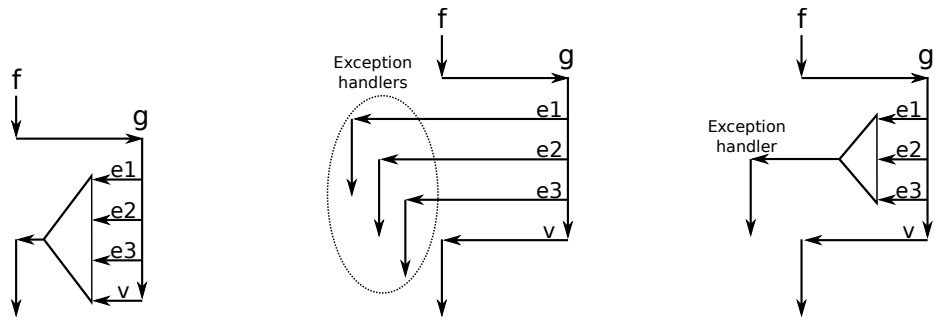
First-Class Control-Flow in Haskell

Sylvain Henry @ Haskus
sylvain@haskus.fr

Static checking of control-flow is important to ensure code correctness and ease of maintenance. In most programming languages using exceptions, however, exceptional function output paths are either not statically checked or hard to deal with (cf Java's checked exceptions and Either-like sum types in functional programming).

In this paper, we propose an Haskell approach using GHC's type extensions and relying on an open sum type that is both statically checked and easy to work with. To back this claim, we show that codes using this approach don't have to declare any boilerplate type class, type class instance nor error multiplexing sum type.

We show that it is possible to create generic methods supporting "control-flow polymorphism", i.e. taking as parameters or returning functions with several output paths/values and working with them in a generic way.



(a) *Multiplexed returned value* **(b)** *Implicit exception handlers* **(c)** *Multiplexed exceptions with single implicit exception handler*

Figure 1: *Handling exit points*

1 Introduction

Exceptional control-flow paths, which are only used when errors occur, are often not very well integrated into programming languages, even those with advanced type-systems like Haskell. The different approaches to handle functions with multiple exit points, hence different exit value types, are:

1. Use a sum type to multiplex the different possible exit values into a single value that is returned by the function.

2. Use an exception mechanism to implicitly pass the address of a handler for each exceptional exit point type.

3. Use a mix of both approaches: the different exceptional exit values are multiplexed into a single value and the address of a single exception handler able to handle this value is implicitly passed.

Figures 1a, 1b and 1c show respectively the three methods on a simple example: a function `f` calling a function `g` which has 4 exit points. The three first exit points are failures and they respectively return values of type `e1`, `e2` and `e3`. The last exit point is the "correct" one and it returns a value of type `v`.

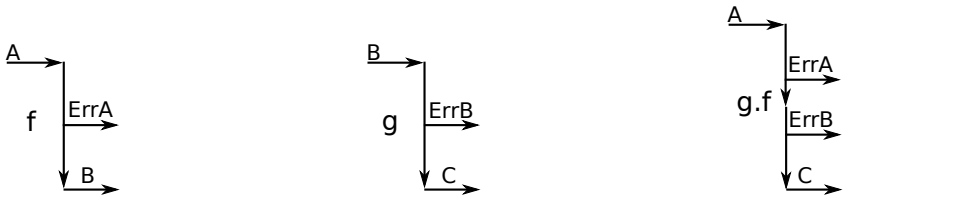
Methods 2 and 3 use implicit function handlers. When the types of the early exit points are indicated in the function prototype (e.g., `e1`, `e2` and `e3` for the function `g`), we say that exceptions are checked. Otherwise, they are unchecked.

Most programming languages provide unchecked exceptions. For instance in Haskell, to catch the exceptions raised by `g`, `f` would have to call: `catch g handler` where `handler` is the exception handler. The type of `catch` is:

```
catch ::  Exception e => m a -> (e -> m a) -> m a
```

You can observe that the `e` parameter comes out of the blue: the compiler and the programmer have no way to know from the type of the first parameter that it may raise an exception.

Among the programming languages, a notorious exception (!) is the Java language which provides both unchecked and checked exceptions. The latter, however, have been debated at lot [3, 5]. Mainly because they lack flexibility especially with polymorphic functions: the type



(a) *Function* `f` **(b)** *Function* `g` **(c)** *Composition of* `f` *and* `g`

Figure 2

Listing 1: multiplexing exit points

```
data E2 a b   = E2a a | E2b b
data E3 a b c = E3a a | E3b b | E3c c

(>>>>) :: (a -> E2 ea b) -> (b -> E2 eb c) -> a -> E3 ea eb c
(>>>>) f g a = case f a of
   E2a ea -> E3a ea
   E2b b  -> case g b of
      E2a eb -> E3b eb
      E2b c  -> E3c c

f :: A -> E2 ErrA B
g :: B -> E2 ErrB C

h :: A -> E3 ErrA ErrB C
h = f >>>> g
```

End of Listing 1

checker of the Java compiler misses some features needed to abstract over the list of checked exceptions that a function can throw [1, 4].

Thanks to GHC's type-system extensions, we show that we can have flexible checked exceptions in Haskell. Our system doesn't rely on GHC's unchecked exception machinery, but builds on the first method presented above (i.e., using a sum type to multiplex the values returned by a function).

Our contributions are:

- We explain the limitations of the current approaches using multiplexing (Section 2);
- We show how to define an efficient open sum type (a typed *Variant*) to circumvent these limitations (Section 3);
- We show how to use this Variant type to represent control-flow and we present several operators useful in this context (Section 4).

2 Multiplexing: Current Approaches

Suppose we want to compose two functions `f` and `g`, respectively shown on Figure 2a and Figure 2b, to get the function on Figure 2c.

To do it in Haskell with the multiplexing approach, we could define some sum types (`E2` and `E3`) and a composition operator `(>>>>)` as shown in Listing 1. The operator takes two functions

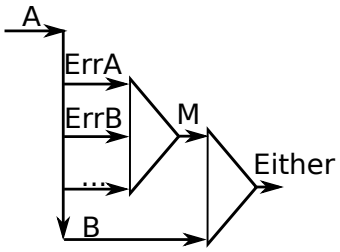


Figure 3: *Multiplexing exceptional exit points.* `M` *is an ad-hoc sum type used to multiplex exceptional exit points.* `Either` *multiplexes* `M` *and the correct output type* `B`*.*

with two exit points and compose them to form a single function `h` with three exit points. We can know which exit point has been taken by pattern-matching on the sum types `E2` and `E3`.

This approach has an issue: we cannot use our composition operator `(>>>>)` to compose `h` with another function because our operator only supports functions with two exit points and `h` has three of them. We could define the 3 operators necessary to compose every combination of functions with two or three exit points along with the `E4` and `E5` sum types. This approach, however, doesn't scale. We would need to define `n Ei` data types for each number of paths `i` we want to support and n^2 composition operators.

2.1 Multiplexer Sum Type

A common solution to the previous issue is to multiplex the different exceptional exit points into a single data type before returning from a function. Instead of defining generic `Ei` sum types, the application has to define its own ad-hoc sum type `M` for the exceptional exit points. Finally, the sum type `E2` is used to multiplex the exceptional exit points represented by `M` and the "correct" value. Figure 3 illustrates this method.

Our `E2` sum type is isomorphic to the common `Either` sum type. You can directly use this approach and compose functions returning `Either M a` (where `M` is the same for all functions) with the `ExceptT` monad transformer [2] (or the older `EitherT` monad transformer [7]).

The composition mechanism is the same as the one we have used for the `(>>>>)` operators and it is represented on Figure 4. Note that if both functions multiplex the same value in `M`, we cannot distinguish if it comes from the first or the second function of the composition in the resulting value.

The main issue of this approach is that all the functions we want to compose must *share the same multiplexer sum type* `M` and that this data type is ad-hoc to the functions we want to compose:

- if it multiplexes too many data types, we can use it for more compositions but we don't know if a given composition may actually return a value of this type. For instance, if `M` can multiplex IO errors such as `ReadError` and `WriteError`, and if we only compose reading functions, we somehow have to handle the `WriteError` case to avoid compiler warnings, even if we know this error currently never occurs. It has a real maintenance cost:
  - if we handle it carefully the first time, it is extraneous work that may never be useful in practice (dead code).
  - if we handle it lightly to avoid the compiler warning (e.g., by triggering a runtime error in the WriteError case), then if we modify the composition to add writing functions, the compiler won't warn us that the case is not really handled.

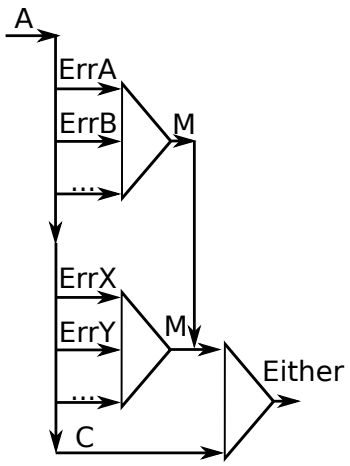


Figure 4: *Composing two functions with* `Either`.

- if it multiplexes only a few data types, when we want to compose with a function producing a value with an unsupported data type, we need to refactor our code. We can add a data constructor to `M`: in this case, the previous code using `M` may need to be modified to handle it (to avoid compiler warnings). We can create a new multiplexer type: in this case we cannot easily compose with functions using `M`.

2.2 Late Multiplexer Binding

A workaround to alleviate the previous issues is to keep the type of the multiplexer parametric and to only fix it when all the functions have been composed. Each function adds type-class constraints to the parametric type of the multiplexer, and GHCs fusions them during the composition.

Listing 2 shows an example using this technique. For each possible value in the multiplexed type, a type-class is defined (here `TErrA` and `TErrB`). These type-classes provide a method to create a multiplexed type from a value and they are used as constraint in the type of `f` and `g`. We can observe that GHC infers the union of the constraints for `h` which composes `f` with `g`. Finally, an effective multiplexer type `MyError` is defined with instances for the constraint it needs to support and `h'` is `h` using `MyError` as a multiplexer type.

We still need to define an ad-hoc multiplexer data type for each composition.

3 An Open Sum Type: Variant

The initial issue in Section 2 was that we couldn't use a single sum type to represent computations with different number of exit points. By using GHC's type extensions, now we can do it and we can provide a simple and non invasive interface to manipulate them. We call our open sum type a *Variant*.

We want our Variant data type to have the same cost as other sum types (as implemented by GHC), hence we use a similar memory representation (cf Listing 3). As indicated by the kind signature (`types :: [*]`), a Variant is parameterized by a list of types. Then a `Word` value is used as a tag to index into the `types` list in order to know the effective type of the stored `a`.

To declare a variant type, we can use the type list notation as shown in following example. In this example, `v` is variant that can contain either a value of type `B`, `ErrA` or `ErrB`.

Listing 2: late multiplexer binding

```
class TErrA e where throwErrA :: ErrA -> e
class TErrB e where throwErrB :: ErrB -> e
class TErrC e where throwErrC :: ErrC -> e

-- f uses throwErrA from the TErrA class to wrap its ErrA error
-- similarly, g uses throwErrB from the TErrB class
f :: TErrA e => A -> Either e B
g :: TErrB e => B -> Either e C

-- the union of the constraints is automatically inferred by GHC
h :: (TErrA e, TErrB e) => A -> Either e C
h = f >>>> g

-- an ad-hoc error type that supports both ErrA and ErrB but not ErrC
data MyError = MyErrA ErrA | MyErrB ErrB
instance TErrA MyError where throwErrA = MyErrA
instance TErrB MyError where throwErrB = MyErrB

-- we only state the error type at the use site
h' :: A -> Either MyError C
h' = h
```

End of Listing 2

Listing 3: variant data type

```
-- A variant contains a single value whose type is in the "types" type-list.
-- The Word field contains a tag (i.e. an index into the types list) and "a"
-- is the actual value
data Variant (types :: [*]) = forall a. Variant Word a
```

End of Listing 3

Listing 4: variant primitives

```
-- | Set the value with the given indexed type
setVariantN :: forall (n :: Nat) (l :: [*]). (KnownNat n)
   => Proxy n -> TypeAt n l -> Variant l
setVariantN _ = Variant (fromIntegral (natVal (Proxy :: Proxy n)))

-- | Get the value if it has the indexed type
getVariantN :: forall (n :: Nat) (l :: [*]). (KnownNat n)
   => Proxy n -> Variant l -> Maybe (TypeAt n l)
getVariantN _ (Variant t a) = do
   guard (t == fromIntegral (natVal (Proxy :: Proxy n)))
   return (unsafeCoerce a) -- we know it is the effective type

-- | Indexed access into a type list: retrieve the type at index n in the type
-- list l
type family TypeAt (n :: Nat) (l :: [*]) where
   TypeAt 0 (x ': xs) = x
   TypeAt n (x ': xs) = TypeAt (n-1) xs
```

End of Listing 4

```
v :: Variant '[B,ErrA,ErrB]
```

Note: another way to implement an open sum type is to use a nest of `Either` data types (with a GADT). The complexity in time and memory, however, is linear in the number of types in the sum as we need to traverse the `Either` nest each time we want to access the actual value in the `Variant`. The advantage of this approach, however, is to avoid an "unsafe" coercion from the stored generic `a` type to the effective type, hence it could be useful for a "Safe Haskell" implementation.

3.1 Using Variants

To get and set the value of a `Variant`, we need to specify a type index: an index into the types supported by the `Variant`. This index is a `Nat`: a natural number at the type level. Listing 4 shows these two fundamental `Variant` primitives.

In addition, by using type-classes, we can fold over the types of a `Variant` (or over a list of indexes) and produce a resulting value. For instance, it allows us to implement the `Show` instance for `Variant`. We fold over the indexes of the types in the list and use `getVariantN` until we find the actual value type. Then we can use the `Show` instance of the value type to show the actual value.

```
> let v = setVariantN (Proxy :: Proxy 1) 10 :: Variant '[Char,Int,String]
> v
10
> let v = setVariantN (Proxy :: Proxy 2) "Hey" :: Variant '[Char,Int,String]
> v
"Hey"
> getVariantN (Proxy :: Proxy 1) v
Nothing
> getVariantN (Proxy :: Proxy 2) v
Just "Hey"
```

We can also index a Variant with a type (the index of the first matching type in the type list is used):

```
> let v = setVariant (10 :: Int) :: Variant '[Char,Int,String]
> v
10
> let v = setVariant "Hey" :: Variant '[Char,Int,String]
> v
"Hey"
> getVariant v :: Maybe Int
Nothing
> getVariant v :: Maybe String
Just "Hey"
```

We can also extract a value from a Variant and get either the value or a new Variant (which type is automatically inferred) :

```
> let v = setVariant "Hey" :: Variant '[Char,Int,String]
> v
"Hey"
> :set -XPartialTypeSignatures
> catchVariant v :: Either _ String

<interactive>:28:26: Warning:
    Found hole _ with type: Variant '[Char, Int]

Right "Hey"
```

We can convert a Variant into an heterogeneous list or into a tuple:

```
> let v = setVariant "Hey" :: Variant '[Char,Int,String]
> v
"Hey"
> :t variantToHList v
variantToHList v :: HList '[Maybe Char, Maybe Int, Maybe [Char]]
> variantToHList v
H[Nothing,Nothing,Just "Hey"]
> :t variantToTuple v
variantToTuple v :: (Maybe Char, Maybe Int, Maybe [Char])
> variantToTuple v
(Nothing,Nothing,Just "Hey")
```

We can lift a Variant into another Variant whose type list is a superset of the input variant's type list:

```
> let v = setVariant "Hey" :: Variant '[Char,Int,String]
> v
"Hey"
> liftVariant v :: Variant '[Int,Double,Word,String,Float,Char]
"Hey"
```

In conclusion, the Variant type is an open sum type that we can use to multiplex several data types.

4 Using Variant for Control-Flow

A function with several exit points is a function that returns a Variant. We define the following Flow type alias to get cleaner function types (we add a type parameter `m` to support monadic functions).

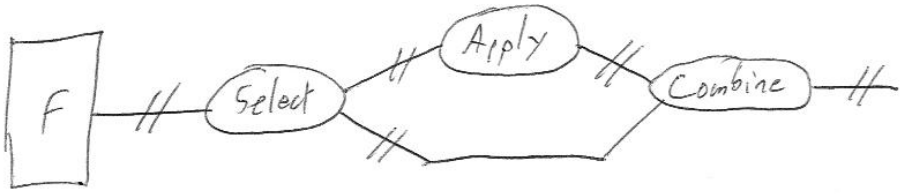


Figure 5: *General overview of a flow composition*

```
type Flow m (l :: [*]) = m (Variant l)
```

To create a flow, we can just return a Variant or use the following helpers:

```
-- | Return in the first position
flowRet0 :: Monad m => x -> Flow m (x ': xs)

-- | Return in the second position
flowRet1 :: Monad m => x -> Flow m (y ': x ': xs)

-- | Return a single element
flowRet0' :: Monad m => x -> Flow m '[x]

-- | Return in the first type-matching position
flowSet :: (Member x xs, Monad m) => x -> Flow m xs
```

To get a value out of a flow, we can use the Variant interface or the following helper. The latter statically ensures that we don't forget to handle a case (i.e., the flow has a single result type).

```
-- | Extract single flow result
flowRes :: Functor m => Flow m '[x] -> m x
```

We can lift a Flow into another with the following helper:

```
flowLift :: (Liftable xs ys , Monad m) => Flow m xs -> Flow m ys
```

4.1 Flow Composition

We want to compose functions returning variants (or "flows") in various ways. By convention, we say that the first type in a Variant returned by a function (the type at index 0) is the type of the "correct" value while the other types are exceptional (errors, etc.). This is just a convention and we can totally define operators that don't suppose this bias.

The general overview of the flow composition is given in Figure 5. If we have a Flow "f" (that is a function that returns a Variant): first we need to *select* some of the output cases, then to *apply* something in these cases and finally to *combine* the result with the unselected cases. The composition results in a new Flow.

4.1.1 Selection

A selection operator has the following type, where `xs` are the selected types in `fs` and `ys` the unselected ones:

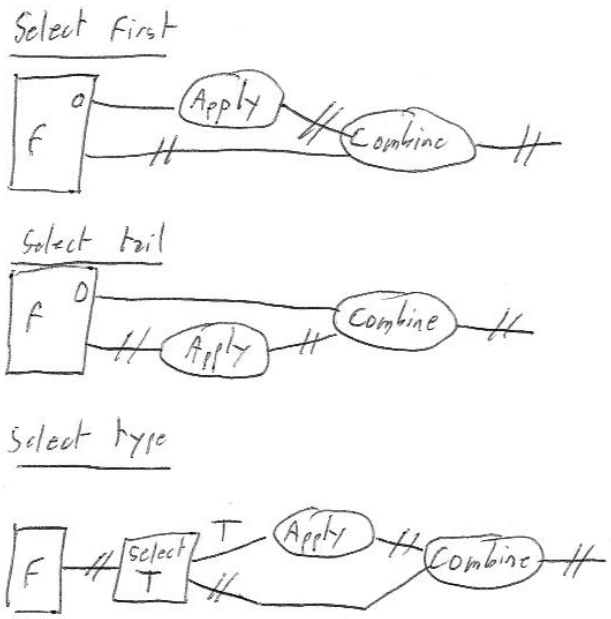


Figure 6: *Selection operators*

```
select :: Variant fs -> Either (Variant ys) (Variant xs)
```

With the convention stated above, usage has shown that we mostly need 4 selection operators: select first, select tail, select by type and select by type in tail (see Figure 6). The latter is just a composition of other composition operators, hence it is not shown here.

```
-- | Select the first value
selectFirst :: Variant (x ': xs) -> Either (Variant xs) (Variant '[x])

-- | Select the tail
selectTail :: Variant (x ': xs) -> Either (Variant '[x]) (Variant xs)

-- | Select by type
selectType ::
   (Catchable x xs
   ) => Variant xs -> Either (Variant (Filter x xs)) (Variant '[x])
```

4.1.2 Combination

Combination operators basically have the following type:

```
combine :: Either (Variant ys) (Variant xs) -> Variant zs
```

For a flow `f`, we select some of the output cases and apply a function (see next section) to get a result type in `xs`, hence a `Variant xs`; unselected output cases of `f` give us a `Variant ys`. As there is only one valid output case at a time, we get either a `Variant xs` or a `Variant ys` that we can combine to get a new Variant.

Each operator has different `xs`, `ys` and `zs` and different constraints between them.

```
-- | Set the first value (the "correct" one)
combineFirst :: Either (Variant xs) (Variant '[x]) -> Variant (x ': xs)
```

```haskell
-- | Set the first value, keep the same tail type
combineSameTail :: Either (Variant xs) (Variant (x ': xs)) -> Variant (x ': xs)

-- | Return the valid variant unmodified
combineEither :: Either (Variant xs) (Variant xs) -> Variant xs

-- | Concatenate unselected values
combineConcat :: Either (Variant ys) (Variant xs) -> Variant (Concat xs ys)

-- | Union
combineUnion :: Either (Variant ys) (Variant xs) -> Variant (Union xs ys)

-- | Lift unselected
combineLiftUnselected ::
   ( Liftable ys xs
   ) => Either (Variant ys) (Variant xs) -> Variant xs

-- | Lift both
combineLiftBoth ::
   ( Liftable ys zs
   , Liftable xs zs
   ) => Either (Variant ys) (Variant xs) -> Variant zs
```

4.1.3 Application

An application is just a function from a Variant to another:

```haskell
apply :: Variant xs -> Flow m ys
```

We distinguish several common application variants (!):

```haskell
-- | Const application
applyConst :: Flow m ys -> (Variant xs -> Flow m ys)

-- | Pure application
applyPure :: Monad m => (Variant xs -> Variant ys) -> Variant xs -> Flow m ys

-- | Lift a monadic function
applyM :: Monad m => (a -> m b) -> Variant '[a] -> Flow m '[b]

-- | Lift a monadic function
applyF :: Monad m => (a -> Flow m b) -> Variant '[a] -> Flow m b
```

4.2 Flow Operators

First we need to choose a selection operator **S**:

.	Select first
..	Select tail
%	Select by type
..%	Select by type in tail

Then we need to choose an apply mode `A`:

~	Flow
-	Pure function
~~	Flow (const variant)

Then we need to choose a combination operator **C**:

.	Combine set first
..	Combine set tail
+	Combine with concatenation
|	Combine with union
^	Combine by lifting unselected values
^^	Combine by lifting both unselected values and transformed selected values
$	Combine when unselected are result suffix (i.e., same tail)
!	Return an empty variant
!!	Return the variant of unselected values, if possible, otherwise fail
=	Passthrough the input variant

Finally we combine **S**, **A** and **C** to form an operator as follow: >**SAC**>. Operators should exist if they make sense, such as:

```
(>.~.>)   :: Flow m (a ': l) -> (a -> m x) -> Flow m (x ': l)
(>.~+>)   :: Flow m (a ': l) -> (a -> Flow m l2) -> Flow m (Concat l2 l)
(>.~~$>)  :: Flow m (a ': xs) -> Flow m (x ': xs) -> Flow m (x ': xs)
(>..~!!>) :: Flow m (x ': xs) -> (Variant xs -> m ()) -> m x
(>%~=>)   :: Catchable x xs => Flow m xs -> (x -> m ()) -> Flow m xs
(>..%~^>) :: (Catchable a xs , Liftable (Filter a xs) ys)
          => Flow m (x ': xs) -> (a -> Flow m ys) -> Flow m (x ': ys)
```

5 Examples

5.1 Error Management

See error management example in Listing 5. You can see that `retryBusy` and `tryBusyOrDie` support functions returning any kind of error as long as the Busy error is among them.

6 Conclusion

This paper has demonstrated that it is now possible to generalize sum type based methods for error handling (Either, etc.) with an open sum type. We have provided an implementation of the approach that is successfully used in a real project (see `www.vipervm.org`).

The proposed approach introduces new operators to deal with the variety of possible control-flow compositions. While there are a lot of them, we have tried to name them in a meaningful and consistent manner (still open to bikeshedding though). It makes them more easier to memorize too. As for any DSL, it takes some time to get used to them.

Finally Haskell may be the first mainstream language to have a kind of "checked exception" mechanism that doesn't impair abstraction and productivity.

Future work could include GHC parser extensions to make the syntax even easier to work with, similarly to the do-notation or the arrow-notation. Performance should be checked against ad-hoc data types.

Listing 5: example of generic error management combinators

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE FlexibleContexts #-}

import Prelude hiding (readFile)
import Control.Concurrent
import ViperVM.Utils.Flow
import ViperVM.Utils.HList
import ViperVM.Format.Binary.Buffer

data Busy         = Busy
data FileNotFound = FileNotFound
data NotAllowed   = NotAllowed String

readFile :: FilePath -> Flow IO '[Buffer,FileNotFound,Busy,NotAllowed]
readFile = undefined

-- | Retry at most n times while a resource is busy
retryBusy :: (Catchable Busy xs) => Int -> Flow IO xs -> Flow IO xs
retryBusy 0 f = f
retryBusy n f = f >%~$> \case
   Busy -> do
      threadDelay 1000
      retryBusy (n-1) f

-- | Die if a resource is busy
tryBusyOrDie ::
   ( Catchable Busy xs
   , Monad m
   ) => Flow m xs -> Flow m (Filter Busy xs)
tryBusyOrDie f = f >%~!!> \case
   Busy -> error "The resource is busy. We can't recover."

-- | readFile with "handled" Busy error
readFile2 :: FilePath -> Flow IO '[Buffer,FileNotFound,NotAllowed]
readFile2 = tryBusyOrDie . retryBusy 5 . readFile

-- | Convert any readFile error into Nothing
readFileMaybe :: FilePath -> IO (Maybe Buffer)
readFileMaybe f =
   readFile f
      >.-.> Just
      >..-.> const Nothing
      |> flowRes
```

End of Listing 5

7 Related Works

With some type-classes machinery, it is possible to provide an exception framework that is close to what we have. However, it is quite invasive. For instance, Iborra's framework [6] requires that computations are evaluated in the context of a specific monad transformer.

See also `https://hackage.haskell.org/package/control-monad-exception` and `https://www.well-typed.com/blog/2015/07/checked-exceptions/` for another approach using type classes.

MonadThrow: runtime exception, not checked.

```
catch ::  Exception e => m a -> (e -> m a) -> m a
```

Checked-Exceptions in Java

`https://wiki.haskell.org/Exception`

`https://www.schoolofhaskell.com/user/commercial/content/exceptions-best-practices`

`http://www.well-typed.com/blog/2015/07/checked-exceptions/`

References

[1] Robert Brautigam. A story of checked exceptions and java 8 lambda expressions. `https://dzone.com/articles/draft-a-story-of-checked-exceptions-and-java-8-lam`, 02 2016.

[2] Andy Gill and Ross Paterson. ExceptT monad transformer module. `https://hackage.haskell.org/package/transformers/docs/Control-Monad-Trans-Except.html`, 2016.

[3] Brian Goetz. Java theory and practice: The exceptions debate. `http://www.ibm.com/developerworks/library/j-jtp05254/`, 05 2004.

[4] Brian Goetz. Exception transparency in java. `https://blogs.oracle.com/briangoetz/entry/exception_transparency_in_java`, 06 2010.

[5] Misko Heveri. Checked exceptions i love you, but you have to go. `https://dzone.com/articles/checked-exceptions-i-love-you`, 09 2009.

[6] José Iborra. Explicitly typed exceptions for haskell. In *Practical Aspects of Declarative Languages*, pages 43–57. Springer, 2010.

[7] Edward A. Kmett. Either monad transformer package. `https://hackage.haskell.org/package/either`, 2015.