

Explicitly Typed Exceptions for GHC

José Iborra
Universidad Politécnica de Valencia

- Compare this code in a programming language:

```
public static void main(String[] args) {  
    String filepath = args[1];  
    Reader f = new FileReader(filepath);  
    doSomethingWith(f);  
}
```

- with the equivalent Haskell code

```
main = do  
    filepath <- liftM head getArgs  
    contents <- readFile filepath  
    doSomethingWith contents
```

- Both can fail with an exception
 - but only one compiler detects it

- Compare this code in a programming language:

```
public static void main(String[] args) {
    String filepath = args[1];
    Reader f = new FileReader(filepath);
    doSomethingWith(f);
}
```

- with the equivalent Haskell code

```
main = do
    filepath <- liftM head getArgs
    contents <- readFile filepath
    doSomethingWith contents
```

- Both can fail with an exception
- but only one compiler detects it

Main.xxxx:10: unreported exception xxxx.io.FileNotFoundException;
must be caught or declared to be thrown

```
Reader f = new FileReader(filepath);
          ^
```

1 error

Checked Exceptions

- Detected and enforced by the compiler
- Usually explicit: possible exceptions appear on the type
 - automatically documented
 - the programmer has no need to think “what exceptions can be raised inside this block ?”
- Checked exceptions are a good thing™

Exception features

- Desirable feature set of exceptions in a programming language:
 - User extensible
 - Hierarchical
 - Checked
 - Explicit
 - Providing Stack Traces

Exception features

- Desirable feature set of exceptions in a programming language:
 - User extensible
 - Hierarchical
 - Checked
 - Explicit
 - Providing Stack Traces

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Double
```

```
eval (Val x)      = x
```

```
eval (Add e1 e2) = eval e1 + eval e2
```

```
eval (Div e1 e2) = eval e1 / eval e2
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Double
```

```
eval (Val x)      = x
```

```
eval (Add e1 e2) = eval e1 + eval e2
```

```
eval (Div e1 e2) = case eval e2 of
    0 -> exception !
    x2 -> ? (eval e1 / x2)
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Double
```

```
eval (Val x)      = x
```

```
eval (Add e1 e2) = eval e1 + eval e2
```

```
eval (Div e1 e2) = case eval e2 of  
    0  -> Left  DivByZero  
    x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = x
```

```
eval (Add e1 e2) = eval e1 + eval e2
```

```
eval (Div e1 e2) = case eval e2 of  
    0 -> Left DivByZero  
    x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x)      = Right x
```

```
eval (Add e1 e2) = eval e1 + eval e2
```

```
eval (Div e1 e2) = case eval e2 of  
    0   -> Left  DivByZero  
    x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x)      = Right x
```

```
eval (Add e1 e2) = case eval e1 of
                    Left e   -> Left e
                    Right x1 -> case eval e2 of
                                  Left e   -> Left e
                                  Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
                    0 -> Left DivByZero
                    x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
  Left e  -> Left e
  Right x1 -> case eval e2 of
    Left e  -> Left e
    Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
  Right 0 -> Left DivByZero
  Right x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of  
    Left e  -> Left e  
    Right x1 -> case eval e2 of  
        Left e  -> Left e  
        Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of  
    Left e  -> Left e  
    Right 0  -> Left DivByZero  
    Right x2 -> Right (eval e1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
    Left e  -> Left e
    Right x1 -> case eval e2 of
        Left e  -> Left e
        Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
    Left e  -> Left e
    Right 0  -> Left DivByZero
    Right x2 -> case eval e1 of
        Left e  -> Left e
        Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Either a b = Left a | Right b
```

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
  Left e  -> Left e
  Right x1 -> case eval e2 of
    Left e  -> Left e
    Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
  Left e  -> Left e
  Right 0  -> Left DivByZero
  Right x2 -> case eval e1 of
    Left e  -> Left e
    Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
    Left e  -> Left e
    Right x1 -> case eval e2 of
        Left e  -> Left e
        Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
    Left e  -> Left e
    Right 0  -> Left DivByZero
    Right x2 -> case eval e1 of
        Left e  -> Left e
        Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
  Left e  -> Left e
  Right x1 -> case eval e2 of
    Left e  -> Left e
    Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
  Left e  -> Left e
  Right 0 -> Left DivByZero
  Right x2 -> case eval e1 of
    Left e  -> Left e
    Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = case eval e1 of
  Left e  -> Left e
  Right x1 -> case eval e2 of
    Left e  -> Left e
    Right x2 -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
  Left e  -> Left e
  Right 0  -> Left DivByZero
  Right x2 -> case eval e1 of
    Left e  -> Left e
    Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = e1 >>= \x1 -> case eval e2 of
                                   Left e    -> Left e
                                   Right x2  -> Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
                    Left e    -> Left e
                    Right 0   -> Left DivByZero
                    Right x2  -> case eval e1 of
                                   Left e    -> Left e
                                   Right x1  -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 ->
                    Right(x1 / x2)
```

```
eval (Div e1 e2) = case eval e2 of
                    Left e    -> Left e
                    Right 0   -> Left DivByZero
                    Right x2  -> case eval e1 of
                                    Left e    -> Left e
                                    Right x1  -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 ->
                    Right(x1 / x2)
```

```
eval (Div e1 e2) = eval e2 >>= \x2 ->
```

```
    case x2 of
```

```
        0 -> Left DivByZero
```

```
        x2 -> case eval e1 of
```

```
            Left e -> Left e
```

```
            Right x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 ->  
                    Right(x1 / x2)
```

```
eval (Div e1 e2) = eval e2 >>= \x2 ->
```

```
    case x2 of
```

```
        0 -> Left DivByZero
```

```
        x2 -> eval e1 >>= \x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = Right x
```

```
eval (Add e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 ->  
                    Right(x1 / x2)
```

```
eval (Div e1 e2) = eval e2 >>= \x2 ->  
                    case x2 of  
                        0 -> Left DivByZero  
                        x2 -> eval e1 >>= \x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = eval e1 >>= \x1 -> eval e2 >>= \x2 ->  
                    Right(x1 / x2)
```

```
eval (Div e1 e2) = eval e2 >>= \x2 ->  
                    case x2 of  
                        0 -> Left DivByZero  
                        x2 -> eval e1 >>= \x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    return (x1 / x2)
```

```
eval (Div e1 e2) = eval e2 >>= \x2 ->  
                  case x2 of  
                    0 -> Left DivByZero  
                    x2 -> eval e1 >>= \x1 -> Right(x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    case x2 of  
                      0 -> Left DivByZero  
                      x2 -> return (x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

Error handling in Haskell

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    case x2 of  
                      0 -> throw DivByZero  
                      x2 -> return (x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
throw = Left
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

The Either Monad

- User extensible
- Hierarchical
- Checked
- Explicit

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                    x2 <- eval e2
                    return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1
                    x2 <- eval e2
                    case x2 of
                        0 -> throw DivByZero
                        x2 -> return (x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

```
throw = Left
```

```
instance Error e => Monad (Either e) where
```

```
Left e >>= f = Left e
```

```
Right x >>= f = f x
```

```
return = Right
```

The Either Monad

- User extensible
- Hierarchical
- Checked
- Explicit

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either DivByZero Double
```

```
eval (Val x)      = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1  
                    x2 <- eval e2  
                    case x2 of  
                      0  -> throw DivByZero  
                      x2 -> return (x1 / x2)
```

```
data DivByZero = DivByZero deriving Show
```

The Either Monad

- User extensible
- Hierarchical
- Checked
- Explicit



```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either Exception Double
```

```
eval (Val x)      = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                    x2 <- eval e2
                    return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1
                    x2 <- eval e2
                    case x2 of
                        0  -> throw DivByZero
                        x2 -> return (x1 / x2)
```

```
data DivByZero    = DivByZero deriving Show
```

```
data AddOverflow  = AddOverflow deriving Show
```

```
type Exception    = Either DivByZero AddOverflow
```

The Either Monad

- User extensible ✗
- Hierarchical ✗
- Checked
- Explicit

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either Exception Double
```

```
eval (Val x)      = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                     x2 <- eval e2
                     return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1
                     x2 <- eval e2
                     case x2 of
                       0  -> throw DivByZero
                       x2 -> return (x1 / x2)
```

```
data DivByZero    = DivByZero deriving Show
```

```
data AddOverflow  = AddOverflow deriving Show
```

```
type Exception    = Either DivByZero AddOverflow
```

The Either Monad

- User extensible ✗
- Hierarchical ✗
- Checked
- Explicit

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either Exception Double
```

```
eval (Val x)      = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                      x2 <- eval e2
                      return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1
                      x2 <- eval e2
                      case x2 of
                        0  -> throw DivByZero
                        x2 -> return (x1 / x2)
```

```
data DivByZero    = DivByZero deriving Show
```

```
data AddOverflow  = AddOverflow deriving Show
```

```
type Exception    = Either DivByZero AddOverflow
```

```
runEval x y = eval x y `catch` \Left DivByZero -> return 0
```

```
catch (Right x) h = return x
```

```
catch (Left e)  h = h e
```

The Either Monad

- User extensible ✗
- Hierarchical ✗
- Checked ✓
- Explicit

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either Exception Double
```

```
eval (Val x) = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                    x2 <- eval e2
```

Warning: Pattern match(es) are non-exhaustive
 In a case alternative:
 Patterns not matched:
 Left (Right AddOverflow)

```
data DivByZero = DivByZero deriving Show
```

```
data AddOverflow = AddOverflow deriving Show
```

```
type Exception = Either DivByZero AddOverflow
```

```
runEval x y = eval x y `catch` \Left DivByZero -> return 0
```

```
catch (Right x) h = return x
```

```
catch (Left e) h = h e
```

The Either Monad

- User extensible ✗
- Hierarchical ✗
- Checked ✓
- Explicit ✗

```
data Expr = Add Expr Expr | Div Expr Expr | Val Double
```

```
eval :: Expr -> Either Exception Double
```

```
eval (Val x)      = return x
```

```
eval (Add e1 e2) = do x1 <- eval e1
                     x2 <- eval e2
                     return (x1 / x2)
```

```
eval (Div e1 e2) = do x1 <- eval e1
                     x2 <- eval e2
                     case x2 of
                       0  -> throw DivByZero
                       x2 -> return (x1 / x2)
```

```
data DivByZero    = DivByZero deriving Show
```

```
data AddOverflow  = AddOverflow deriving Show
```

```
type Exception    = Either DivByZero AddOverflow
```

```
runEval x y = eval x y `catch` \Left DivByZero -> return 0
```

```
catch (Right x) h = return x
```

```
catch (Left e)  h = h e
```

Fixing the Error monad

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

```
class Exception e where  
  fromException :: SomeException -> e  
  toException   :: e -> SomeException
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

```
class Exception e where  
  fromException :: SomeException -> e  
  toException   :: e -> SomeException
```

```
newtype EM a = EM(Either SomeException a) deriving Monad
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

```
class Exception e where  
  fromException :: SomeException -> e  
  toException   :: e -> SomeException
```

```
newtype EM a = EM(Either SomeException a) deriving Monad
```

```
throw e = EM . Left . toException
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

```
class Exception e where  
  fromException :: SomeException -> e  
  toException   :: e -> SomeException
```

```
newtype EM a = EM(Either SomeException a) deriving Monad
```

```
throw e = EM . Left . toException  
catch m h = ...
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

class

from
toEx

newtype

throw
catch

- User extensible ✓
- Hierarchical ✓
- Checked
- Explicit

```
data SumOverflow = SumOverflow
```

```
data DivByZero = DivByZero
```

```
data ArithException = ArithException
```

```
instance Exception DivOverflow
```

```
instance Exception SumOverflow
```

```
.....
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

class

from
toEx

newtype

throw
catch

- User extensible ✓
- Hierarchical ✓
- Checked ✗
- Explicit

```
data SumOverflow = SumOverflow
```

```
data DivByZero = DivByZero
```

```
data ArithException = ArithException
```

```
instance Exception DivOverflow
```

```
instance Exception SumOverflow
```

```
.....
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

class

from
toEx

newtype

throw
catch

- User extensible ✓
- Hierarchical ✓
- Checked ✗
- Explicit ✗

```
data SumOverflow = SumOverflow
```

```
data DivByZero = DivByZero
```

```
data ArithException = ArithException
```

```
instance Exception DivOverflow
```

```
instance Exception SumOverflow
```

```
.....
```

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Fixing the Error monad

- Apply the extensible + hierarchical exceptions framework of Marlow*

```
data SomeException =  $\forall$  e. (Show e, Typeable e) => e -> SomeException
```

class

from
toEx

newtype

throw
catch

- User extensible ✓
- Hierarchical ✓
- Checked ✗
- Explicit ✗

```
data SumOverflow = SumOverflow
data DivByZero  = DivByZero
```

```
data ArithException = ArithException
```

```
instance Exception DivOverflow
instance Exception SumOverflow
```

.....

- Due to the existential encoding we give up checking !

* *An extensible dynamically typed hierarchy of Exceptions* - S. Marlow (2006)

Explicitly Typed Exceptions

Our fix: exceptions in the types

- We want to recover checking + make exceptions explicit
- Approach: encode exceptions in the type system using a phantom type argument

```
throw :: e -> EM a
```

```
newtype EM a = EM(Either Exception a)
```

Really fixing the Error monad

- We want to recover checking + make exceptions explicit
- Approach: encode exceptions in the type system using a phantom type argument

```
throw :: Throws e l => e -> EM l a
```

```
newtype EM l a = EM(Either Exception a)
```

Really fixing the Error monad

- We want to recover checking + make exceptions explicit
- Approach: encode exceptions in the type system using a phantom type argument

```
throw :: Throws e l => e -> EM l a  
throw = EM . Left . toException
```

```
newtype EM l a = EM(Either Exception a)
```

```
class Exception e => Throws e l
```

Really fixing the Error monad

- We want to recover checking + make exceptions explicit
- Approach: encode exceptions in the type system using a phantom type argument

```
throw :: Throws e l => e -> EM l a  
throw = EM . Left . toException
```

```
newtype EM l a = EM(Either Exception a)
```

```
class Exception e => Throws e l
```

```
catch :: Exception e => EM (Caught e l) a -> (e -> EM l a) -> EM l a  
catch = ... (as before)
```

```
data Caught e l  
instance Throws e (Caught e l)  
instance Throws e l => Throws e (Caught e1 l)
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
instance Exception Exception1
data Exception2 = ...
...

f1 x = throw Exception1
f2 x = throw Exception2
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
```

```
instance Exception Exception1
```

```
data Exception2 = ...
```

```
...
```

```
f1 x = throw Exception1
```

```
f2 x = throw Exception2
```

```
> :t f1
```

```
f1 :: Throws Exception1 l => a -> EM l b
```

```
f3 0 = f1 0 `catch` \Exception1 -> return 0
```

```
> :t f3 :: Int -> EM l Int
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
instance Exception Exception1
data Exception2 = ...
...

f1 x = throw Exception1
f2 x = throw Exception2

> :t f1
f1 :: Throws Exception1 l => a -> EM l b

f3 0 = f1 0 `catch` \Exception1 -> return 0
f3 1 = f2 0

> :t f3 :: Throws Exception2 l => Int -> EM l Int
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
```

```
instance Exception Exception1
```

```
data Exception2 = ...
```

```
...
```

```
f1 x = throw Exception1
```

```
f2 x = throw Exception2
```

```
> :t f1
```

```
f1 :: Throws Exception1 l => a -> EM l b
```

```
f3 0 = f1 0 `catch` \Exception1 -> return 0
```

```
f3 1 = f2 0
```

```
> :t f3 :: Throws Exception2 l => Int -> EM l Int
```

```
runEM :: EM l a -> a
```

```
runEM (EM (Right a)) = a
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
```

```
instance Exception Exception1
```

```
data Exception2 = ...
```

```
...
```

```
f1 x = throw Exception1
```

```
f2 x = throw Exception2
```

```
> :t f1
```

```
f1 :: Throws Exception1 l => a -> EM l b
```

```
f3 0 = f1 0 `catch` \Exception1 -> return 0
```

```
f3 1 = f2 0
```

```
> :t f3 :: Throws Exception2 l => Int -> EM l Int
```

```
runEM :: EM l a -> a
```

```
runEM (EM (Right a)) = a
```

```
> :t runEM (f3 0)
```

Programming with Explicit Exceptions

```
data Exception1 = Exception1 deriving ...
```

```
instance Exception Exception1
```

```
data Exception2 = ...
```

```
...
```

```
f1 x = throw Exception1
```

```
f2 x = throw Exception2
```

```
> :t f1
```

```
f1 :: Throws Exception1 l => a -> EM l b
```

```
f3 0 = f1 0 `catch` \Exception1 -> return 0
```

```
f3 1 = f2 0
```

```
> :t f3 :: Throws Exception2 l => Int -> EM l Int
```

```
runEM :: EM l a -> a
```

```
runEM (EM (Right a)) = a
```

```
> :t runEM (f3 0)
```

```
No instance for (Throws Exception2 l)
```

```
Possible fix:
```

```
add an instance declaration for (Throws Exception2 l)
```

Dealing with hierarchical exceptions

Dealing with hierarchical exceptions

- The Top of the hierarchy is SomeException

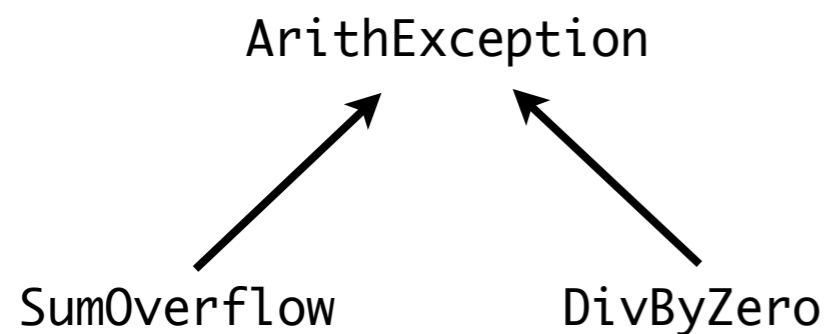
`instance` Throws e (Caught SomeException l)

Dealing with hierarchical exceptions

- The Top of the hierarchy is `SomeException`

```
instance Throws e (Caught SomeException l)
```

- User hierarchies can be encoded with `Throws` instances



```
instance Throws SumOverflow (Caught ArithException l)
```

```
instance Throws DivByZero (Caught ArithException l)
```

A simple semantics

$$M, N = \begin{array}{l} \text{return } H \\ | \\ M \gg N \\ | \\ \text{throw } e \\ | \\ \text{catch } M (\lambda e \rightarrow N) \\ | \\ \text{if } H \text{ then } M \text{ else } N \end{array} \quad H \in \text{pure}$$

$$\boxed{|M| = e^2}$$

$$|\text{return } H| = \{\}$$

$$|\text{throw } e| = \{e\}$$

$$\frac{|M| = m \quad |N| = n}{|M \gg N| = m \cup n}$$

$$\frac{|M| = m \quad |N| = n}{|\text{catch } M(\lambda e \rightarrow N)| = (n \setminus \{e\}) \cup n}$$

$$\frac{|M| = m \quad |N| = n}{|\text{if } H \text{ then } M \text{ else } N| = m \cup n}$$

Conclusion

- Never feel embarrassment anymore !
- We introduce an encoding of checked and explicit exceptions which
 - have a simple semantics
 - impose no runtime costs
- Not Haskell 98
 - Multi Param.Type Classes + Overlapping Instances
- The `control-monad-exception` library is available in Hackage