe_divide i j = return (i `div` j)
```

- All we have changed is the type signature

# throwError

```
safe_divide :: MonadError ArithmeticError m =>
  Integer -> Integer -> m Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 = throwError NotDivisible
safe_divide i j = return (i `div` j)
```

- Two questions remain:
  1. How do we know that `m` is a monad?
  2. How do we know that this monad is somehow connected to `ArithmeticError` (like `Either ArithmeticError` is)?

# throwError

- How do we know that `m` is a monad?
- We might expect that the type signature would have to be

```
safe_divide :: (Monad m, MonadError ArithmeticError m) =>
  Integer -> Integer -> m Integer
```

- This is unnecessary because the class definition is:

```
class Monad m => MonadError e m ...
```

- Conclusion: if `ArithmeticError` and `m` are an instance of `MonadError`, `m` must be a monad

# `throwError`

- How do we know that the monad `m` is somehow connected to the error type `ArithmeticError`?
- This will be revealed shortly when we discuss functional dependencies
- For now, let's move on to error recovery using `catchError`

# catchError

- Let's look at how **`catchError`** would work for the specific types:
  - **`ArithmeticError`** as the error type
  - **`Either ArithmeticError`** as the monad
- In this case, **`catchError`** has this type signature:

```
catchError :: Either ArithmeticError Integer ->
  (ArithmeticError -> Either ArithmeticError Integer) ->
  Either ArithmeticError Integer
```

# catchError

- The first argument to **`catchError`** is the result of a computation in the **`Either ArithmeticError`** monad
- It will be
  - **`Left err`** if an error occurred
  - **`Right val`** if not
- If no error occurred, then the error handler (second argument to **`catchError`**) is never called, and the value **`Right val`** is the result of the entire call to **`catchError`**

# `catchError`

- If an error did occur, the error value must be unpacked from the **`Left err`** value and passed as the argument to the error handler
- The result of this function application will be the result of the entire call to **`catchError`**
- Together, this gives us the definition of **`catchError`** we want

# catchError

```
catchError :: Either ArithmeticError Integer ->
  (ArithmeticError -> Either ArithmeticError Integer) ->
  Either ArithmeticError Integer
catchError (Left err)  h = h err
catchError (Right val) _ = Right val
```

- And that's it!

# catchError

```
catchError :: Either ArithmeticError Integer ->
  (ArithmeticError -> Either ArithmeticError Integer) ->
  Either ArithmeticError Integer
catchError (Left err)  h = h err
catchError (Right val) _ = Right val
```

- We have defined a complete exception handling system as a couple of trivial functions!

# catchError

```
catchError :: Either ArithmeticError Integer ->
  (ArithmeticError -> Either ArithmeticError Integer) ->
  Either ArithmeticError Integer
catchError (Left err)  h = h err
catchError (Right val) _ = Right val
```

- (Contrast with most languages, where exception handling is a complex feature which must be "baked into" the language)

# MonadError

- The entire instance definition is:

```
instance MonadError (Either ArithmeticError)
  where
    throwError = Left
    catchError (Left err)  h = h err
    catchError (Right val) _ = Right val
```

- In fact, none of this is specific to **`ArithmeticError`**, so we can generalize this further

# MonadError

- The most general instance definition of **`MonadError`** for the **`(Either e)`** monad is:

```
instance MonadError (Either e)
  where
    throwError = Left
    catchError (Left err)  h = h err
    catchError (Right val) _ = Right val
```

- This works for all error-handling monads of the form **`(Either e)`** where **`e`** is the error type

# Error handling

- In reality, most of the complexity is being handled by the error handling function
- Consider our previous function **`divide`**:
  - we caught **`NotDivisible`** errors and divided the numbers, throwing out the remainder (no longer an error)
  - we let **`DivideByZero`** errors pass through (*i.e.* it's still an error)
- What would the error handling function look like?

# Error handling

- First try:

```
handler :: ArithmeticError
                -> Either ArithmeticError Integer
handler DivideByZero = Left DivideByZero

handler NotDivisible = Right ???
```

- Not clear what to do in the second case
- Before we deal with this, let's clean up the code by getting rid of the **`Left`**/**`Right`** constructors

# Error handling

- Second try:

```
handler :: ArithmeticError
                -> Either ArithmeticError Integer
handler DivideByZero = throwError DivideByZero

handler NotDivisible = return ???
```

- Problem with the second case:
- We don't have the information needed to recover from the error!
- In previous functions, we have been acquiring this information from an outer scope (not available here)

# Error handling

- To fix this, we need to augment our error type to contain information needed to recover from the error:

```
data ArithmeticError =
    DivideByZero
  | NotDivisible Integer Integer
  -- save the indivisible values
  deriving Show
```

# Error handling

- Using this, we need to rewrite **`safe_divide`** again:

```
safe_divide :: Integer -> Integer
                    -> Either ArithmeticError Integer
safe_divide _ 0 = throwError DivideByZero
safe_divide i j | i `mod` j /= 0 =
  throwError (NotDivisible i j)  -- note difference!
safe_divide i j = return (i `div` j)
```

- Now the **`NotDivisible`** error contains all the information needed to recover from the error
- Let's go back to our error handler

# Error handling

- Third try:

```
handler :: ArithmeticError
                -> Either ArithmeticError Integer
handler DivideByZero = throwError DivideByZero
handler (NotDivisible i j) = return (i `div` j)
```

- Now we can write our **`divide`** function from last time as:

```
divide :: Integer -> Integer
    -> Either ArithmeticError Integer
divide i j = catchError (i `safe_divide` j) handler
```

# Error handling

- We can make this a bit prettier by putting the **`catchError`** after the main computation:

```
divide :: Integer -> Integer
    -> Either ArithmeticError Integer
divide i j = (i `safe_divide` j) `catchError` handler
```

# Error handling

- We can inline the definition of **`handler`**:

```
divide :: Integer -> Integer
    -> Either ArithmeticError Integer
divide i j =
  (i `safe_divide` j)
    `catchError`
      (\e ->
          case e of
            DivideByZero -> throwError DivideByZero
            NotDivisible i j -> return (i `div` j))
```

# Error handling

- In this case, we have `i` and `j` available from context, so `NotDivisible` doesn't need arguments anymore
  - Can change the definition of `ArithmeticError` back to what it was before (design choices!)

```
divide :: Integer -> Integer
    -> Either ArithmeticError Integer
divide i j =
  (i `safe_divide` j)
    `catchError`
      (\e ->
          case e of
            DivideByZero -> throwError DivideByZero
            NotDivisible -> return (i `div` j))
```

# Error handling

- Note the overall structure of functions that handle errors:

```
function arg1 arg2 ... =
  <computation>
    `catchError`
      (\<error> ->
          case <error> of
            <error1> -> <handle error case 1>
            <error2> -> <handle error case 2>
            ...)
```

# Error handling

```
function arg1 arg2 ... =
  <computation>
    `catchError`
      (\<error> ->
          case <error> of
            <error1> -> <handle error case 1>
            <error2> -> <handle error case 2>
            ...)
```

- Very reminiscent of *e.g.* Java exception handling except that multiple **`catch`** clauses have been combined into a single error handler that handles all cases

# Functional dependencies

- We still haven't dealt with one issue: the relationship between the error type `e` and the monad type constructor `m` in the `MonadError` type class
- We have seen the case where
  - `m` is `Either ArithmeticError`
  - `e` is `ArithmeticError`
- It makes sense that there should be a relationship between these two entities
- The question: what kind of a relationship should there be, and how can we enforce it?

# Functional dependencies

- Let's look again at the **`throwError`** function for our monad:

```
throwError :: ArithmeticError
                  -> Either ArithmeticError a
throwError err = Left err
```

- It should be clear that if we are using the **`Either ArithmeticError`** monad, we can only throw errors of the type **`ArithmeticError`** into the monadic computations

# Functional dependencies

- We want to be able to enforce a *constraint* that says: for this monad type (`Either ArithmeticError`) we want to be certain that *only* `ArithmeticError` errors are used with it
- We could generalize this to say that for any `MonadError` instance of the form (`Either e`) we can only use error type `e` as the error type
- Haskell's solution is even more general than this!

# Functional dependencies

- Recall the class definition for **`MonadError`**:

```
class Monad m => MonadError e m | m -> e where
  throwError :: e -> m a
  catchError :: m a -> (e -> m a) -> m a
```

# Functional dependencies

- Consider the first line:

```
class Monad m => MonadError e m | m -> e where
```

- The `| m -> e` part is a *functional dependency* (also known as a *fundep*)
- It enforces a very general relationship between the monadic type constructor `m` and the error type `e`

# Functional dependencies

```
class Monad m => MonadError e m | m -> e where
```

- This constraint does not say anything about the internal structure of monad `m`, or that it has to somehow "contain" the type `e`
- What it does say is that monad `m` *determines* the type `e`
- So: if you know what `m` is, there can be one and only one choice for `e`
- If you define two instances with the same `m` but different `e`s, it will not compile!

# Functional dependencies

```
class Monad m => MonadError e m | m -> e where
```

- In our case, we defined this instance:

```
instance MonadError ArithmeticError
                    (Either ArithmeticError)
```

- Given this, we could later also define

```
instance MonadError String (Either String)
```

- but we couldn't define

```
instance MonadError String
                    (Either ArithmeticError)
```

- because that would violate the functional dependency

# Functional dependencies

- Note that there is no requirement that we define the "sensible" instance first!
- We could start by defining the stupid instance

```
instance MonadError String
                        (Either ArithmeticError)
```

- and if so, we couldn't later define the sensible instance

```
instance MonadError ArithmeticError
                        (Either ArithmeticError)
```

- It's up to you to use the power of fundeps correctly to prevent users from defining stupid instances

# Functional dependencies

- Imagine if we didn't use fundeps with **`MonadError`**
- We could then define these two instances:

```
instance MonadError String
                        (Either ArithmeticError)
instance MonadError ArithmeticError
                        (Either ArithmeticError)
```

- Then in a particular computation in the **`Either ArithmeticError`** monad, we have no idea what kind of errors could be thrown!

# Functional dependencies

- We could legitimately try to catch errors of type `String` inside a computation that throws errors of type `ArithmeticError`, which will not work properly
- Functional dependencies prevent problems like this
- They allow the programmer to state the relationship between types / type constructors in a multiparameter type class so that everything behaves correctly (and enforceably!)

# Functional dependencies

- Functional dependencies have more uses than just in error-handling monads
- They are a basic tool when working with multi-parameter type classes to express relationships between the type (or type constructor) variables of the type class
- We will revisit them in the discussion of state monads

# Using `Either` as a monad

- We don't actually have to define `Either e` as a monad
- You can just import the module `Control.Monad.Except`
  - in older versions of GHC, this was `Control.Monad.Error`, which still exists but is deprecated

# Trade-offs

- This form of exception handling is very simple, but not as powerful as many built-in exception handling systems
- Notice that we had to handle all cases of exceptions in our exception handler, even ones that just propagated through the computation
- Also, propagating exceptions have to go through each function they propagate through
  - can't just "jump down" to the right point in the stack
  - therefore somewhat inefficient

# Coming up

- Existential types!
  - How to simulate dynamic types in a statically-typed language
- State monads!
  - How to simulate state passing in a non-imperative language