An Extensible Dynamically-Typed Hierarchy of Exceptions

Simon Marlow
Microsoft Research
simonmar@microsoft.com

Abstract

In this paper we address the lack of extensibility of the exception type in Haskell. We propose a lightweight solution involving the use of existential types and the `Typeable` class only, and show how our solution allows a fully extensible hierarchy of exception types to be declared, in which a single overloaded `catch` operator can be used to catch either specific exception types, or exceptions belonging to any subclass in the hierarchy. We also show how to combine the existing object-oriented framework OOHaskell with our design, such that OOHaskell objects can be thrown and caught as exceptions, with full support for implicit OOHaskell subtyping in the `catch` operator.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.3 [*Language Constructs and Features*]: Data Types and Structures

General Terms Languages, Design

Keywords Haskell, Exceptions

1. Introduction

Exceptions have been evolving in the context of Haskell since their introduction in Haskell 1.3. We start with a brief history of exceptions in Haskell.

Haskell 1.3 introduced monadic IO, and with it, the means for exceptions to be thrown and caught within the IO monad, and this interface to exceptions carried through into Haskell 98. Exceptions have the type `IOError`, are thrown using `ioError`, and caught using `catch`. The `IOError` type is abstract; the standard only specifies a selection of predicates and projections over it, and there isn't even a way to construct an `IOError` (although nowadasys compilers do provide a standard way to do this). The abstract `IOError` type means that an implementation is free to extend the range of errors represented by `IOError`, although library code cannot.

Imprecise exceptions [10], and later also asynchronous exceptions [7] were introduced in GHC. The interface provided by GHC went through several iterations, finally ending up with what we have now; there is a fixed datatype `Exception`:

```
data Exception
  = ArithException ArithException
  | IOException IOException
  | PatternMatchFail String
  | UserError String
  | ...
```

`Exception` encodes all the possible exceptions that the system knows about. In particular, `Exception` subsumes `IOError` (`IOException` is a type synonym for `IOError`). There is a library `Control.Exception` that provides the means to throw and catch exceptions:

```
throw :: Exception -> a
catch :: IO a -> (Exception -> IO a) -> IO a
```

The obvious problem with this formulation is that the `Exception` type is not extensible at all: there is no way for library code or programs to extend the range of exceptions with their own types. Haskell 98's `IOError` type is designed to be extensible by an implementation: the standard does not specify the type concretely, but rather specifies a number of predicates and projections on it, but this is insufficient to allow arbitrary library code to extend the `IOError` type with its own exceptions.

As a result, today we often see library code that simply throws `UserError` exceptions for errors, or worse, just calls `error`. Users of these libraries have no way to reliably catch and handle these exceptions, and there is no documentation, aside from the source code of the library, to indicate which kinds of exception may be thrown.

There are two ways commonly used to work around this deficiency. Firstly, we can serialise the exception value that we want to throw as a `String`, and use the `UserError` exception to transport it. All we need to do is make sure the type we want to throw is an instance of `Show` and `Read`, and we can throw it using `throw (UserError (show x))`. To catch it, we could provide our own catching function:

```
catchMyType :: IO a -> (MyType -> IO a) -> IO a
catchMyType io handler =
  io `catch` \e ->
     case e of
       UserError s -> case reads s of
                        [(x,"")] -> handler x
                        _ -> throw e
       _ -> throw e
```


Haskell'06 September 17, 2006, Portland, Oregon, USA.

If the `String` successfully parses using the `Read` instance for the type we are looking for, then the supplied handler is invoked, otherwise the exception is re-thrown.

This approach suffers from several problems:

- defining `Show` and `Read` instances isn't always possible (for example when the type contains functions),
- the `Show` instance for this type must be unique, that is we won't mistake another type for our type,
- serialising/deserialising via `String` is unnecessarily slow.
- The extra code generated by deriving `Show` and `Read` instances is not insignificant, and might not otherwise be required.

Encoding arbitrary types as `String`s is just a poor man's version of dynamic typing. So the second approach to allowing arbitrary types to be thrown and caught is to use real dynamic types, as provided by the `Typeable` class [5]. The `Exception` type already contains a `DynException` constructor for this purpose:

```
data Exception  = ... | DynException Dynamic | ...

throwDyn :: (Typeable ex) => ex -> b

catchDyn :: (Typeable ex)
         => IO a
         -> (ex -> IO a)
         -> IO a
```

So as long as our type is an instance of `Typeable` (which can be derived for arbitrary types in GHC), we can throw and catch it using `throwDyn` and `catchDyn` respectively. This works, but the interface is a little clunky to say the least. The programmer has to decide whether to use `throwDyn` and `catchDyn` versus plain `throw` and `catch` based on whether the exception is a built-in one or not.

Moreover, we still cannot extend, say, the range of IO exceptions or the range of arithmetic exceptions: it should be possible to write an exception handler that catches all IO exceptions, even the as-yet-unknown ones.

Contrast the above solutions with what is typically provided by an object oriented language such as Java. There is an `Exception` class, which has a number of subclasses for categories of exceptions (`IOException`, `RuntimeException`, and so on). User code and libraries can extend the hierarchy at will, simply by declaring a new class to be a subclass of an existing class in the `Exception` hierarchy. Java has dynamic typing and subtyping built-in, in the sense that you can ask whether a given object is an instance of a particular class (a *downcast*), so catching an exception can check whether an exception being caught is a member of the new class.

To sum up the requirements, we would like our exception library to provide:

- A hierarchy of exception types, such that a particular catch can choose to catch only exceptions that belong to a particular subclass and re-throw all others.
- A way to add new exception types at any point in the hierarchy from library or program code.
- The boilerplate code required to add a new type to the exception hierarchy should be minimal.
- Exceptions should be thrown and caught using the same primitives, regardless of the types involved.

Efficiency is not a priority, since we expect exceptions to be used for erroneous conditions rather than as a general mechanism for control flow.

The main contribution of this paper is to describe a lightweight solution that meets the above requirements and more. The code for the core of the library is entirely contained in Sections 2 and 3. An intermediate Haskell programmer should be able to grasp the details of the implementation without too much difficulty, and a beginner could easily follow the patterns and extend the exception hierarchy themselves.

We will discuss related work in detail in Section 8, but it is worth briefly putting this work in context first. Exceptions are one place where Haskell's choice of algebraic data types and polymorphism, as opposed to classes and subtyping, does not yield a natural way to express the interface we desire. For exceptions, we need the data to be extensible, whereas in Haskell typically the data is fixed, and the range of functions is extensible. In constrast, object-oriented languages emphasize extensible data with a fixed range of operations (this insight comes from the O'Haskell rationale page [9], although it has doubtless been expressed elsewhere). In both settings there are techniques for working around the respective limitations. This paper can be seen as exploring a solution to the problem of expressing an object-oriented-style API in the context of Haskell, albeit a very special-purpose API, namely exceptions.

For a good survey of the known techniques for encoding subtyping hierarchies in Haskell see the OOHaskell[1] paper [4]. The requirements of exceptions are slightly unusual however, in that the catch operator needs to perform a dynamic downcast; oddly enough, although the OOHaskell paper does describe various techniques for downcasting, none of them applies in this setting. Furthermore, in this paper we are aiming for a lightweight solution to the problem, and OOHaskell comes with an elaborate type-level-programming framework that seems overkill for exceptions. Nevertheless, it is interesting to investigate whether OOHaskell objects can be thrown and caught as exceptions, while retaining the subtyping properties that OOHaskell provides. We appreciate that some users will want to do just that; the full story is given in Section 7.

To give a feel for the kind of facilities that our proposal provides, there follows a few examples of our library in use. The examples are taken directly from a GHCi session, except that the normally-long prompt has been replaced by `>`, and some extra newlines have been added to fit the code into the column.

Firstly, we can throw any exception using the `throw` primitive, and catch it again using `catch`:

```
> :t DivideByZero
DivideByZero :: DivideByZero
> throw DivideByZero
      `catch` \(e::DivideByZero) -> print "caught"
"caught"
```

The type of the handler determines which exceptions are caught; if an exception is not of the desired type, it is not caught and is passed up to the next enclosing `catch`. For example, a `DivideByZero` will not be caught by a handler looking for the end-of-file exception, but it will be caught by a handler looking for any exception[2]:

[1] not to be confused with O'Haskell

[2] infix `` `catch` `` is defined to be left-associative

```
> throw DivideByZero
      `catch` (\(e::EOF) -> print "caught EOF")
      `catch` (\(e::SomeException) -> print "other")
"other"
```

Exceptions are structured in a hierarchy, so it is possible to match classes of exceptions. For example, `DivideByZero` is an arithmetic exception:

```
> throw DivideByZero
       `catch` \(e::SomeArithException) ->
                  print "caught"
"caught"
```

The exception hierarchy is fully extensible: new exception types can be added to an existing node in the hierarchy easily (less than 5 lines of code per type), and new nodes can be added to the hierarchy (about 10 lines per node). We show how to do this later in the paper.

Finally, we can catch several kinds of exception with a single handler:

```
  e `catches` [
        Catch $ \(x::DivideByZero)    -> print x,
        Catch $ \(x::SomeIOException) -> print x
    ]
```

The code presented in this paper requires two extensions to Haskell 98: existential types, and the `Data.Typeable` library. Both are well-understood and implemented by the major compilers, and both are likely to be in the next revision of the Haskell language.

For convenience only, we use several more Haskell extensions in this paper. These aren't fundamental to the design of the library, although they make using it easier. They are: scoped type variables (for putting type signatures on patterns), deriving the `Typeable` class, generalised deriving for newtypes, and pattern guards. All of these are also likely to be in the next revision of Haskell.

2. An extensible class of exceptions

Haskell already has a fine mechanism for defining open-ended extensible sets of types, namely type classes. Let us start, then, by making an extensible *set* of exceptions, and then proceed to extend it to a hierarchy.

First, we define a class of exception types, `Exception`:

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

The `Exception` class has no methods; it is really just a synonym for `Typeable` and `Show`. A type that we want to throw as an exception must be an instance of `Typeable`, and we also require that all exceptions provide `Show`, so that the system can always print out the values of uncaught exceptions.

Our simple interface for throwing and catching is as follows:

```
throw :: (Exception e) => e -> a

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

Any type that is an instance of `Exception` can be thrown. A particular `catch` will catch only a certain type of exceptions, which must be an instance of `Exception`. These `throw` and `catch` functions are equivalent to the `throwDyn` and `catchDyn` described earlier.

A new type can be used as an exception in a straightforward way:

```
data AssertionFailed = AssertionFailed String
  deriving (Typeable, Show)

instance Exception AssertionFailed
```

throwing and catching the new exception is simple:

```
>  throw (AssertionFailed "foo")
    `catch` \(e::AssertionFailed) -> print e
AssertionFailed "foo"
```

The underlying implementation must in fact always throw a value of a single, fixed, type. This is because `catch` cannot know the type of the exception that was thrown, and yet it must be able to interpret the exception value that it catches. In Haskell we don't have implicit runtime reflection; it is not possible to ask the type of an arbitrary value. So we define the type of objects that are thrown as follows:

```
data SomeException
  = forall a . (Exception a) => SomeException a
  deriving Typeable
```

`SomeException` is defined to be a value of an existentially-quantified type `a`, which ranges over instances of the class `Exception`. That is, `SomeException` is essentially just a dynamically typed value; it is similar to the type `Dynamic`, but an existential is more useful here, as we will see shortly.

Throwing and catching are defined as follows:

```
throw e = primThrow (SomeException e)

catch io handler
  = io `primCatch` \(SomeException e) ->
          case cast e of
              Just e' -> handler e'
              Nothing -> throw e
```

Where the function `cast` is part of the `Typeable` library:

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

The functions `primThrow` and `primCatch` are the low-level throwing and catching primitives provided by the implementation. For the purposes of experimentation, we can implement these using the existing `Control.Exception` library:

```
primThrow = Control.Exception.throwDyn
primCatch = Control.Exception.catchDyn
```

We can make `SomeException` an instance of `Exception` in the normal way; this is quite useful as it means that the existing `catch` can be used to catch *any* exception. In order to do this, we must first make `SomeException` an instance of `Show`:

```
instance Show SomeException where
  show (SomeException e) = show e

instance Exception SomeException
```

The `Show` instance for `SomeException` prints out its contents. This works because `Show` is a superclass of `Exception`, and so the `Show` instance for the value inside `SomeException` is available through the existential `Exception` predicate.

Unfortunately the definition of `catch` above cannot accommodate handlers that catch `SomeException`, it must be elaborated slightly[3]:

```
catch io handler
  = io `primCatch` \e@(SomeException e') ->
          case cast e of
              Just e'' -> handler e''
              Nothing  -> case cast e' of
                            Just e'' -> handler e''
                            Nothing  -> throw e'
```

Now that `SomeException` is an instance of `Exception`, we can catch an arbitrary exception and print it:

```
>  throw (AssertionFailed "foo")
    `catch` \(e::SomeException) -> print e
AssertionFailed "foo"
```

We can also define a finally combinator[4], which performs its first argument followed by its second argument. The second action is always performed, even if the first action throws an exception:

```
finally :: IO a -> IO b -> IO a
finally io at_last
 = do a <- io `catch` \(e::SomeException) ->
                        do at_last; throw e
      at_last
      return a
```

3. Extending the set to a hierarchy

The design in the previous section allows exceptions to be added to a class `Exception`, with a single type `SomeException` representing an arbitrary exception value.

This gives us a clue as to how we might extend the technique to a hierarchy. The previous design can be viewed as a two-level hierarchy, in which `SomeException` is the root, and each of the instances of `Exception` are subclasses of that. To extend the scheme to a hierarchy of arbitrary depth, each non-leaf node of the hierarchy must be a dynamic type like `SomeException`, because the dynamic downcast that `catch` embodies must compare a path through the hierarchy (from root to node) from the catch site, with a path (root to leaf) in the exception value.

First we add two methods to the `Exception` class:

```
class (Typeable a, Show a) => Exception a where
  toException   :: a -> SomeException
  fromException :: SomeException -> Maybe a

  toException = SomeException
  fromException (SomeException e) = cast e
```

The `toException` method maps an instance of `Exception` to the root of the hierarchy, `SomeException`. The `fromException` method dynamically compares the type of an exception against a supplied type, for use in `catch`.

Our `throw` and `catch` primitives are now defined like this:

```
throw e = primThrow (toException e)
catch io handler
  = io `primCatch` \e ->
       case fromException e of
           Nothing -> throw e
           Just e' -> handler e'
```

The default methods of `toException` and `fromException` work for direct children of `SomeException`, so that we can continue to define new exceptions at the top of the hierarchy as before.

Defining a new node in the hierarchy is quite easy. Let's define a class of arithmetic exceptions, `ArithException`:

```
data SomeArithException
  = forall a . (Exception a) => SomeArithException a
  deriving Typeable

instance Show SomeArithException where
  show (SomeArithException e) = show e

instance Exception SomeArithException
```

This type is isomorphic to `SomeException`. In fact we could use a `newtype`, but as we will see later we may want to define nodes that have more existential constraints besides `Exception`.

We don't need to define the methods of the `Exception` instance, because the default methods work fine: `SomeArithException` is a direct child of `SomeException`.

We now define two helper functions that will be used when subclassing `ArithException`:

```
arithToException :: (Exception a)
                 => a -> SomeException
arithToException = toException
                 . SomeArithException

arithFromException :: (Exception a)
                   => SomeException
                   -> Maybe a
arithFromException x = do
    SomeArithException a <- fromException x
    cast a
```

We'll explain how these functions work later.

In total, that's about 10 lines of boilerplate code to create a new node in the hierarchy (another 2 lines is required if the node isn't a child of the root, because the methods of `Exception` are required).

[3] Thanks to a reviewer of an earlier version of this paper for pointing out this problem.

[4] For simplicity, this version of finally doesn't take account of asynchronous exceptions [7].

Now, let's create an instance of an arithmetic exception, the divide-by-zero exception. This will be a child of `SomeArithException` in the hierarchy:

```
data DivideByZero = DivideByZero
  deriving (Typeable, Show)

instance Exception DivideByZero where
  toException   = arithToException
  fromException = arithFromException
```

It took an extra 2 lines to declare a type to be a child of a non-root node, compared to a child of the root, `SomeException`.

Now, we can write code that catches any arithmetic exception. For example:

```
> throw DivideByZero
         `catch` \(e::SomeArithException) -> print e
DivideByZero
```

or we can catch just `DivideByZero` exceptions:

```
> throw DivideByZero
         `catch` \(e::DivideByZero) -> print e
DivideByZero
```

The intuition for how this works goes as follows. Each type that you can throw, like `DivideByZero`, is an instance of `Exception`, and notionally resides at the leaf of a virtual hierarchy. The hierarchy isn't manifest anywhere, because it is dynamically extensible, but it is embodied in the implementations of the `toException`/`fromException` methods of the `Exception` instances.

When we `throw` an exception value, it is wrapped in constructors, one for each parent node successively until the root is reached. For example, when we throw `DivideByZero`, the value actually thrown is

```
SomeException (SomeArithException DivideByZero)
```

Catching an exception and comparing it against the desired type does the reverse: `fromException` unwraps the value, and at each level of the tree compares the type of the next child against the desired type at that level. We can make dynamic type comparisons at each level because of the existential `Typeable` constraint embedded in each node. See `arithFromException` earlier for example: it starts by attempting to extract a `SomeArithException` from the `SomeException` it is passed, and then proceeds by attempting to cast the contents of the `SomeArithException` to the desired type.

Creating a further subclass should help to illustrate how the mechanism extends:

```
data SomeFloatException
  = forall x . (Exception x) => SomeFloatException x
  deriving Typeable

instance Exception SomeFloatException where
  toException = arithToException
  fromException = arithFromException

instance Show SomeFloatException where
  show (SomeFloatException x) = show x

floatToException :: (Exception x)
                 => x -> SomeException
floatToException = toException
                 . SomeFloatException

floatFromException :: (Exception x)
                   => SomeException
                   -> Maybe x
floatFromException x = do
   SomeFloatException a <- fromException x
   cast a
```

4. Attaching methods and data to subclasses

We have a hierarchy of exception types, which is open-ended extensible, and we can do dynamic type comparisons of types that inhabit the hierarchy. Our requirements from Section 1 are satisfied; but is this enough? Compared to the object oriented formulation, we are still somewhat impoverished: in an object-oriented language, each node of the object oriented hierarchy can also contain methods and instance variables that are inherited by subclasses[5].

Consider I/O exceptions in Haskell 98. The existing interface for I/O exceptions lets you query an exception value in various ways:

```
ioeGetErrorString :: IOException -> String
ioeGetHandle      :: IOException -> Maybe Handle
ioeGetFileName    :: IOException -> Maybe FilePath
```

additionally we can ask a value of type `IOException` what kind of error it represents:

```
isEOFError         :: IOException -> Bool
isIllegalOperation :: IOException -> Bool
isPermissionError  :: IOException -> Bool
```

So every `IOException` contains information about the context in which the error occurred (the `Handle` and `FilePath` involved in the operation, if any), and the kind of error.

In our new framework, we could make `IOException` an instance of `Exception` and be done with it, but that doesn't seem right: we couldn't add new kinds of I/O exceptions from library code in the future. Really, we want I/O exceptions to be an extensible subclass, like arithmetic exceptions. Furthermore, we also want to be able to use generic methods like `ioeGetHandle` on anything that is an I/O Exception.

I/O exceptions are essentially an object-oriented class, and we simply require a way to model this in Haskell. The solution we adopt, namely to replace the `IOException` type by a type class, is one of the alternatives proposed by Shields and Peyton Jones in the context of reflecting the .NET object hierarchy in the Haskell type system [12]. Our `IOException` class is as follows:

```
class IOException a where
  ioeGetErrorString :: a -> String
  ioeGetHandle      :: a -> Maybe Handle
  ioeGetFileName    :: a -> Maybe FilePath
```

[5] and can be overriden by subclasses, but we will not worry about that in this paper.

Next, we make a node in the exception hierarchy for IO exceptions:

```
data SomeIOException
  = forall a . (Exception a, IOException a) =>
      SomeIOException a
  deriving Typeable

instance Show SomeIOException where
  show (SomeIOException x) = show x

instance Exception SomeIOException

ioToException :: (IOException x, Exception x)
              => x
              -> SomeException
ioToException = toException . SomeIOException

ioFromException :: (IOException x, Exception x)
                => SomeException
                -> Maybe x
ioFromException x = do
   SomeIOException a <- fromException x
   cast a
```

Note that the `SomeIOException` constructor has a new existential constraint: `IOException a`, which ensures that all children of `SomeIOException` in the hierarchy are instances of `IOException`. This means we can catch any IO exception and apply methods of the `IOException` class, for example:

```
  x `catch` \(SomeIOException e) ->
               print (ioeGetErrorString e)
```

Note that we're pattern matching directly on the `SomeIOException` constructor, rather than just constraining the type of the exception as in previous examples. This is necessary because we need to extract the child of the `SomeIOException` constructor; also note that this requires `SomeIOException` to be non-abstract.

Now we can define some actual I/O exceptions. For example, the end-of-file exception:

```
data EOF = EOF ...
  deriving Typeable

instance Exception EOF where
  toException = ioToException
  fromException = ioFromException

instance IOException EOF where
  ...

instance Show EOF where
  ...
```

The ellipses (`...`) represent sections of code that are private to the implementation of the `EOF` datatype: we don't mind how it is implemented, as long as it provides the methods of the `IOException` class.

We haven't given a way to construct one of these exceptions yet. Of course in general, constructing an instance of `IOException` depends on the exception itself, since it may contain data specific to that particular exception. However, many IO exceptions contain just the data necessary to implement the methods of `IOException`, and so can be built using a common interface. Suppose we provide:

```
class (IOException a) => BasicIOException a where
  newIOException :: Maybe Handle
                 -> Maybe FilePath -> String -> a
```

then we can provide an instance of `BasicIOException` for each of the exising I/O exception types: `EOF`, `NoSuchThing`, `AlreadyExists`, and so on.

4.1 Reducing code duplication

This is still rather cumbersome, however. For each new I/O exception, we need to define a new datatype that contains the same three fields, together with instances of `Exception`, `IOException`, `SimpleIOException`, and `Show`. We can cut down on the amount of duplicated code as follows:

```
data IOExceptionInfo = IOExceptionInfo
  { ioeHandle       :: Maybe Handle,
    ioeFilePath     :: Maybe FilePath,
    ioeErrorString  :: String }
  deriving Typeable

instance IOException IOExceptionInfo where
  ioeGetHandle      = ioeHandle
  ioeGetFilePath    = ioeFilePath
  ioeGetErrorString = ioeErrorString

instance BasicIOException IOExceptionInfo where
  newIOException = IOExceptionInfo
```

Then, each new exception type can be defined as a `newtype` of `IOExceptionInfo`:

```
newtype EOF = EOF IOExceptionInfo
  deriving (Typeable,IOException,BasicIOException)
```

The instances of `Exception` and `Show` are still required, but we can use GHC's generalised `newtype` deriving to automatically provide instances of `IOException` and `BasicIOException`. In fact, there will be no code generated for these instances at all, GHC just reuses the dictionary for the instance of `IOExceptionInfo`.

5. A failed alternative

The reader might wonder why, instead of defining our hierarchy with layers of existentially typed wrappers as we did above, we didn't just use parameterised datatypes – after all, a parameterised datatype doesn't restrict which parameters it may be instantiated with, and so it must be extensible. So, imagine that we have the simple `Exception` class defined in Section 2, and we wish to define a subclass of arithmetic exceptions like this:

```
newtype ArithException e = ArithException e
  deriving Typeable

instance (Show x) => Show (ArithException x) where
  show (ArithException x) = show x

instance (Typeable x, Show x) =>
           Exception (ArithException x)
```

One can think of `ArithException` here as a degenerate case of an extensible record implemented using tail-polymorphism [1, 4],

and it is similar to the use of phantom types for encoding subtype hierarchies [3, 2]. If we were to elaborate this example, using tail-polymorphism would ensure that our hierarchy retained the desired extensibility.

So far so good. Now we define an instance of an arithmetic exception:

```
data DivideByZero = DivideByZero
  deriving (Typeable, Show)
```

and indeed we can throw and catch an instance of `ArithException`:

```
> throw (ArithException DivideByZero)
     `catch` \(e::ArithException DivideByZero) ->
                print "caught"
"caught"
```

It is mildly annoying that we have to write out the exception in full when throwing it, the system doesn't know that `DivideByZero` *is* an arithmetic exception. We could work around this partially by providing a `divideByZero` constant to throw instead.

However, the real problem with this approach is evident when we try to write an exception handler that catches any arithmetic exception. Given this code:

```
test = throw (ArithException DivideByZero)
         `catch` \(e::ArithException x) ->
                    print "hello"
```

GHC complains thus:

```
Failed.hs:48:43:
    Ambiguous type variable `x' in the constraints:
      `Typeable x' arising from use of `catch'
         at Failed.hs:48:43-49
      `Show x' arising from use of `catch'
         at Failed.hs:48:43-49
    Probable fix: add a type signature that fixes
        these type variable(s)
```

the problem is that the argument to our handler function is polymorphic in the type variable `x`, and the type of `catch` requires that the argument to the handler is an instance of `Typeable`.

Intuitively, we require `catch` to not match the whole type of the exception against the handler, but recognise that this is a polymorphic handler, and only match the necessary parts of the type. There isn't a way (that I know of) to make a `catch` that behaves like this, but we can define a variant `catch1` that does the right thing:

```
catch1 :: (Typeable1 t)
       => IO a
       -> (forall x. t x -> IO a)
       -> IO a

catch1 io h =
  io `primCatch`
         \(SomeException e) ->
             case gapply1 h e of
                  Nothing -> Ex.throwDyn e
                  Just io -> io
```

The `Typeable1` class is a variant of `Typeable` for unary type constructors. It is provided by `Data.Typeable`:

```
class Typeable1 t where
  typeOf1 :: t a -> TypeRep
```

The definition of `catch1` mentions a function `gapply1`, that looks like it should be provided by `Data.Typeable`, but isn't. Here is its type:

```
gapply1 :: (Typeable1 t, Typeable a)
        => (forall x. t x -> b)
        -> a
        -> Maybe b
```

`gapply1` attempts to apply the polymorphic function in its first argument to the dynamic type in its second argument, succeeding only if the type constructor of the dynamic type matches the type constructor expected by the polymorphic function. For reference, here is an implementation:

```
gapply1 ftx a
  | fst (splitTyConApp (typeOf a))
      == fst (splitTyConApp (typeOf1 (getarg ftx)))
  = Just (ftx (unsafeCoerce# a))
  | otherwise
  = Nothing
  where getarg :: (t x -> b) -> t x
        getarg = undefined
```

This is all very interesting, but academic: this solution is clearly inferior to the one proposed in Section 3, because instead of a single `catch`, we need a family of them: `catch`, `catch1`, `catch2`, and so on. Moreover, the values are more cumbersome (`ArithException DivideByZero` instead of just `DivideByZero`), and we need more extensions (higher-rank polymorphism in the type of `gapply1`).

It is possible that a more elaborate system of dynamic typing, such as that of Clean [11], would eliminate the need for a separate `catch1` here. We have not explored this possibility.

6. Catching multiple exception classes

The programmer might want to catch multiple classes of exception with a single handler. For example, suppose we wish to catch both overflow and divide-by-zero exceptions arising from a particular computation, and return the value zero:

```
expr `catch` \DivideByZero -> return 0
     `catch` \Underflow    -> return 0
```

(we treat infix `catch` as left-associative). Using nested `catch` as in this example works, but it is not ideal: at run-time there will be two nested exception handlers, and if the inner handler does not match the exception, then it will be re-thrown, caught by the outer handler, and matched again.

It is possible to define a version of `catch` that takes multiple alternatives, by wrapping each alternative in an existential:

```
data Catch a = forall e . (Exception e)
                        => Catch (e -> IO a)
```

Then we can write `catches`, a multi-alternative variant of `catch`, as follows:

```
catches :: IO a -> [Catch a] -> IO a
catches io alts = io `catch` catchAlts alts

catchAlts :: [Catch a] -> SomeException -> IO a
catchAlts alts e = foldr check (throw e) alts
  where
        check (Catch f) rest =
            case fromException e of
                  Just h  -> f h
                  Nothing -> rest
```

`catches` can be used as follows:

```
expr `catches` [
   Catch $ \DivideByZero -> return 0
   Catch $ \Underflow    -> return 0
 ]
```

Note that the alternatives are tried in sequence, so more specific handlers must come before less specific. This allows for the common case of catching a specific exception, with a fallback handler for other exceptions in the class.

7. OOHaskell records as exception types

OOHaskell[4] is a type-level-programming framework that provides a full object-oriented type system in Haskell, complete with structural record subtyping. OOHaskell requires more Haskell extensions: it uses multi-parameter type classes with functional dependencies, and also overlapping/undecidable instances.

Since OOHaskell already provides subtyping, it is natural to ask whether OOHaskell records can be used as exceptions in our framework. The answer is yes; although OOHaskell as it stands does not provide the fully dynamic downcast that we require to implement `catch` for records. OOHaskell provides two ways to downcast:

- An upcast that retains the original type as a `Dynamic`, where the upcast value may be downcast to the original type again, but *only* the original type. This means that in order to downcast to a supertype of the original type, one must know or guess the original type, and that isn't possible in the context of `catch`.
- A fully typed upcast, with downcasting to any supertype of the original type. This also isn't appropriate for `catch`, because it relies on having full type information for the upcasted value, and `catch` only has a dynamic type to work with.

Nevertheless, it is possible to define a fully dynamic downcast for OOHaskell records; we built a prototype, and following personal communication the OOHaskell authors were kind enough to explain how to construct an elegant solution, an implementation of which is given in Figure 1 at the end of this paper. Briefly, the following are required:

```
class FieldsTypeable a

recToDyn  :: (FieldsTypeable a)
          => Record a
          -> DynRecord

narrowDyn :: (FieldsTypeable a)
          => DynRecord
          -> Maybe (Record a)
```

where `FieldsTypeable` is a new class, with instances provided for all record types with `Typeable` fields. The function `recToDyn` upcasts a record to a dynamic record, and `narrowDyn` downcasts a dynamic record to an arbitrary supertype of the original record type (one could also think of `narrowDyn` as an upcast, if `DynRecord` is just a dynamic representation of the original record).

Given these definitions, we can incorporate OOHaskell records into our exception framework quite straightforwardly. We start by defining a node in the exception hierarchy for records:

```
data SomeRecord =
  forall r. (ShowComponents r, FieldsTypeable r)
         => SomeRecord r
  deriving Typeable

instance Show SomeRecord where
  show (SomeRecord r) = show (Record r)

instance Exception SomeRecord
```

The `ShowComponents` constraint is part of the OOHaskell framework, it is required for converting records to `String`s.

Now, the magic part is that we can make every record an instance of `Exception`:

```
instance ( FieldsTypeable a,
           Typeable a,
           ShowComponents a )
         => Exception (Record a)
  where
  toException (Record a) =
    toException (SomeRecord a)

  fromException (SomeException a) = do
    SomeRecord r <- cast a
    narrowDyn (recToDyn (Record r))
```

The `toException` method is boilerplate: every record is wrapped in `SomeRecord` when thrown. In `fromException`, we convert the record in the exception to a `DynRecord` using `recToDyn`, and then attempt to use `narrowDyn` to cast it to the required type. `narrowDyn` will return `Nothing` if the desired type is not a supertype of the record in the exception, in which case the result of `fromException` will be `Nothing`.

The following examples illustrate that we can throw and catch arbitrary records, with subtyping and full type inference. First, we define some example record types. `L1`, `L2`, and `L3` are labels, defined with some boilerplate required by OOHaskell:

```
data L1 deriving Typeable
l1 :: Proxy L1
l1 = proxy

data L2 deriving Typeable
l2 :: Proxy L2
l2 = proxy
```

```
data L3 deriving Typeable
l3 :: Proxy L3
l3 = proxy
```

`rec` is an example record with three fields:

```
rec = (   l1 .=. True
      .*. l2 .=. "fish"
      .*. l3 .=. 642
      .*. emptyRecord
     )
```

Now, we can throw `rec` and catch it as an arbitrary exception:

```
*Main> throw rec `catch`
          \(e::SomeException) -> print e
Record{l1=True,l2="fish",l3=642}
```

The following are some types that we expect to be supertypes of the type of `rec`, by selecting a subset of the fields:

```
type JustL1   = Record ( L1 :=: Bool :*: HNil )

type JustL2L1 = Record ( L2 :=: String
                       :*: L1 :=: Bool
                       :*: HNil )
```

To demonstrate that we can throw `rec` and catch a supertype:

```
*Main> throw rec `catch` \(e::JustL1) -> print e
Record{l1=True}
*Main> throw rec `catch` \(e::JustL2L1) -> print e
Record{l2="fish",l1=True}
```

An interesting aspect of this formulation is that it combines two forms of subtyping; the limited nominal subtyping provided by our framework of existential types, together with the general record subtyping provided by OOHaskell. Yet, the programmer's interface is simple and intuitive.

8. Related Work

Haskell has an unusually expressive type system, and in many ways the community is only beginning to understand its power; many programming idioms that were previously thought to require new extensions to Haskell have recently been discovered to be already possible in Haskell 98, or with common existing extensions.

So as one might expect, there is more than one way to achieve the goals set out in Section 1 in Haskell. The contribution of this paper is to describe a solution that is relatively lightweight in that it doesn't rely on external scaffolding, and can be completely described in this short paper. This means that the technique will be accessible and understandable to many, which is a useful property for something as central to the language as exceptions.

In this section we outline some of the other methods that could lead to solutions to the problem, and where possible compare them to ours.

Open types. Open data types and open functions [6] are proposed extensions to Haskell to solve the "expression problem", in which most programming languages provide either a way to extend the range of operations on a type, or the range of constructors of the type, but not both. The authors even cite the extensible exception type problem as one target for their work, and describe how it is addressed by their solution. Compared to our approach, theirs requires new extensions to the language (although not deep), and has difficulties with separate compilation.

Arguably the open data types approach is more direct and more accessible, as is often the case with extensions designed to solve a particular problem. Still, the argument for adding open data types to the language is weakened by the fact that they are subsumed by type classes: in fact the authors give an encoding of open data types into type classes, but they argue that using type classes directly is less convenient than open data types, due to the lack of pattern matching and the inconvenience of the extra syntactic clutter. The approach described in this paper would benefit from direct pattern matching when writing a handler for multiple types of exception, but in the (common) case of catching a single class of exceptions we don't miss it.

Phantom types. Phantom types are useful for expressing subtyping hierarchies [3, 2], so it seems reasonable to wonder whether they might offer a solution to the extensible exception types problem. However, it turns out that phantom types are not applicable in this context, because using parameterised types leaves us with the problems described in Section 5, where we cannot easily write catch expressions that catch a class of exceptions.

HList and OOHaskell. We explored connecting OOHaskell[4] with our exception framework in Section 7. Does OOHaskell subsume our work here? Strictly speaking no: OOHaskell as it stands doesn't provide the required dynamic downcast operation, although we demonstrated how to add it earlier. Given this, in a sense OOHaskell does subsume the exception framework presented herein: if we were prepared to use OOHaskell records for exceptions exclusively, then we could easily define `throw` and `catch` using the OOHaskell library, and the user benefits from OOHaskell's subtyping instead of our ad-hoc framework. Furthermore, subtyping in OOHaskell is implicit, there is no need to declare subtypes as we do in this paper.

The main difference between OOHaskell and this work is that we are aiming for a lightweight solution. Bringing in a full type-level-programming framework seems overkill to solve the extensible exceptions problem. Furthermore, our solution works with arbitrary algebraic datatypes: any type can be spliced into the exception hierarchy by the addition of an instance of the `Exception` class. Additionally we have shown that should the programmer wish to use OOHaskell for exceptions, doing so in the context of our framework is eminently possible and the resulting interface is seamless.

O'Haskell. O'Haskell[8] extends Haskell with object-oriented subtyping. As such, it would be entirely possible to implement extensible exceptions using inheritance in O'Haskell. However, O'Haskell is a significant increment over Haskell, and our goal here was to achieve the simple task of an extensible exception type within Haskell using as few extensions as possible.

Exceptions in ML. In the ML family, including O'Caml, an extensible exception type is provided as a built-in language feature. The exception type is flat; there is no support for classes of exceptions.

9. Discussion and Conclusion

The question of whether Haskell should include support for extensible types comes up from time to time, and for a long time we assumed that in order to provide an extensible exception library we

would need to extend Haskell with some kind of extensible types. As we have shown in this paper, new extensions are not necessary to achieve a lightweight and attractive solution to the problem.

The questions before the community is: is this design suitable for adoption by the standard? We argue that, provided the extensions that we rely on (existentials and `Typeable`) are in the standard, then this framework is a suitable basis for exceptions.

References

[1] F. W. Burton. Type extension through polymorphism. *ACM Trans. Program. Lang. Syst.*, 12(1):135–138, 1990.

[2] S. Finne, D. Leijen, E. Meijer, and S. Peyton Jones. Calling Hell from Heaven and Heaven from Hell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 114–125, Paris, Sept. 1999. ACM.

[3] M. Fluet and R. Pucella. Phantom types and subtyping. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 448–460, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[4] O. Kiselyov and R. Lämmel. Haskell's overlooked object system, 2005. `http://homepages.cwi.nl/~ralf/OOHaskell/`.

[5] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, pages 26–37, New Orleans, Jan. 2003. ACM.

[6] A. Löh and R. Hinze. Open data types and open functions. In *Eighth ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, Venice, Italy, July 2006. ACM.

[7] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *ACM Conference on Programming Languages Design and Implementation (PLDI'01)*, pages 274–285, Snowbird, Utah, June 2001. ACM.

[8] J. Nordlander. O'Haskell. `http://www.cs.chalmers.se/~nordland/ohaskell/`.

[9] J. Nordlander. O'Haskell rationale. `http://www.cs.chalmers.se/~nordland/ohaskell/rationale.html`.

[10] S. Peyton Jones, A. Reid, C. Hoare, S. Marlow, and F. Henderson. A semantics for imprecise exceptions. In *ACM Conference on Programming Languages Design and Implementation (PLDI'99)*, pages 25–36, Atlanta, May 1999. ACM.

[11] M. Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[12] M. Shields and S. L. P. Jones. Object-oriented style overloading for haskell. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

```
class FieldsTypeable a where
  getFieldDynamics :: a -> [Dynamic]
  reconstruct      :: a{-dummy-} -> [Dynamic] -> Maybe a

instance FieldsTypeable HNil where
  getFieldDynamics HNil = []
  reconstruct      _ _  = Just HNil

instance (Typeable f, FieldsTypeable r) => FieldsTypeable (HCons f r) where
  getFieldDynamics (HCons f r) = toDyn f : getFieldDynamics r

  reconstruct undef all_fields = go all_fields
    where HCons _ r = undef

          go [] = Nothing
          go (f : fields)
            | Just f' <- fromDynamic f = do
                rest <- reconstruct r all_fields
                Just (HCons f' rest)
            | otherwise =
                go fields

newtype DynRecord = DynRecord [Dynamic]

recToDyn :: (FieldsTypeable a) => Record a -> DynRecord
recToDyn (Record r) = DynRecord (getFieldDynamics r)

narrowDyn :: (FieldsTypeable a) => DynRecord -> Maybe (Record a)
narrowDyn (DynRecord fields) = result
  where
    result = Record `liftM` reconstruct dummy fields
    dummy = undefined `asTypeOf` case fromJust result of Record a -> a
```

Figure 1. Implementation of dynamic downcast in OOHaskell