

Explicitly Typed Exceptions for Haskell ^{*}

José Iborra

pepeiborra@gmail.com,

DSIC, Universidad Politecnica de Valencia, Spain

Abstract. We describe a monad for checked, explicitly typed exceptions, which provides as a simple Haskell library what for other languages is a native feature. Multi parameter type classes and overlapping instances are the only essential extensions to Haskell 98 required.

1 Introduction

Even well-typed programs may on occasions fail. Error handling is a time-consuming programming task as for every function call, the programmer must write code to check whether the result is an error and handle it appropriately. As our running example we will use a tiny interpreter of arithmetic addition and division.

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

```
eval0 :: Expr -> Double
eval0 (Val x)      = x
eval0 (Add e1 e2) = eval0 e1 + eval0 e2
eval0 (Div e1 e2) = eval0 e1 / eval0 e2
```

```
main0 :: Double
main0 = eval0 (Div (Val 6.0) (Val 2.0))
```

When done naively, error handling results in a cascade of nested if/case expressions which obscure the essential intention of the code. This is demonstrated in Figure 1 by extending the interpreter to check for division by zero.

Now every call to `eval1` can result in an error and the code must check whether this is the case before continuing. Because this requires a lot of extra boilerplate which ultimately obscures the essential intent of the code, most modern programming languages feature a mechanism based on the notion of *exceptions*. Arguably in Haskell the need is much less pressing, thanks to the widespread use of *monads* [13] to hide the noise introduced by error handling and even provide exceptions as a library. A monad is a parameterized type constructor `m` of computations that support the following two operations.

^{*} This work has been partially supported by the EU (FEDER) and Spanish MEC/MICINN under grant TIN2007-68093-C02-02, Generalitat Valenciana under grant Emergentes GV/2009/024, and UPV-VIDI grant 3249 PAID0607

```

data ArithError = DivByZero | ...
eval1 :: Expr -> Either ArithError Double
eval1 (Val x)      = Right x
eval1 (Add e1 e2) = case eval1 e1 of
    Left e  -> Left e
    Right x1 -> case eval1 e2 of
        Left e  -> Left e
        Right x2 -> Right(x1 + x2)
eval1 (Div e1 e2) = case eval1 e2 of
    Left e  -> Left e
    Right 0  -> Left DivByZero
    Right x2 -> case eval1 e1 of
        Left e  -> Left e
        Right x1 -> Right(x1 / x2)

main1 :: Either ArithError Double
main1 = case eval1 (Div (Val 6.0) (Val 2.0)) of
    Left DivByZero -> putStrLn "division by zero"
    Right v        -> print v

```

Fig. 1: eval with explicit error handling

```

return :: Monad m => a -> m a
>>=   :: Monad m => m a -> (a -> m b) -> m b

```

`return x` creates the unit computation that returns a value `x`, and `>>=` (pronounced bind) applies a monadic function to a computation creating a new computation. Monads are extremely useful to model a number of computational effects. For instance, `MonadError` is a standard extension to the `Monad` interface to model exceptions via two additional operations `throwError` and `catchError`.

```

throwError :: MonadError e m => e -> m a
catchError :: MonadError e m => m a -> (e -> m a) -> m a

```

Haskell includes the so-called *do notation*, syntactic sugar to simplify programming with monads. The use of bind is implicit in do notation; for instance, the expression `getContents >>= \x -> return (length x)` is written more conveniently as `do {x <- getContents; return (length x);}`. Using do notation and the `MonadError` operations we can rewrite `eval1` in monadic style successfully hiding the error handling noise, as seen in figure 1. For a number of reasons however, `MonadError` has never been very popular among Haskell programmers. We blame this to the fact that it is not easy to combine computations throwing different types of exceptions. One must introduce a new datatype to carry every possible exception type. Suppose we want to pair `eval2` with a parser for expressions which can throw a `ParseError`.

```

parseExpr :: MonadError ParseError m => String -> m Expr

```

```

eval2 :: MonadError ArithError m => Expr -> m Double
eval2 (Val x) = return x
eval2 (Add a1 a2) = do
    v1 <- eval2 a1
    v2 <- eval2 a2
    return (v1 + v2)

eval2 (Div a1 a2) = do
    v1 <- eval2 a1
    v2 <- eval2 a2
    if v2 == 0 then throwError DivByZero else return (v1 / v2)

main2 :: MonadError ArithError m => m Double
main2 = eval2 (Div (Val 6.0) (Val 2.0) )
        'catchError' \DivByZero -> putStrLn "division by zero"

```

Fig. 2: eval in the Error Monad

To combine parsing and evaluation, we are forced to introduce a new datatype modelling the sum of `ParseError` and `ArithError`, and then lift all the monadic operations to use this new error type.

```

data PError = ParseError ParseError | ArithError ArithError
liftParse :: MonadError PError m => String -> m Expr
liftParse = ...
liftEval  :: MonadError PError m => Expr -> m Double
liftEval  = ...

```

Aside from the extra boilerplate code and naming overhead, now the programmer is expected to handle *both* parse errors and arithmetic errors every time `catchError` is used in such a computation, even if the parsing stage has already been completed and thus parsing errors cannot arise anymore. In general, with this approach the programmer is expected by the compiler pattern match checker to handle every exception type in the sum, even in situations when it is known that certain kind of exceptions cannot arise anymore.

A more refined option would be to employ an extensible sum type, in the style of [8], to handle the combination of exceptions from different kinds of computations. While this option would help in reducing the amount of boilerplate code needed, the programmer would still be expected to handle every exception type in the sum, even the ones the computation cannot possibly throw.

Let us remark this point again before considering the next alternative. The programmer should be warned at compile time if the program does not handle all the possible exception types. No exception should be allowed to escape a program, since this constitutes an unhandled runtime error equivalent to a pattern matching error in a functional language or a null pointer exception in an imperative language. The Java programming language introduces checked

exceptions [3], providing exception coverage via exception-specific type annotations produced by the compiler, which result in type errors if exceptions are not eventually handled. The `MonadError` class provides exception coverage via the totality checking of pattern matching in the handlers, but this solution is strictly inferior since in Java one can handle a subset of the exceptions thrown by a computation, while in a `MonadError` computation a handler must have a case for every exception in the set, as shown in the previous example. Therefore we claim that the need to monolithically combine different exception types in `MonadError` amounts to giving up exception coverage.

Since the introduction of monadic IO, Haskell has also supported *native* exceptions inside the IO monad. These are not modeled by a monad; it is the actual runtime which handles them directly, as done traditionally in most programming languages. In Haskell native exceptions the `Exception` type is abstract and *fixed*, and the primitives `throwIO` and `catchIO` have the following signatures.

```
throwIO :: Exception -> IO a
catchIO :: IO a -> (Exception -> IO a) -> IO a
```

Native exceptions as shown above suffer from several shortcomings which limit their usefulness, including poor extensibility and null exception coverage.

In its current version (6.10), the Glasgow Haskell Compiler features a new library for extensible hierarchical exceptions [9], which solves the composability problem by introducing dynamically typed exceptions and exception handlers, but in the process it also gives up on exception coverage, as we will see in the next section.

1.1 Plan

In this paper, we examine one way to restore exception coverage, accomplished by computing the list of the exceptions a computation can throw and showing it in its type. The scheme is based in the extensible exceptions of Marlow [9], and employs type class constraints to track the exceptions. Moreover, the scheme is purely static and involves absolutely no performance penalties at runtime.

We summarise extensible exceptions in Section 2, then in Section 3 our scheme is introduced and applied to extensible exceptions in the IO monad. Section 4 takes advantage of the fact that there is nothing IO specific in our approach to provide generally useful explicit exceptions as a monad transformer. We discuss unchecked exceptions in Section 5 and mechanisms for preventing the user from overriding the scheme in Section 6. Section 7 concludes.

2 Extensible Hierarchical Exceptions

In [9], Marlow identifies several deficiencies including the lack of exception coverage in Haskell 98 native exceptions. The main deficiency pointed by Marlow is that since the `Exception` type is fixed to roughly a string type, programmers are forced to fall back to nasty hacks, e.g. serialization techniques, if they

want to have their own exception types. From there, Marlow constructs a list of desirable requirements, and designs an encoding of exceptions which satisfies them. The requirements can be summed up as *extensibility* and the ability to manipulate exceptions in sets or *hierarchically*. That is, in addition to an extensible exception type, one also wants it to be hierarchical; it is very natural to model exceptions as a hierarchy and define handlers which catch entire sets of exceptions. Unfortunately exception coverage is not included in Marlow's list of requirements.

In the following we summarize the essential aspects of the encoding, in order to make apparent why it gives up exception coverage.

Firstly the fixed `Exception` type is replaced by a class of types.

```
class (Typeable a, Show a) => Exception a where
```

Types which wish to instantiate this class need to also belong to the `Show` and `Typeable` classes. That is, they must be equipped with an operation to serialize a value to a string, and they must support dynamic typing (`Typeable` is the standard encoding of dynamic typing in Haskell, providing operations to reify a type as well as type casting). As an example, in order to make `ArithError` an instance of `Exception` we declare it as follows.

```
data ArithError = DivOverflow | ...      deriving (Show,Typeable)
instance Exception ArithError
```

Throwing and catching exceptions is straightforward, the only thing to note is that since exceptions are now dynamically typed, a type annotation is included to pin down the type of the handler.

```
throw DivOverflow 'catch' \(e :: ArithError) -> print e
```

Exceptions are boxed in the *existential* container `SomeException`, and the underlying implementation works with this single, fixed type. `SomeException` also serves as the root of the exception hierarchy: every exception type introduced by the user is a subclass, as we will see in section 3.2.

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

`throw` and `catch` handle the boxing and unboxing on top of the primitives provided for native exceptions in the implementation. They are defined (at least conceptually) as follows:

```
throw :: Exception e => e -> IO a
throw e = primThrow (SomeException e)

catch :: Exception e => IO a -> (e -> IO a) -> IO a
catch m handler = primCatch m h' where
  h' e = case cast e of
    Just e' -> handler e'
    Nothing -> throw e
```

```

newtype EIO l a = EIO {runEIO::IO a} deriving Monad

class Exception e => Throws e s

throwEIO :: (Throws e l, Exception e) => e -> EIO l a
throwEIO e = EIO (Control.Exception.throw e)

data Caught e l

instance Exception e => Throws e (Caught e l)
instance Throws e l => Throws e (Caught e1 l)
instance Exception e => Throws e (Caught SomeException l)

catchEIO :: Exception e =>
          EIO (Caught e l) a -> (e -> EIO l a) -> EIO l a

catchEIO (EIO action) h = EIO (primCatch action (runEIO . h))
  where primCatch = Control.Exception.catch

```

Fig. 3: The EIO Monad

where the function cast is part of the `Typeable` library for dynamic casts:

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

And now it should be clear why exception coverage is lost. Since the type checker does not know anything about the type of exceptions being thrown by a computation -and for good reason, as this is the core feature which allows the library to provide extensible exceptions- checking for completeness of pattern matching cannot help with coverage checking any more.

3 Explicitly Typed Exceptions

Our position is that the list of requirements given in [9] is missing two essential points.

- It should be possible to determine statically the exceptions a computation can throw.
- The compiler should check that every exception is eventually handled.

In this section we discuss a lightweight extension to the extensible exceptions framework in order to cover these two requirements. The main idea is to track the list of exceptions that a function can raise in its own type. This directly satisfies the first point above, but also the second one, at least indirectly, since now one can express what a *exception-free* computation is in the type system, and thus can construct a *run* function which accepts only exception-free computations.

In order to track the list of exceptions we will unsurprisingly be using a monad, the EIO monad of IO computations with explicit exceptions. Figure 3

contains all the code related to the `EIO` monad. `EIO` is declared as a newtype wrapper around `IO`¹, with an extra *phantom* [6] type parameter.

The `throwEIO` function is defined exactly in the same way as the `throw` primitive for extensible exceptions. But its type signature additionally attaches a `Throws` constraint to the type of the resulting computation. Let us see an example of the types returned by the Haskell compiler for expressions involving `throwEIO`.

```
> e1 = throwEIO DivByZero
e1 :: Throws ArithError 1 => EIO 1 a

> f x = if x == (0::Int) then throwEIO DivByZero else return x
f :: Throws ArithError 1 => Int -> EIO 1 Int
```

The crux of the approach is the addition of the `l` phantom type parameter, used to carry `Throws` constraints denoting the exceptions that can be thrown by the `EIO 1` computation. They are encoded as `Throws e 1` type constraints, where `Throws e 1` is a type class with no methods. It can be seen as a binary relation on types, although a more practical intuition is that it expresses a property of a computation `EIO 1`, namely the fact that it can throw an exception `e`.

Now, we need a definition of `catch` that, in addition to capturing the exception at run time, removes it from the set of constraints at type checking time. We introduce a datatype `Caught` with no constructors as a witness of the fact that an exception is captured and thus can be removed from the constraint set. The instances of `Throws` explain this story to the type checker. The first instance states that `Caught e 1` removes exception `e`, the second one states that any other exception `e1` remains, and the third one states that `SomeException` is the root of the hierarchy and capturing it removes all exceptions. As with `throwEIO`, `catchEIO` is nothing more than vanilla `catch` for extensible exceptions with only more structure at the type level.

Finally, the `runEIO` function executes a `EIO` computation `m` returning a plain `IO` computation. The typechecker ensures that `m` cannot fail with an uncaught exception. As an example, we can modify the code for `eval2` to replace the use of `throwError` by `throwEIO`, as shown in figure 4. Now the type of `main3` rightly states that it throws no exception, as opposed to the type of `main2` in the previous example.

If we try to run a computation with `eval3` without handling `ArithError`, we get a compile time error.

```
> :type runEIO (eval3 (...))
Error: No instance for (Throws ArithError 1)
```

The new encoding for explicitly typed exceptions does not affect the good compositionality features of Marlow's exceptions. For example, we can easily combine a parsing function with our evaluator.

¹ and hence is a monad too by construction

```

eval3 :: Throws ArithError l => Expr -> EIO l Double
eval3 (Val x) = return x
eval3 (Add a1 a2) = ...
eval3 (Div a1 a2) = do
    v1 <- eval3 a1
    v2 <- eval3 a2
    if v2 == 0 then throwEIO DivByZero else return (v1 / v2)

main3 :: EIO l Double
main3 = eval3 (Div (Val 6.0) (Val 2.0) )
        'catchEIO' \DivByZero -> putStrLn "division by zero"

```

Fig. 4: eval in the EIO monad

```

parseExpr1 :: Throws ParseError l => String -> EIO l Expr

```

```

f :: (Throws ParseError l, Throws ArithError l) => EIO l Double
f = do {x <- getContents; p <- parseExpr1 x; eval3 p}

```

3.1 Generalizing our approach

One of Marlow's requirements is that the primitives for throwing and handling exceptions are always the same, regardless of the types involved. In this section we introduce overloaded versions of `throw` and `catch` which work with a family of monads, including the `EIO` monad and the regular `IO` monad. Thanks to these functions one can write code which works with any exceptions framework.

For each overloaded primitive we introduce a type class and two instances for `EIO` and `IO`. The `MonadThrow` class is defined without much effort below.

```

class (Monad m, Exception e) => MonadThrow e m where
    throw :: e -> m a
instance Exception e => MonadThrow e IO where
    throw = Control.Exception.throw
instance Throws e l => MonadThrow e (EIO l) where
    throw = throwEIO

```

The situation is quite more involved for the `catch` primitive. It is possible to define the `MonadCatch` class, but the Haskell 98 system is simply not expressive enough, even when extended when multi parameter type classes. Either functional dependencies (FDs) [5] or associated types (ATs) [1] are required in order to preserve type inference. Figure 5 shows an encoding of `MonadCatch` using FDs and the two instances for `IO` and `EIO`.

The details of the encoding of `MonadCatch` are not discussed here for the sake of simplicity, as it is not essential to the scheme, but notice how defining the instances themselves is rather simple. We also point out that the use of the overloaded `catch` method can be less convenient in practice. Consider a new version of `eval` using the overloaded version of `throw`. The type inferred will be

```

class (Exception e, Monad m, Monad m') =>
  MonadCatch e m m' | e m -> m', e m' -> m where
  catch :: m a -> (e -> m' a) -> m' a

instance Exception e => MonadCatch e IO IO where
  catch = Control.Exception.catch

instance Exception e => MonadCatch e (EIO (Caught e l)) (EIO l)
  where catch = catchEIO

```

Fig. 5: MonadCatch with Functional Dependencies

```
eval4 :: MonadThrow ArithError m => Expr -> m Double
```

This type is the most desirable one. It tells us that `eval4` may fail with an arithmetic error in any monad `m` which supports `throw`. However, the types inferred in presence of the overloaded version of `catch` are not so crisp.

```

main4 :: (MonadThrow ArithError m, MonadCatch ArithError m m') =>
  Expr -> m' Double
main4 = eval4 (Div (Val 6.0) (Val 2.0))
  'catch' \DivByZero -> putStrLn "division by zero"

```

The `MonadThrow ArithError` constraint is not discharged automatically. Instead the constraint is propagated, and a new `MonadCatch` constraint is introduced, leading to a rather long type context. On the other hand, it is still possible to instantiate `m'` to a concrete monad and recover the standard types. Instantiating to `IO` will always eliminate the constraints, by definition. Instantiating to `EIO` will make them go away in this case too, since indeed `main4` can throw no exception. In general, instantiating to `EIO` computes the list of remaining unhandled exceptions and makes them explicit as `Throws` constraints.

3.2 Dealing with the hierarchy of exceptions

The design proposed by Marlow allows for a limited form of exception subtyping based on a hierarchy of layers of existential type wrappers. For instance, we may wish to keep track of whether an overflow exception comes from a sum or from a division, by defining an exception `Overflow` and two subclasses `SumOverflow` and `DivisionOverflow`. This is done by creating a new existential wrapper `Overflow` which will be used as an intermediate layer before `SomeException`.

```

data OverflowException -- left abstract for our purposes

data SumOverflow      = SumOverflow      deriving (Show, Typeable)
data DivisionOverflow = DivisionOverflow deriving (Show, Typeable)

```

It is not essential to describe the encoding here in any further detail. The only relevant bit is that it is not possible to learn from the types any information about

the subtyping relation, as the relation is encoded at the value level and not at the type level. That is, when an exception `e` is handled, the `Throws e` constraint should be discharged *together with* any `Throws` constraints corresponding to a subclass of `e`. Unfortunately, there is no way to find out which are the subclasses of an exception.

Since this information is not readily available at the type level, the programmer will have to “introduce” it manually. This is far from ideal, but fortunately not too hard and could be automated using a macro system like CPP or Template Haskell. A `Throws` instance must be introduced for every ancestor-child relation. For the overflow example, this means that two `Throws` instances are needed.

```
instance Throws SumOverflow      (Caught OverflowException 1)
instance Throws DivisionOverflow (Caught OverflowException 1)
```

At this point it becomes more apparent that the `Throws` type class is encoding a relation, at the type level, between exception types and their handlers. To simplify the treatment of multi level hierarchies it would be desirable to declare that `Throws` is a transitive relation.

```
instance ( Throws parent (Caught grandparent 1)
         , Throws child  (Caught parent 1)
         ) => Throws child (Caught grandparent 1)
```

But this instance clearly conflicts with the second instance for `Caught` defined before in figure 3, as both have the same head modulo variable names. It should not be surprising that it is not possible to encode transitivity *directly* in this way, since after all the Haskell type checker is not a theorem prover. So even though with due effort there may be a more indirect way to encode transitivity indeed, we don’t discuss that problem here. It is likely that the price paid would be too high, in the form of unreadable error messages or poor robustness of the encoding.

4 General purpose explicitly typed exception monads

There is nothing specific to the IO Monad in our scheme. In fact we can combine it with the extensible exceptions of Marlow in order to obtain a general purpose monad for exceptions. We identified the main problem with the existing `MonadError` encoding as the inability to compose code throwing exceptions of different types. As Marlow has already solved the composability problem and our scheme recovers exception coverage, all we need to do is to put them together. We package the result as a monad transformer, which can then be used to add explicitly typed exceptions to any existing monad.

A monad transformer [8] is a type-level function that takes a monad as input and creates another monad. Monad transformers can be stacked, and every transformer adds zero or more effects to the stack. In this way a monad can be

```

newtype EMT l m a = EMT {unEMT :: m(Either SomeException a)}

runEMT :: Monad m => EMT l m a -> m a
runEMT (EMT m) = liftM fromRight m where fromRight (Right x) = x

instance Monad m => Monad (EMT l m) where
  return = EMT . return . Right
  EMT emt >>= f = EMT (do v <- emt
                        case v of
                          Left e -> return (Left e)
                          Right x -> unEMT (f x))

instance MonadTrans (EMT l) where
  lift = EMT . liftM Right

throwEMT :: (Monad m, Exception e, Throws e l) -> e -> EMT l m a
throwEMT = EMT . return . Left . toException

instance (Monad m, Exception e, Throws e l) => MonadThrow e (EMT l m) where
  throw = throwEMT

catchEMT (Monad m, Exception e) =>
  EMT l m a -> (e -> EMT (Caught e l) m a) -> EMT l m a
catchEMT (EMT m) h = EMT (do v <- unEMT m
                             case v of Right x -> return (Right x)
                                       Left e -> case fromException e of
                                                 Nothing -> EM (Left e)
                                                 Just e' -> h e')

instance (Monad m, Exception e) =>
  MonadCatch e (EMT (Caught e l) m) (EMT l m) where
  catch = catchEMT

```

Fig. 6: A monad transformer for checked, explicit exceptions

constructed piecemeal from a library of monad transformers. A monad transformer is represented in Haskell as a type constructor equipped with an instance of `MonadTrans`. The `lift` method lifts a computation in the underlying monad to the transformed monad.

```
class MonadTrans t where lift :: m a -> t m a
```

The monad transformer `EMT` of computations with checked, explicit exceptions is defined in figure 6. The code for the `Monad` instance follows from the `Either` monad studied at the beginning of the paper, only extended to deal with an underlying monad. The meaning of the `MonadThrow` and `MonadCatch` instances should be clear from the `EIO` monad defined in the previous section. Finally, the `runEMT` function safely removes the `Right` constructor and returns the computation inside; no test for a `Left` constructor is necessary since it is guaranteed that the result cannot be an exception.

```

newtype Identity a = Identity {runIdentity :: a}
instance Monad Identity where
  return = Identity
  Identity m >>= k = k m

```

Fig. 7: The Identity Monad

The EMT transformer can be instantiated with the standard `Identity` monad (figure 7) to obtain a exceptions monad. Using it it is possible can produce a version of our arithmetic evaluator which is guaranteed to never terminate with an unhandled exception and, as opposed to `eval3`, does not require running in the IO monad. We could define a new version `eval5` using the `throwEMT` primitive, but actually `eval4` can be reused directly, thanks to the `MonadThrow` instance of EMT. The compiler will infer the type `Double` for `main5` below.

```

main5 = let runEM = runIdentity . runEMT
         in runEM (eval4 (..) 'catch' \DivByZero -> ..)

```

The EMT monad and the `MonadThrow` and `MonadCatch` classes are available for experimentation in the `control-monad-exception` package in Hackage [4] released as a companion of this paper.

5 Unchecked exceptions

Checked exception mechanisms often include a facility to escape the rigidity of the mechanism in a controlled way. For instance, in Java every exception which is a subclass of `RuntimeException` is not checked. Our scheme is flexible enough to offer

- turning off the checking altogether.
- selective unchecked exceptions, by defining an exception to be unchecked.
- provide maximum static coverage checking even unchecked exceptions.

Checking can be turned off by introducing a function `tryEMT` (resp. `tryEIO`) that will accept any computation regardless of the `Throws` constraints associated to it, and will return either a result or an exception. `tryEMT` can be defined with the help of a *type flag* `AnyException` with a `Throws` instance that discharges any existing `Throws` constraint.

```

data AnyException
instance Throws e AnyException

tryEMT :: EMT AnyException m a -> m (Either SomeException a)
tryEMT (EMT m) = m

```

To turn an exception into an unchecked exception, all that is needed is to define an unconditional `Throws` instance, i.e. one which is trivially satisfied. For example, `ArithError` can be turned into an unchecked exception as follows.

```
instance Throws ArithError 1
```

Although this is very simple and convenient, and even though the fact that an exception `e` is unchecked is documented in the type system (by its `Throws` instance), it might be better to make this fact more explicit. We can do better by using an auxiliary type class `UncheckedException` to declare that an exception is unchecked.

```
class Exception e => UncheckedException e
```

Now, if we do nothing more, unchecked exceptions are still checked. If we wish to turn off the checking of unchecked exceptions, we instantiate the phantom type of our monad, be it `EIO` or `EMT`, with a type flag `WithUnchecked`.

```
data WithUnchecked
instance UncheckedException e => Throws e (WithUnchecked 1)
```

```
runEMTWithUnchecked :: EMT WithUnchecked m a -> m a
runEMTWithUnchecked = runEMT
```

Thanks to the accompanying `Throws` instance, exceptions which are instances of `UncheckedException` see their `Throws` constraints discharged when the flag is enabled. In effect, they are still explicitly typed, but `runEMTWithUnchecked`² will not complain if they escape without being handled.

6 Closing the `Throws` class

Defining the `EMT` (resp. `EIO`) type as an abstract datatype by not exporting its constructors and the type flags defined in the previous section is not enough to ensure that there is no way to bypass the type checker and disable checked exceptions. Even without access to the constructors, the user can work around the scheme by conjuring a `Forgetful` type flag similar to the `AnyException` flag used before to disable checked exceptions.

```
unsafeRunEMT :: EMT Forgetful m a -> m a
unsafeRunEMT = runEMT
```

In order to patch this hole, the ability of the user to define new type flags must be restricted. The only way to do this is to *close* the `Throws` type class using one of the existing techniques (see e.g. [10]). Closing the `Throws` type class is apparently at odds with the problem of handling exception hierarchies, where user defined `Throws` instances encoding hierarchical relations are needed, but it turns out that it is possible to selectively close the second type parameter of `Throws`, allowing instantiations to `Caught e`, while leaving the first one open.

² although `runEMTWithUnchecked` is naively defined as a synonym of `runEMT`, in practice one would add a default handler to capture any unchecked exceptions and avoid a pattern match failure error at runtime.

Following [10] we define a new type class `Allowed`, which is not exported, and add a `Allowed` constraint on the second parameter `l` in the declaration of the class `Throws l`.

```
class Allowed l
instance Allowed l => Allowed (Caught e l)

class (Exception e, Allowed l) => Throws e l
```

Trying to define `instance Throws Forgetful l` will now fail with a type error, since the `Allowed` constraint is not satisfied.

```
No instance for (Allowed Forgetful)
Possible fix: add an instance ... for (Allowed Forgetful)
```

Since the user has no way of manufacturing a new `Allowed` instance, the net effect is that the second parameter of the `Throws` class is *closed*.

7 Discussion

Exceptions are a feature of most programming languages nowadays. Haskell supports them either via the `IO` or the `MonadError` monad, but both encodings are still missing an important feature: static coverage checking of exceptions.

This paper shows a way to recover static coverage when extensible exceptions are used, providing self-documenting, explicitly typed exceptions. Most of the tricks used in the article are part of the functional programming folklore, but the reader will agree that they are put together with great effect: our scheme provides explicitly typed, checked and unchecked exceptions, is easy to understand, fits in a few lines of code, and best of all, comes *for free*: the implementation is purely static and imposes no extra runtime cost at all. Finally, all this is publicly available in `Hackage` in the package `control-monad-exception` [4].

Related work We have already mentioned the work of Marlow. The next closest work is by Teller et al. [12] on the excellent *Catch Me* library of type-safe, monadic exceptions for `Ocaml`. They analyze an error monad like the one used in the introduction of this article, and point out the shortcomings we identified: lack of compositionality and loss of coverage. Their library uses polymorphic variants [2] (extensible sum types) to improve compositionality by eliminating the boilerplate needed to combine different exception types. This goes a long way towards getting coverage exception too, although it does not completely solve the problem as one is still expected to handle every exception type in the sum. As a side note, they mention that the use of the dynamic facilities provided by `Typeable` would forbid any automatic coverage check. We just showed that actually the use of `Typeable` is no obstacle at all.

On a broader scope, the *Catch* tool [11] for Haskell uses static analysis to guarantee that a program cannot fail with a pattern match failure, even in the presence of non exhaustive pattern matches. Similarly, the *OcamlExc* [7] tool

uses static analysis to infer the exceptions an Ocaml computation can produce and to provide coverage. It is unclear how well these would interact with the dynamic mechanism used by the extensible exceptions of Marlow.

Acknowledgements The author would like to express his gratitude to Bernie Pope for his valuable feedback on a draft version of the article.

References

1. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM.
2. Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.
3. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 11.2. Sun Microsystems, 1996.
4. José Iborra. <http://hackage.haskell.org/cgi-bin/hackage-scripts/package/control-monad-exception>.
5. Mark P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag.
6. Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*, pages 9–9, Berkeley, CA, USA, 1999. USENIX Association.
7. Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000.
8. Shen Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, Jan 1995.
9. Simon Marlow. An extensible dynamically-typed hierarchy of exceptions. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 96–106, New York, NY, USA, 2006. ACM.
10. Conor McBride. <http://www.mail-archive.com/haskell-cafe@haskell.org/msg62512.html>.
11. Neil Mitchell and Colin Runciman. Not all patterns, but enough - an automatic verifier for partial but sufficient pattern matching. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 49–60. ACM, September 2008.
12. David Teller, Arnaud Spiwack, and Till Varoquaux. Catch me if you can: Towards type-safe, hierarchical, lightweight, polymorphic and efficient error management in ocaml. In *ML Workshop 2008*, 2008.
13. Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM.