Exceptionally Monadic Error Handling

Looking at *bind* and squinting really hard

Jan Malakhovski*

IRIT, University of Toulouse-3 and ITMO University

February 2014 - October 2018

Abstract

We notice that the type of `catch :: c a -> (e -> c a) -> c a` operator is a special case of monadic `bind` operator `(>>=) :: m a -> (a -> m b) -> m b` and the semantic (surprisingly) matches.

For instance, the reader is probably aware that the monadic essence of the `(>>=)` operator of the error monad $\lambda A.E \vee A$ is to behave like identity monad for "normal" values and to stop on "errors". The unappreciated fact is that handling of said "errors" with a `catch` operator of the "flipped" "conjoined" error monad $\lambda E.E \vee A$ is, too, a monadic computation that treats still unhandled "errors" as "normal" values and stops when an "error" is finally handled.

This fact has several immediate practical consequences for monadic parser combinators and similar structures. More importantly, however, it can be generalized to provide a uniform error handling framework for *all* monadic error handling mechanisms we are aware of. It also provides some surprising perspectives on error handling in general.

Contents

*papers@oxij.org; preferably with paper title in the subject line

1 Extended Abstract

In this article we shall use Haskell programming language extensively for the purposes of precise expression of thought (including Haskell type class names for the names of the respective algebraic structures where appropriate, e.g. "Monad" instead of "monad").

- We note that the types of

  ```
  throw :: e -> c a
  catch :: c a -> (e -> c a) -> c a
  ```

  operators are special cases of Monadic `return` and `(>>=)` (`bind`) operators

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

  (substitute $[a \mapsto e, m \mapsto \lambda\_.c\ a]$ into their types, see sections 6 and 7).

- Hence, a type `c e a` with two indexes where `e` signifies a type of errors and `a` signifies a type of values is a Monad twice: once for `e` and once for `a`.

  ```
  class ConjoinedMonads c where
    return :: a -> c e a
    (>>=)  :: c e a -> (a -> c e b) -> c e b

    throw  :: e -> c e a
    catch  :: c e a -> (e -> c f a) -> c f a
  ```

  Moreover, for such a structure `throw` is a left zero for `(>>=)` and `return` is a left zero for `catch` (see sections 8 and 10).

- We prove that the type of the above `catch` is most general type for any Monadic structure `\a -> c e a` with additional `throw` and `catch` operators satisfying conventional computational rules (via simple unification of types for several equations that follow from operational semantics of said operators, see section 7). Or, dually, we prove that `(>>=)` has the most general type for expressing computations for Monadic structure `\e -> c e a` (with operators named `throw` and `catch`) with additional `return` and `(>>=)` operators satisfying conventional computational rules (see footnote 20).

- Substituting a Constant Functor for `c` into ConjoinedMonads above (i.e., fixing the type of errors) produces the definition of MonadError, and, with some equivalent redefinitions, MonadCatch (see section 12). Similarly, IO with similar redefinitions and with the usual caveats of remark 4 is a ConjoinedMonads instance too (see section 16).

- ExceptT (section 9) and some other lesser known and potentially novel concrete structures (see all sections with "Instance:" in the title, most interestingly, section 14) have operators of such a type and their semantics matches (or they can be redefined in an equivalent way such that the core part of the resulting structure then matches) the semantics of Monad exactly.

- Monad type class has a well-known "fish" representation where "`bind`" `(>>=)` operator is replaced by "`fish`" operator

  ```
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  ```

  and Monad laws are just monoidal laws.

  Hence, all those structures can be seen as a pairs of monoids over bi-indexed types with identity elements for respective `bind`s as left zeros for conjoined `bind`s (section 8). We find this symmetry to be hypnotic and generalize it in section 17.

- The answer to "Why didn't anyone notice this already?" seems to be that this structure cannot be expressed well in Haskell (see section 11).
- Meanwhile, it has at least several practically useful instances:
  - Parser combinators that are precise about errors they produce and that reuse common `Monad`ic combinators for both parsing and handling of errors. For instance, the type of `many` for such a parser combinator guarantees that it cannot throw any errors

    ```
    many :: c e a -> c f [a]
    ```

    (since `f` can be anything, it cannot be anything in particular) and

    ```
    choice :: [c e a] -> c e a
    ```

    is an instance of `foldM` (see section 13).
  - Conventional exceptions expressed using `Reader` `Monad` and second-rank `callCC` (the whole idea of which seems to be novel, see section 14).
  - Error-explicit `IO` (section 15), the latter and similar structures with similar motivation were proposed before, but they did not use the fact that their "other half" is a `Monad` too.

Every item on the above list, to our best knowledge, is a headline contribution.

2 Preliminaries

Most of the results of this paper are **language-agnostic** and can be applied (if not straight to practice, then at least to inform design choices) to any programming language (that permits at least two computationally distinguishable program states and some kind of dynamic control flow control) as our definition of an "*error*" in "error handling" is just "an abnormal program state causing execution of an abnormal code path" and both "abnormal"s can be arbitrarily defined (see footnote 2).

While most of our results are applicable to any programming language, we need *some* language to express them in and Haskell seems to be the most natural choice to host this discussion since

- most of the cited literature uses Haskell or some variant of ML;
- it has the largest number of error handling mechanisms in active use of all the programming languages we are aware of;
- as a consequence, most other programming languages implement a subset of Haskell's enormous library of error handling mechanisms;
- while it is not ideal for our purposes (Haskell cannot properly express the main result and the improper encoding of the main result is not particularly convenient, see section 11), it is expressive enough to show how a convenient encoding could have been implemented in theory;
- it is surprisingly popular for an "academic" language.

Using Haskell also allows this paper to be encoded as a set of Literate Haskell programs in a single Emacs Org-Mode tree [1, 2].[1]

Our preferred compiler is The Glorious Glasgow Haskell Compiler (GHC) [3] version 8.2 as we shall use a number of its extensions over Haskell 2010 [4] specification.

Readers unfamiliar with Haskell are advised to read through any tutorial introduction into Haskell at least until they start feeling like Haskell is just a syntax for school-level arithmetic with user-definable functions, lambdas, types, algebraic datatypes and type classes. After that it is recommended to look over Diehl's web-page [5], the table of contents (just the list of modules) of GHC's `base` package [6], and the types and descriptions of functions from the `Prelude` module of `base`.

The rest can be learned on-demand from sections 4 and 5 and cited documentation.

[1] The source code is available at https://oxij.org/paper/ExceptionallyMonadic/. It is also embedded straight into the PDF version of this article (click here  or look for "attachments" in your PDF viewer).

3 Introduction

Definition 1. *Generally, when program encounters an "*error*" all it can do is to switch to an "*exceptional*" execution path [7]. The latter can then either encounter an "*error*" itself or*

1. *gracefully "*terminate*" some part of the previous computation (including the whole program as a degenerate case) and continue (when there is something left to continue),*
2. *"*fix*" the "*problem*" and resume the computation as if nothing has happened.*

Error handling[2] is an algebraic subfield of the programming languages theory that studies this sort of seemingly simple control structures.

Different substitutions for "*error*", "*exceptional*" and "*terminate*" into definition 1 variant 1 and substitutions for "*error*", "*exceptional*", "*fix*" and "*problem*" into definition 1 variant 2 produce different error handling mechanisms. Some examples:

- Identity substitution for variant 1 gives programming with error codes, programming with algebraic datatypes [8, 9] that encode errors, programming with algebraic datatypes with errors [10, 11] (not the same thing), exceptions in conventional programming languages [7, 12–15] (with so called "termination semantics" [16, 16.6 Exception Handling: Resumption vs. Termination]), error handling with monads [17–22], monad transformers [23–25], Scheme's and ML's `call/cc` [26], and delimited `callCC` [25, 27, 28].
- Substituting "*unparsable string*", "*alternative*", "*backtrack*" for variant 1 gives monadic parser combinators [29].
- Identity substitution for variant 2 gives error handling in languages with so called "resumption semantics" [16, 16.6 Exception Handling: Resumption vs. Termination] like, for instance, Common LISP [30] (*condition handling*) and Smalltalk [14].
- Substituting "*effect*", "*effect handler*", *handle*", "*it*" for variant 1 or 2 (depending on the details of the calculus) produces effect systems [24, 31–35] and effect systems based on modal logic with names [36, 37].
- "*System call*", "*system call handler*", "*handle*", "*it*" for variant 2 produces conventional *system calls* [38].[3][4]
- Substituting "*signal*", "*signal handler*", "*handle*", "*it*", "*it*" for variant 2 gives hardware interrupts and POSIX signals [38].[5]

[2] Not a consensus term. Some people would disagree with this choice of a name as they would not consider some of our examples below to be about "errors". However, for the purposes of this article we opted into generalizing the term "*error*" of "error handling" instead of inventing new terminology or appropriating terminology like "exceptions", "interrupts", "conditions" or "effects" that has other very specific uses. To see the problem with the conventional terminology consider how would you define "program encountered an error" formally and generally for **any** abstract interpreter (you can not). Now consider the case where an interpreter is a sequence of interpreters interpreting one another. Clearly, what is an "*error*" for one interpreter can be considered normal execution for the next one. A simple example of such a structure is the `Maybe Monad` discussed in section 4.2.5 in which expressions using `do`-syntax never consider `Nothing`s while handling of said `Nothing`s by the `Monad`ic `(>>=)` operator is a completely ordinary `case` for the underlying Haskell interpreter. Hence, in this article we consider anything that matches definition 1 to be about *"error" handling*. If the reader still feels like disagreeing with our argument we advise mentally substituting every our use of "error" with something like "an abnormal program state causing execution of an abnormal code path" (where definitions of both "abnormal"s are interpreter-specific).

[3] Except in most UNIX-like operating systems system calls cannot call other system calls directly and have to use an equivalent kernel API instead.

[4] Indeed, algebraic effects from the point of view of an OS-developer are just properly typed system calls with nesting and modular handling.

[5] Indeed, POSIX signals and hardware interrupts are "system calls in reverse" (with some complications outside of the scope of this article): kernel and/or hardware raises and applications handle them.

The first complication of the above scheme is the question of whenever for a given error handling mechanism the "*error*" raising operator

1. passes control to a statically selected (lexically closest or explicitly specified) enclosing error handling construct (e.g. `throw` and `catch` in Emacs LISP [39], POSIX system calls and signals) or

2. the language does dynamic dispatch to select an appropriate error handler (like exceptions in most conventional languages like C++, Java, Python, etc do).

Another complication is ordering:

1. Most conventional programming languages derive their error handling from SmallTalk [14] and Common LISP [30] and the order in which the program handles "*errors*" corresponds to the order in which execution encounters them.

2. Meanwhile, some CPU ISAs[6] expose the internal non-determinism and allow different independent data-flows to produce hardware exceptions in non-deterministic manner (e.g. arithmetic instructions on DEC Alpha). So do Haskell [40] (see section 4.5) and, to some extent, C++ [41] programming languages.

Finally, another dimension of the problem is whenever the objects signifying "*errors*" (e.g. arguments of `throw`) are

1. first-class values (error codes, algebraic datatypes) as in most conventional languages,

2. labels or tags as in modal logic with names and, to some degree, with `call/cc` and `callCC`.

In short, despite its seemingly simple computational semantics, error handling is an algebraically rich field of programming languages theory.

Meanwhile, from the perspective of types there are several schools of thought about effects.

- The first one, started by Gifford and Lucassen [42–44] represents effects as type annotations. This works well in programming languages with eager evaluation, but becomes complicated in lazy languages (application in a lazy language delays effects until thunk's evaluation, hence type system has to either put nontrivial restrictions on the use of effects in expressions or annotate both arrows and values with effects, the latter, among other things, breaks type preservation of η-conversion since $\lambda x.fx$ moves effect annotation from the arrow to the result type).

- The second one, started by Moggi and Wadler [17, 19] confines effects to monadic computations. The latter can then be annotated with effect annotations themselves [45]. Monads work well for small programs with a small number of effects, but, it is commonly argued, they don't play as nice in larger programs because they lack in modularity [32] (hence, the need for monad transformers, which are then critiqued as hard to tame [34]) and produce languages with non-uniform syntax (pure functions look very different from monadic ones and functions that are useful in both contexts have to be duplicated, think e.g. `map` and `mapM`).

- The third one, started by Nanevski [36] represents effects using modal logic with names. Practical consequences of this way of doing things are unknown, as this construction didn't get much adoption yet.

[6] Instruction Set Architecture (ISA) is a specification that specifies a set of Operation Codes (OPcodes, which are a binary representation of an assembly language) with their computational semantics. "i386", "i686", "amd64" ("x86_64"), "aarch64", "riscv64", etc are ISAs.

In short, from type-theoretic point of view the progression of topics in the cited literature can be seen as pursuing calculi that are, at the same time, computationally efficient, algebraically simple (like monads), but modular (like effect systems).

Note, however, that all of those schools of thought consider exceptions to be effects, they only disagree about the way to represent the latter. Meanwhile, from a perspective of a programming language implementer, there are several problems with that world view:

- mechanisms that support resumption semantics are commonly disregarded as useless and computationally expensive error handling mechanisms (most notably [16, 16.6 Exception Handling: Resumption vs. Termination, pp. 390–393]),
- in particular, all popular programming languages implement builtin exceptions even if they have more general error handling mechanisms like *condition handling* in Common LISP and `call/cc` in Scheme and ML because those are just too computationally expensive for emulation of conventional exceptions [28],
- and even in languages with nothing but exceptions and termination semantics, high-performance libraries that do a lot of error handling frequently prefer not to use exceptions for performance reasons and to remove any non-local control-flow.

In short, from practical point of view *most* of those type-theoretic constructs are an overkill for *most* programs. Meanwhile, we are not aware of any non-ad-hoc language-agnostic algebraic structure that captures all of the exception handling (both `throw`ing, and `catch`ing) without introducing any other superfluous structure on top. In this article we shall demonstrate a fairly straightforward but surprisingly useful solution to this problem.

4 Not a Tutorial: Side A

While algebraic structures used in this article are simple, there are a lot of them. This section is intended as a reference point for all algebraic structures relevant in the context of error handling that are referenced in the rest of the paper (for reader's convenience and for high self-sufficiency of the Literate Haskell version). Most of those are usually assumed to be common knowledge among Haskell programmers. Note however, that this section is not intended to be a tutorial on either

- functional/declarative programming in general,
- Haskell language in particular (see section 2 for pointers),
- error handling in Haskell in general,
- practical usage of error handling structures discussed is this section in particular (we show only very primitive examples, if any; for the interesting ones the reader will have to look into citations and examples given in the original sources).

All structures of this section are ordered from semantically simple to more complex (that is, we do not topologically sort them by their dependencies in GHC sources). For the reasons of simplicity, uniformity, self-containment, and novel perspective some of the given definitions differ slightly from (but are isomorphic/equivalent to) the versions provided by their original authors. The most notable difference is the use of `Pointed` type class (see section 4.1.2) instead of conventional `Monad`ic `return` and `Applicative` `pure`. All structures are listed alongside references to the corresponding papers, documentation and original source code.

This section can be boring (although, we feel like most remarks and footnotes are not). On the first reading we advise to skip straight to section 5 and refer back to this section on demand.

4.1 Before-Monadic

This subsection describes type classes that have less structure than `Monad` but are useful for error handling nevertheless.

4.1.1 Monoid

GHC.Base from base [6] package defines Monoid type class as follows[7]

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  -- defined for performance reasons
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

and wants its instances to satisfy the following conventional equations ("Monoid laws")

```
-- `mempty` is left identity for `mappend`,
mempty `mappend` x == x

-- `mempty` is right identity for `mappend`,
x `mappend` mempty == x

-- `mappend` is associative,
x `mappend` (y `mappend` z)
  == (x `mappend` y) `mappend` z
```

and an additional constraint

```
-- and `mconcat` is extensionally
-- equal to its default implementation
mconcat == foldr mappend mempty
```

Signature and default implementation for mconcat is defined in the type class because mconcat is a commonly used function that has different extensionally equal intensionally non-equal definitions with varied performance trade-offs. For instance,

```
mconcat' :: Monoid a => [a] -> a
mconcat' = foldl' mappend mempty
```

(where foldl' is a strict left fold) is another definition that satisfies the law given above (since mappend is associative), but this implementation will not produce any superfluous thunks for strict mappend.

Monoids are not designed for error handling per se but programmers can use their neutral elements to represent an error and associative composition to ignore them. Whenever "ignoring" is "handling" is a matter of personal taste.

One of the simpler instances is, of course, a list

```
instance Monoid [a] where
  mempty = []
  mappend = (++)
```

and hence, for instance, functions generating errors can produce empty lists on errors and singleton lists on successes.

[7] Note that by following Pointed logic used below we should have split Monoid into two type classes, but since we will not use Monoids that much in the rest of the article we shall use the original definition as is.

4.1.2 Functor, Pointed, Applicative

Most of the error handling mechanisms that follow are Applicative Functors. GHC.Base from base [6] package defines those two algebraic structures as follows

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

infixl 4 <*>
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

and wants their instances to satisfy

```
-- `fmap` preserves identity
fmap id == id

-- `(<*>)` is `fmap` for pure functions
pure f <*> x == fmap f x
```

and some more somewhat more complicated equations [46]. We shall ignore those for the purposes of this article (we will never use them explicitly). Meanwhile, for the purposes of this article we shall split the pure function out of Applicative into its own Pointed type class and redefine Applicative using it as follows (this will simplify some later definitions).

```
class Pointed f where
  pure :: a -> f a

infixl 4 <*>
class (Pointed f, Functor f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
```

We shall give all definitions and laws using this hierarchy unless explicitly stated otherwise.

The most trivial example of Applicative is the Identity Functor defined in Data.Functor.Identity of base

```
newtype Identity a = Identity
  { runIdentity :: a }

instance Pointed Identity where
  pure = Identity

instance Functor Identity where
  fmap f (Identity a) = Identity (f a)

instance Applicative Identity where
  (Identity f) <*> (Identity x) = Identity (f x)
```

The most trivial example of a Functor that is not Applicative is Constant Functor defined in Data.Functor.Const of base as

```
newtype Const a b = Const
  { getConst :: a }

instance Functor (Const a) where
```

```haskell
  -- note that it changes type here
  fmap f (Const a) = Const a
  -- so the following would not work
  -- fmap f x = x
```

It is missing a `Pointed` instance. However, if the argument of `Const` is a `Monoid` we can define it as

```haskell
instance Monoid a => Pointed (Const a) where
  pure a = Const mempty

instance Monoid a => Applicative (Const a) where
  Const x <*> Const a = Const (mappend x a)
```

Remark 1. *One can think of `Applicative f` as representing* generalized function application *on structure `f`: `pure` lifts pure values into `f` while `(<*>)` provides a way to apply functions to arguments over `f`. Note however, that `Applicative` is not a structure for representing* generalized functions *(e.g. `Applicative` gives no way to compose functions or to introduce lambdas, unlike the `Monad`, see remark 2).*

4.1.3 Alternative

`Control.Applicative` module of `base` [6] defines `Alternative` class as a monoid on `Applicative Functor`s.[7]

```haskell
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  -- defined for performance reasons
  some :: f a -> f [a]
  some v = fmap (:) v <*> many v

  many :: f a -> f [a]
  many v = some v <|> pure []
```

requiring monoid laws to hold for `empty` and `(<|>)`

```haskell
-- `empty` is left identity for `(<|>)`,
empty <|> x == x

-- `empty` is right identity for `(<|>)`,
x <|> empty == x

-- `(<|>)` is associative,
x <|> (y <|> z)
  == (x <|> y) <|> z

-- and both `some` and `many` are
-- extensionally equal to their
-- default implementations
some v == fmap (:) v <*> many v
many v == some v <|> pure []
```

Combinators `some` and `many`, similarly to `mconcat`, commonly occur in functions handling `Alternative`s and can have different definitions varying in performance for different types. The

most common use of `Alternative` type class is parser combinators (section 5.2) where `some` and `many` coincide with `+` ("one or more") and `*` ("zero or more", Kleene star) operators from regular expressions/EBNF. Before the introduction of `Alternative` that role was played by now deprecated `MonadPlus` class, currently defined in `Control.Monad` of `base` as follows

```
class (Alternative m, Monad m) => MonadPlus m where
  mzero :: m a
  mzero = empty

  mplus :: m a -> m a -> m a
  mplus = (<|>)
```

We shall give example instance and usage of `Alternative` in section 5.2.

4.2 Purely Monadic

This subsection describes algebraic structures that involve `Monad` type class and its instances.

4.2.1 Monad definition

`GHC.Base` from `base` [6] defines `Monad` in the following way using the original (i.e. not `Pointed`) hierarchy (also, at the time of writing `base` uses a bit uglier definition which is discussed in section 4.2.3)

```
infixl 1 >>=
class Applicative m => Monad m where
  return  :: a -> m a
  (>>=)   :: m a -> (a -> m b) -> m b
```

and wants its instances to satisfy the following equations known as "`Monad` laws"

```
-- `return` is left identity for `(>>=)`
return a >>= f == f a

-- `return` is right identity for `(>>=)`
f >>= return == f

-- `(>>=)` is associative
(f >>= g) >>= h == f >>= (\x -> g x >>= h)
```

Note that this definition also expects the following additional "unspoken laws" from its parent structures (see section 4.3 for definitions of `liftM` and `ap`).

```
fmap == liftM
pure == return
(<*>) == ap
```

Moreover, we feel that the name "return" itself is an unfortunate accident since `return` only injects pure values into `m` and does not "return" anywhere. We shall avoid that problem and simplify the above equations by redefining `Monad` using `Pointed` hierarchy instead

```
infixl 1 >>=
class Applicative m => Monad m where
  (>>=)   :: m a -> (a -> m b) -> m b

-- for backward-compatibility
return :: Monad m => a -> m a
return = pure
```

Remark 2. *Note that while* `Applicative` *is too weak to express* generalized functions *(remark 1),* `Monad`*, in some sense, is too strong since* `(>>=)` *combines function composition (the whole type) with lambda introduction (the type of the second argument). This might be easier to see with the definition given in section 4.2.2.*

What is the "just right" structure for representing a generalized function *is a matter of debate: some would state "an* `Arrow`*!" [47], others "a (Cartesian Closed)* `Category`*!" [48], yet others might disagree with both.*

A very common combinator used with `Monad`s bears a name of `(>>)` and can be defined as

```
(>>) :: Monad m => m a -> m b -> m b
a >> b = a >>= const b
       -- a >>= \_ -> b
```

The following subsections will provide many example instances.

4.2.2 MonadFish

A somewhat lesser known but equivalent way to define `Monad` is to define `(>>=)` in "fish" form as follows

```
infixl 1 >=>
class Applicative m => MonadFish m where
  (>=>)   :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

This way `Monad` laws become `Monoid` laws

```
-- `pure` is left identity for `(>=>)`
pure >=> f == f

-- `pure` is right identity for `(>=>)`
f >=> pure == f

-- `(>=>)` is associative
(f >=> g) >=> h == f >=> (g >=> h)
```

Lemma 1. *`(f >=> g) . h == (f . h) >=> g`*

Proof. For pure values `(>=>)` is a composition with flipped order of arguments `(.)`

```
instance MonadFish Identity where
  f >=> g = g . runIdentity . f
```

In other words, `f >=> g == g . f`, which gives the following

```
(f >=> g) . h == h >=> (f >=> g)
              == (h >=> pure) >=> (f >=> g)
              == ((h >=> pure) >=> f) >=> g
              == (h >=> f) >=> g
              == (f . h) >=> g
```

which, with some abuse of notation (`(>=>)` is not heterogeneous, the above lifts pure values into `m` with `pure`), can be written simply as

```
(f >=> g) . h == h >=> (f >=> g)
              == (h >=> f) >=> g
              == (f . h) >=> g
```

□

Lemma 2. *`Monad` and `MonadFish` define the same structure.*

Proof. The cross-definitions:

```
instance (Applicative m, Monad m) => MonadFish m where
  f >=> g = \a -> (f a) >>= g -- (1)

instance {-# OVERLAPPABLE #-}
         (Applicative m, MonadFish m) => Monad m where
  ma >>= f = (id >=> f) ma -- (2)
```

- (1) implies (2):

```
ma >>= f == (id >=> f) ma
         == (\a -> id a >>= f) ma
         == ma >>= f
```

- (2) implies (1):

```
f >=> g == \a -> (f a) >>= g
        == \a -> (id >=> g) (f a)
        == (id >=> g) . f
        == (id . f) >=> g
        == f >=> g
```

□

4.2.3 Monad's fail and MonadFail

Section 4.2.1 did not give the complete definition of `Monad` as is defined in the current version of `base` [6]. Current `GHC.Base` module defines `Monad` in the following way using the original (not `Pointed`) hierarchy

```
infixl 1 >>=
class Applicative m => Monad m where
  return  :: a -> m a
  (>>=)   :: m a -> (a -> m b) -> m b

  fail    :: String -> m a
  fail s  = error s
```

Note the definition of the `fail` operation. That function is invoked by the compiler on pattern match failures in `do`-expressions (see section 4.7 for examples, see section 4.5.6 for the definition of `error`), but it can also be called explicitly by the programmer in any context where the type permits to do so.

The presence of `fail` in `Monad` class is, clearly[8], a hack. There is an ongoing effort (aka "`MonadFail` proposal", "MFP") to move this function from `Monad` to its own type class defined as follows (in both hierarchies)

```
class Monad m => MonadFail m where
  fail :: String -> m a
  fail s  = error s
```

[8] It involves an error handling mechanism that is more complicated than the thing itself. It creates semantic discrepancies (e.g. `Maybe` is not equivalent to `Either ()`, see section 4.2.6).

As of writing of this article the new class is available from `Control.Monad.Fail`, but `fail` from the original `Monad` is not even deprecated yet. We shall use `MonadFail` instead of the original `fail` in our hierarchy for simplicity.

4.2.4 Identity monad

We can define the following `Monad` and `MonadFail` instances for the `Identity Functor`

```
instance Monad Identity where
  (Identity x) >>= f = f x

instance MonadFail Identity where
  -- default implementation
```

despite this instance it is still usually referenced as "`Identity Functor`" even though it is also an `Applicative` and a `Monad`.

4.2.5 Maybe monad

The simplest form of `Monad`ic error handling (that is, not just "error ignoring") can be done with `Maybe` datatype and its `Monad` instance defined in `Data.Maybe` of `base` [6] equivalently to

```
data Maybe a = Nothing | Just a

instance Pointed Maybe where
  pure = Just

instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= _ = Nothing

instance MonadFail Maybe where
  -- custom `fail`
  fail _         = Nothing
```

The `pure` operator simply injects a given value under `Just` constructor, while the definition of `(>>=)` ensures that

- injected values are transparently propagated further down the computation path,
- computation stops as soon as the first `Nothing` gets emitted.

In other words, `Maybe Monad` is `Identity Monad` that can stop its computation on request. A couple of examples follow

```
maybeTest1 :: Maybe Int
maybeTest1 = do
  x <- Just 1
  pure x

maybeTest2 :: Maybe Int
maybeTest2 = do
  x <- Just 1
  pure x
  Nothing
  Just 2

maybeTest = maybeTest1 == Just 1
         && maybeTest2 == Nothing
```

4.2.6 Either monad

`Either` datatype is defined in `Data.Either` of `base` [6] equivalently to

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

instance Pointed (Either e) where
  pure = Right

instance Monad (Either e) where
  Left  l >>= _ = Left l
  Right r >>= k = k r

instance MonadFail (Either e)
  -- default `fail`
```

`Either` is a computation that can stop and report a given value (the argument of `Left`) when falling out of `Identity` execution. The intended use is similar to `Maybe`

```
eitherTest1 :: Either String Int
eitherTest1 = do
  x <- Right 1
  pure x

eitherTest2 :: Either String Int
eitherTest2 = do
  x <- Right 1
  pure x
  Left "oops"
  Right 2

eitherTest = eitherTest1 == Right 1
          && eitherTest2 == Left "oops"
```

Purely by its datatype definition `Maybe a` is isomorphic to `Either () a` (where `()` is Haskell's name for the `unit` type), but their `Monad` instances (in the original hierarchy, `MonadFail` in our hierarchy) differ: `Maybe` has non-default `fail`, while `Either` does not. This produces some observable differences discussed in section 4.7.

4.3 An intermission on Monadic boilerplate

Haskell does not support default definitions for functions in superclasses that use definitions given in subclasses. That is, Haskell has no syntax to define `Functor` and `Applicative` defaults from `Monad` instance of the same type.

Which is why to compile the code above we have to borrow a couple of functions from `Control.Monad` of `base`

```
liftM :: (Monad m)
      => (a -> b) -> m a -> m b
liftM f ma = ma >>= pure . f

ap :: (Monad m)
   => m (a -> b) -> m a -> m b
ap mf ma = mf >>= \f -> liftM f ma
```

and use them to define

```haskell
instance Functor Maybe where
  fmap = liftM

instance Applicative Maybe where
  (<*>) = ap
```

and analogously for `Either`. For all the listings that follow we shall silently hide this type of boilerplate code from the paper version where appropriate (it can still be observed in the Literate Haskell version).

4.4 MonadTransformers

The problem with `Monad`s is that they, in general, do not compose. `Monad` transformers [23] provide a systematic way to define structures that represent "a `Monad` with a hole" that allow computations from an inner `Monad` `m` to be `lift`ed through a hole in an outer `Monad (t m)` (`t` transforms monad `m`, hence "monad transformer"). The main type class is defined in `Control.Monad.Trans.Class` module of `transformers` [25] package as follows

```haskell
class MonadTrans t where
  lift :: (Monad m) => m a -> t m a
```

Haskell type class system is not flexible enough to encode the requirement that `t m` needs to be a `Monad` in a single definition, so it has to be encoded in every instance by using the following instance schema

```haskell
instance Monad m => Monad (t m) where
  -- ...
```

Different `MonadTrans`formers (`t1`, `t2` … `tn`) can then be composed with an arbitrary `Monad` `m` (usually called "*the* inner `Monad`") using the following scheme

```haskell
newtype comp m a = t1 (t2 (.. (tn (m a))))
```

and the whole composed stack would get a `Monad` instance inferred for it. Popular choices for the inner `Monad` `m` include `Identity Functor` and `IO Monad` (see section 4.5).

In short, `MonadTrans`formers are, pretty much, composable `Monad`ic structures. The following subsections will provide many example instances. For an in-depth tutorial readers are referred to [49] and [23].

4.4.1 Identity

The simplest `MonadTrans`former is `IdentityT` defined in `Control.Monad.Trans.Identity` of `transformers` [25] package equivalently to

```haskell
newtype IdentityT m a = IdentityT
  { runIdentityT :: m a }

instance MonadTrans IdentityT where
  lift = IdentityT

instance Monad m
      => Pointed (IdentityT m) where
  pure = lift . pure

instance Monad m
      => Monad (IdentityT m) where
```

```
  x >>= f = IdentityT $ do
    v <- runIdentityT x
    runIdentityT (f v)
```

Remark 3. *Note that `IdentityT` MonadTransformer is different from `Identity` Monad and cannot be redefined as simply*

```
type IdentityT' m a = Identity (m a)
```

(even though the datatype definition matches exactly) because `IdentityT` "inherits" `Monad` implementation from its argument `m` while `Identity` provides its own. I.e. `IdentityT` is an identity on MonadTransformers while `Identity` is an identity on types.

In particular, for `Identity (Maybe a)`

```
pure == Identity
```

while for `IdentityT Maybe a`

```
pure == IdentityT . pure == IdentityT . Just
```

4.4.2 Maybe

Transformer version of `Maybe` called `MaybeT` is defined in `Control.Monad.Trans.Maybe` from `transformers` [25] package equivalently to

```
newtype MaybeT m a = MaybeT
  { runMaybeT :: m (Maybe a) }

instance MonadTrans MaybeT where
  lift = MaybeT . liftM Just

instance Monad m
      => Pointed (MaybeT m) where
  pure = lift . pure

instance Monad m
      => Monad (MaybeT m) where
  x >>= f = MaybeT $ do
    v <- runMaybeT x
    case v of
      Nothing -> pure Nothing
      Just y  -> runMaybeT (f y)

instance MonadFail m
      => MonadFail (MaybeT m) where
  fail _ = MaybeT (pure Nothing)
```

4.4.3 Except

Transformer version of `Either` for historical reasons bears a name of `ExceptT` and is defined in `Control.Monad.Trans.Except` from `transformers` [25] package equivalently to

```
newtype ExceptT e m a
  = ExceptT { runExceptT
            :: m (Either e a) }
```

```
instance MonadTrans (ExceptT e) where
  lift = ExceptT . liftM Right

instance Pointed m
      => Pointed (ExceptT e m) where
  pure a = ExceptT $ pure (Right a)

instance Monad m
      => Monad (ExceptT e m) where
   m >>= k = ExceptT $ do
     a <- runExceptT m
     case a of
       Left  e -> pure (Left e)
       Right x -> runExceptT (k x)

instance MonadFail m
      => MonadFail (ExceptT e m) where
   fail = ExceptT . fail
```

The main attraction of `ExceptT` for the purposes of this article is the fact that it provides its own non-imprecise non-dynamic-dispatching `throw` and `catch` operators defined as

```
throwE :: (Monad m) => e -> ExceptT e m a
throwE = ExceptT . pure . Left

catchE :: (Monad m) =>
    ExceptT e m a
    -> (e -> ExceptT f m a)
    -> ExceptT f m a
m `catchE` h = ExceptT $ do
    a <- runExceptT m
    case a of
        Left  l -> runExceptT (h l)
        Right r -> pure (Right r)
```

There also exists deprecated `ErrorT` (defined in `Control.Monad.Trans.Error` from `transformers` package) which at the time of writing has exactly the same definition as `ExceptT`

```
newtype ErrorT e m a
  = ErrorT { runErrorT
           :: m (Either e a) }
```

but its instances require type class `Exception` (see section 4.5.4) from its argument `e`. Older versions of `transformers` package made this requirement in the definition of `ErrorT`

```
newtype ErrorT e m a
  = Exception e =>
    ErrorT { runErrorT
           :: m (Either e a) }
```

but that mechanism itself was deprecated awhile ago.

4.4.4 Reader and State

While there seems to be no way to directly use `Reader` and `State` `Monad`s for error handling, these structures are used in `IO` `Monad` of section 4.5 and parser combinators of section 5.2. This seems to be as good place as any to define them.

`Reader` `Monad` is defined in `Control.Monad.Trans.Reader` module of `transformers` [25] package equivalently to

```
type Reader s = ReaderT s Identity

newtype ReaderT s m a = ReaderT { runReaderT :: s -> m a }

instance MonadTrans (ReaderT s) where
  lift m = ReaderT $ \_ -> m

instance Pointed m => Pointed (ReaderT s m) where
  pure a = ReaderT $ \_ -> pure a

instance Monad m => Monad (ReaderT s m) where
  m >>= k  = ReaderT $ \s -> do
    a <- runReaderT m s
    runReaderT (k a) s

instance MonadFail m => MonadFail (ReaderT s m) where
  fail str = ReaderT $ \_ -> fail str
```

Meanwhile, `State` `Monad` is defined in `Control.Monad.Trans.State.Lazy` and `Control.Monad.Trans.State.Strict` modules (the difference between them does not matter for the purposes of this article, so we shall ignore it) from `transformers` [25] package equivalently to

```
type State s = StateT s Identity

newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }

instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> do
    a <- m
    pure (a, s)

instance Pointed m => Pointed (StateT s m) where
  pure a = StateT $ \s -> pure (a, s)

instance Monad m => Monad (StateT s m) where
  m >>= k  = StateT $ \s -> do
    (a, s') <- runStateT m s
    runStateT (k a) s'

instance MonadFail m => MonadFail (StateT s m) where
  fail str = StateT $ \_ -> fail str
```

Both structures provide `Monad`ic structures that handle state. `ReaderT` simply applies variable `s` throughout its whole computation via its `(>>=)` operator thus supplying computations with a *context* (i.e. read-only *state*). Meanwhile, `StateT` chains its `s` between computations, thus providing computations with a (read-write) *state*.

4.5 Imprecise exceptions

As we mentioned in the introduction, GHC implements *imprecise exceptions* mechanism proposed in [40]. Such exceptions look superficially similar to those of C++/Java/Python/etc but differ in two important aspects.

Firstly, GHC imprecise exceptions in pure computations are completely imprecise. That is, evaluation of `(a `op` b)` with `a` raising `e` and `b` raising `f` (and assuming `op` can evaluate either argument first) can raise either or even both (on different evaluations) of `e` and `f`. Haskell is not the only language that does this, C++, for instance, defines *sequence points* that serve the same purpose [41]. However, in GHC the order in which exception are raised is limited only by data dependencies, while C++'s sequence points add some more ordering on top.

Secondly, the C++/Java/Python exceptions have dynamic dispatch builtin, while GHC's dynamically dispatched exceptions are implemented as a library on top of statically dispatched exceptions. To be more specific

- on the base level GHC runtime defines `raise#` and `catch#` operations for which `raise#` "simply"[9] unwinds the stack to the closest `catch#` (i.e. `raise#` is "just"[9] a `GOTO`; `cast`ing, re-`raise`ing, `finally`, etc are left for the libraries to implement and are not builtins),
- on top of that GHC libraries then provide dynamically dispatched exceptions by `cast`ing elements of `Typeable` types from/to `SomeException` existential type [50].

In the following subsections we shall discuss the details of the actual implementation.

4.5.1 IO

GHC defines the mystical `IO` `Monad` in `GHC.Types` (the types) and `GHC.Base` (the instances), pretty much, as a `State` `Monad` (see section 4.4.4) on `State#` `RealWorld` (definitions of both of which are beyond the scope of this article)

```
type IO# a = State# RealWorld
          -> (# State# RealWorld, a #)

newtype IO a = IO { runIO :: IO# a }

instance Pointed IO where
  pure a = IO $ \s -> (# s, a #)

instance Monad IO where
  m >>= f = IO $ \s -> case runIO m s of
    (# s', a #) -> runIO (f a) s'
```

The `IO#` definition given above is not actually in GHC but without it all of the definitions below become unreadable. We also renamed `unIO` to `runIO` for uniformity with `State`. Note however, that we did not swap the elements of the result tuple of `IO#` to match those of `State` since that would make it incompatible with GHC runtime we reuse in Literate Haskell version.

Remark 4. *Note that `IO` is not a proper `Monad` since it cannot satisfy the laws simply for the fact that `RealWorld` cannot have an equality.*[10]

In this article, however, for the purposes of formal arguments involving `IO` we shall treat `IO` as if it was just a `State` over some state type with some simple denotational semantics (although,

[9] We put "simply" and "just" in quotes since unwinding of the stack must unwind into the lexically correct handler which is nontrivial in a lazy language like Haskell where thunks can be evaluated in an environment different from the one they were created in. In short, thunks must capture exception handlers as well as variables.

[10] Although `IO` can be reformulated as a free `Monad` made of "requests to the interpreter" and continuations if one is willing to forget about the internal structure of the `RealWorld` [51].

possibly unknown value). This, of course, immediately disqualifies our proofs for `IO` from using non-determinism, hence, for instance, we will not be able to prove things about imprecise exceptions or threads.

The alternative would be to split every lemma and theorem mentioning `IO` into two: one for a `RawMonad` (`Monad` without laws) for cases mentioning `IO`, and one for `Monad` for all other cases. This would make a very little practical sense for this article since we will not attempt proofs involving non-determinism anyway.

4.5.2 raise# and catch#

Primitive `raise#` and `catch#` operations are "defined" (those, of course, are just stubs to be replaced by references to the actual implementations in GHC runtime) in `GHC.Prim` module like follows

```
raise# :: a -> b
raise# = raise#

catch# :: IO# a -> (b -> IO# a)
       -> IO# a
catch# = catch#
```

Evaluating `raise#` "simply"[9] unwinds computation stack to the point of the closet `catch#` with the appropriate type and applies raised value to the second argument of the latter. Note, however, that while the type of `raise#` permits its use anywhere in the program, `catch#` is sandboxed to `IO#` on the lowest observable level and GHC provides no "`unsafeCatch`". This allows GHC to perform many useful optimizations that influence evaluation order without exposing pure computations to non-determinism.

4.5.3 Typeable

GHC implements dynamic casting with `Typeable` type class. The details of its actual implementation are beyond the scope of this article. For our purposes it suffices to say that it is a type class of types that have type representations that can be compared at runtime

```
class Typeable a where
  -- magic beyond the scope of this article
```

and it provides a `cast` operation with the following type signature that shows that it compares said representations of types of its argument and result and either returns its argument value wrapped in `Just` constructor when the types match or `Nothing` else

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

Interested readers should inspect the source code of `Data.Typeable` module of `base` [6].

4.5.4 Exception

On top of `Typeable` in `GHC.Exception` module of `base` [6] GHC provides the `Exception` type class that casts values to and from `SomeException` existential type (the following syntactic `forall` should be read as type-theoretic `exists`, for historic reasons)

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

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

  toException = SomeException
  fromException (SomeException e) = cast e

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

instance Exception SomeException where
  toException = id
  fromException x = Just x
```

4.5.5 throw and catch

Finally, `throw` and `catch` operators defined in `GHC.Exception` module of `base` [6] use all of the above to implement dynamic dispatch of exceptions.

The `throw` operator simply wraps given exception into `SomeException` and `raise#`s

```
throw :: Exception e => e -> a
throw e = raise# (toException e)
```

The `catchException` operator defined in `GHC.IO` does the actual dynamic dispatch

- it `catch#`es an exception produced by its first argument ("computation"),
- tries to `cast` it to a type expected by its second argument ("handler") and either calls the latter on success, or `raise#`s again on failure.

```
catchException :: Exception e
               => IO a -> (e -> IO a)
               -> IO a
catchException (IO io) handler
  = IO $ catch# io handler'
  where
    handler' e = case fromException e of
          Just  f -> runIO (handler f)
          Nothing -> raiseIO# e
```

The `catch` operator simply calls `catchException` after forcing its first argument into a thunk with `lazy` operator (this wrapping is necessary to prevent GHC from performing strictness analysis on the "computation" to prevent its evaluation before the exception is even raised; this fact can be ignored for the purposes of this article) which is yet another special GHC runtime function (this time, extentionally equal to its definition, i.e. identity).

```
lazy :: a -> a
lazy x = x

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

That is, `catch` is extentionally equal to `catchException`. `Control.Exception` module of `base` simply reexports `throw`, `catch`, and `Exception` type class and implements a bunch of practically convenient combinators using them.

We should also mention that older versions of `base` package had another special `catch` that handled only `IOError`s defined in `Prelude` and `System.IO.Error` respectively. Those were deprecated in 2011 and as of writing of this article are completely gone from current version of `base`. But they are are occasionally mentioned in tutorials, usually in the context of "don't use `catch` from `Prelude`, use the one from `Control.Exception`", nowadays the `catch` from `Prelude` *is* the `catch` from `Control.Exception`.

4.5.6 error and undefined

`error` and `undefined` primitives are defined in `GHC.Err` of `base` as follows

```
newtype ErrorCall = ErrorCall String

instance Exception ErrorCall where

error :: String -> a
error s = throw (ErrorCall s)

undefined :: forall a . a
undefined = error "Prelude.undefined"
```

Actually, this implementation is taken from the older version of `base`, modern version also implements call stack capture, which is beyond the scope of this article. Interested readers are referred to the source code of `GHC.Err`.

4.6 Precise raiseIO# and throwIO

Besides imprecise exceptions GHC's `IO` also has operators for precise exceptions a-la `ExceptT` defined in `GHC.Prim` and `GHC.Exception` as follows

```
raiseIO# :: a -> IO# b
raiseIO# = raiseIO#

throwIO :: Exception e => e -> IO a
throwIO e = IO $ raiseIO# (toException e)
```

While `throwIO` has a type that is an instance of `throw`, their semantics differ: `throwIO` produces `Monad`ic actions while `throw` produces values. For example, both functions in the following example will raise `SomethingElse`, not `ErrorCall`.

```
data SomethingElse = SomethingElse

instance Exception SomethingElse where

throwTest :: IO ()
throwTest = do
  let x = throw (ErrorCall "lazy")
  pure (Right x)
  throwIO SomethingElse

throwTest' :: IO ()
throwTest' = do
  let x = throw (ErrorCall "lazy")
  pure x
  throwIO SomethingElse
```

The `catch` operator, however, can be reused for handling both imprecise and precise exceptions.

Remark 5. *In other words, we can say that `IO` has two different exception mechanisms (precise and imprecise exceptions) with a single exception handling mechanism (`catch`). (And this is pretty weird.)*

4.7 Non-exhaustive patterns

As a side note, non-exhaustive pattern matches (and `case`s) `throw` `PatternMatchFail` exception, while the default `fail` implementation calls `error` which `throws` `ErrorCall`.

```
{-# LANGUAGE ScopedTypeVariables #-}

import Control.Exception

check t =
  (evaluate t >> print "ok")
  `catch`
  (\(e :: PatternMatchFail)
    -> print "throws PatternMatchFail")
  `catch`
  (\(e :: ErrorCall)
    -> print "throws ErrorCall")

patFail 1 x = case x of 0 -> 1
fail1 = patFail 1 1
fail2 = patFail 2 2
maybeDont = do { 1 <- Just 1 ; return 2 }
maybeFail = do { 0 <- Just 1 ; return 2 }
eithrDont = do { 1 <- Right 1 ; return 2 }
eithrFail = do { 0 <- Right 1 ; return 2 }

testPatterns = do
  check fail1     -- throws PatternMatchFail
  check fail2     -- throws PatternMatchFail
  check maybeDont -- ok
  check maybeFail -- ok (`Nothing`)
  check eithrDont -- ok
  check eithrFail -- throws ErrorCall
```

4.8 Monadic generalizations

In previous subsections we have seen a plethora of slightly different error handling structures with different `throw` and `catch` operators. In this subsection we shall describe several Hackage packages that provide structures that try to unify this algebraic zoo.

4.8.1 MonadError

`MonadError` class (`Control.Monad.Error.Class` from `mtl` [52] package) is defined as

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

This structure simply generalizes `ExceptT`

```
instance Monad m => MonadError e (ExceptT e m) where
  throwError = throwE
  catchError = catchE
```

in a way that is transitive over many other MonadTransformers, for instance

```
-- (these require UndecidableInstances GHC extension, however)

instance MonadError e m => MonadError e (IdentityT m) where
  throwError = lift . throwError
  catchError a h = IdentityT $ catchError (runIdentityT a) (runIdentityT . h)

instance MonadError e m => MonadError e (MaybeT m) where
  throwError = lift . throwError
  catchError a h = MaybeT $ catchError (runMaybeT a) (runMaybeT . h)
```

4.8.2 MonadThrow and MonadCatch

MonadThrow and MonadCatch classes (Control.Monad.Catch from exceptions [53]) are defined as[11]

```
class Monad m => MonadThrow m where
  throwM :: Exception e => e -> m a

class MonadThrow m => MonadCatch m where
  catchM :: Exception e
         => m a -> (e -> m a) -> m a
```

This structure, too, generalizes ExceptT

```
instance MonadThrow m => MonadThrow (ExceptT e m) where
  throwM = lift . throwM

instance MonadCatch m => MonadCatch (ExceptT e m) where
  catchM x f = ExceptT $ catchM (runExceptT x) (runExceptT . f)
```

and is transitive over common MonadTransformers

```
-- (this time without UndecidableInstances)

instance MonadThrow m => MonadThrow (IdentityT m) where
  throwM = lift . throwM

instance MonadCatch m => MonadCatch (IdentityT m) where
  catchM x f = IdentityT $ catchM (runIdentityT x) (runIdentityT . f)

instance MonadThrow m => MonadThrow (MaybeT m) where
  throwM = lift . throwM

instance MonadCatch m => MonadCatch (MaybeT m) where
  catchM x f = MaybeT $ catchM (runMaybeT x) (runMaybeT . f)
```

but it constraints argument e to type class Exception and it also generalizes the imprecise exceptions

[11] Except for the fact that MonadCatch from exceptions names its operator catch, not catchM, we renamed it for uniformity and so that it would not be confused with the operator from Control.Exception.

```
instance MonadThrow IO where
  throwM = throw

instance MonadCatch IO where
  catchM = catch
```

The latter fact complicates the use of these two structures somewhat since one can not be sure about the dynamic-dispatch part of the semantics without actually looking at the definitions for a particular instance.

5 Not a Tutorial: Side B

This section, logically, is a continuation of section 4. However, in contrast to that section this section discusses non-basic structures that are of particular importance to the rest of the article. While this section does not introduce any non-trivial novel ideas, some perspectives on well-known ideas seem to be novel.

5.1 Continuations

When speaking of "continuations" people usually mean one or more of the three related aspects explained in this subsection.

5.1.1 Continuation-Passing Style

Any (sub-)program can be rewritten into Continuation-Passing Style (CPS) [54, 55] by adding a number of additional *continuation* arguments to every function and tail-calling into those arguments with the results-to-be at every return point instead of just returning said results.

For instance, the following pseudo-Haskell program

```
foo =
  if something
     then Result1 result1
     else Result2 result2

bar = case foo of
  Result1 a -> bar1 a
  Result2 b -> bar2 b
```

can be transformed into (here we CPS-ignore `something` and the `if` for illustrative purposes)

```
fooCPS cont1 cont2 =
  if something
    then cont1 result1
    else cont2 result2

barCPS = fooCPS bar1 bar2
```

In conventional modern low-level imperative terms this transformation requires all functions to receive their return addresses as explicit parameters instead of `pop`ing them from the bottom of their stack frame with `ret` operation and give control to other functions with `jmp` instructions.

The latter, of course, means that we can treat "*normal*" programs (in which all functions have a single return address) as a degenerate case of programs written in "*implicit-CPS*" (in fact, `Cont Monad` of section 5.1.3 is exactly such an "*implicit-CPS*") — a syntactic variant of CPS in which

- every function has an implicit argument that specifies a default return address (which is set to the next instruction following a function call by default)

- that can be reached from the body of the function by tail-calling a special symbol that `jmp`s to the implicitly given address

Finally, one can even imagine a computer with a "*CPS-ISA*" (i.e. an ISA where each instruction explicitly specifies its own return address) in which case all programs for such a computer would have to be translated into an explicit CPS form to be executed. In fact, drum memory-based computers like IBM 650 had exactly such an ISA. From the point of view of an IBM 650 programmer modern conventional CPUs simply convert their non-CPS OPcodes into their CPS forms on the fly, thus applying CPS-transform to any given program on the fly.

Returning to the pseudo-Haskell listing above, note that programs written in CPS

- introduce a linear order on their computations, hence they are not particularly good for parallel execution,
- consume somewhat more memory in comparison to their "*normal*" representations (as they have to handle more explicit addresses),
- can have poorer performance on modern conventional CPUs (since said CPUs split their branch predictors into "jump" and "call" units and the latter unit rests completely unused by CPS programs),
- are harder to understand.

However, the advantage of the CPS form is that it allows elimination of duplicate computations. For instance, in the example above `foo` produces different results depending on the value of `something` and `bar` has to duplicate that choice (but not the computation of `something`) again by switching `case`s on the result of `foo`. Meanwhile, `barCPS` is free from such an inefficiency. Applying this transformation recursively to a whole (sub-)program allows one to transform the (sub-)program into a series of tail calls whilst replacing all constructors and eliminators in the (sub-)program with tail calls to newly introduced continuation arguments and `case` bodies respectively.

The logical mechanic behind this transformation is a technique we call *generalized Kolmogorov's translation* (since it is a trivial extension of Kolmogorov's translation[56]) of types of functions' results. That is, double negation followed by rewriting by well-known isomorphisms until formula contains only arrows, bottoms and variables followed by generalizing bottoms by a bound variable.

For instance, the result of a function of type

$$i \to j \to b$$

is b, which can be doubly negated as

$$\neg\neg b$$

$$(b \to \bot) \to \bot$$

and generalized to either of

$$\forall c.(b \to c) \to c$$

$$\lambda c.(b \to c) \to c$$

which allows us to generalize the whole function to either of

$$former = \forall c.i \to j \to (b \to c) \to c$$

$$latter = \lambda c.i \to j \to (b \to c) \to c$$

depending on the desired properties:

- the former term requires a rank-2 type system but it does not add any new type lambdas or free type variables, thus keeping the transformation closed,
- the latter term does not need rank-2 types, but it requires tracking of these new type variables,

- the latter term also retains full control over c variable, (for instance, it can produce the former term in rank-2 type system on demand with $\forall c.latter\ c$).

Similarly, `Either a b` may be seen as logical $a \vee b$ which can be rewritten as

$$\neg\neg(a \vee b)$$
$$\neg(\neg a \wedge \neg b)$$
$$(a \to \bot \wedge b \to \bot) \to \bot$$
$$(a \to \bot) \to (b \to \bot) \to \bot$$

and a pair of `(a, b)` is logical $a \wedge b$ and can be rewritten as

$$\neg\neg(a \wedge b)$$
$$\neg(a \wedge b) \to \bot$$
$$(a \wedge b \to \bot) \to \bot$$
$$(a \to b \to \bot) \to \bot$$

Hence, $i \to j \to (a \vee b)$ can be rewritten into either of

$$\forall c.i \to j \to (a \to c) \to (b \to c) \to c$$
$$\lambda c.i \to j \to (a \to c) \to (b \to c) \to c$$

and $i \to j \to (a \wedge b)$ into either of

$$\forall c.i \to j \to (a \to b \to c) \to c$$
$$\lambda c.i \to j \to (a \to b \to c) \to c$$

5.1.2 Scott-encoding

A technique of applying generalized Kolmogorov's translation to datatypes and their constructors and eliminators instead of normal functions in a (sub-)program is called Scott-encoding (apparently, Dana Scott did not publish, to our best knowledge the first mention in print is [57, p. 219] and first generic description of the technique for arbitrary datatypes is [58]).

As before, `Either` can be replaced with either of

$$\forall c.(a \to c) \to (b \to c) \to c$$
$$\lambda c.(a \to c) \to (b \to c) \to c$$

which can be encoded in Haskell as either of

```
newtype EitherS a b = EitherS
  { runEitherS
      :: forall c
       . (a -> c) -> (b -> c) -> c }

left :: a -> EitherS a b
left a = EitherS (\ac bc -> ac a)

right :: b -> EitherS a b
right b = EitherS (\ac bc -> bc b)

newtype EitherS' c a b = EitherS'
```

```
  { runEitherS'
      :: (a -> c) -> (b -> c) -> c }

left' :: a -> EitherS' c a b
left' a = EitherS' (\ac bc -> ac a)

right' :: b -> EitherS' c a b
right' b = EitherS' (\ac bc -> bc b)
```

with `runEitherS` (`runEitherS'`) taking the role of an eliminator (`case` operator) and `left` and `right` (`left'` and `right'`) taking the roles of `Left` and `Right` constructors respectively.

Similarly, `(a, b)` can then be generalized to either of

$$\forall c.(a \to b \to c) \to c$$

$$\lambda c.(a \to b \to c) \to c$$

and encoded in Haskell as either of

```
newtype PairS a b = PairS
  { runPairS
      :: forall c
       . (a -> b -> c) -> c }

pair :: a -> b -> PairS a b
pair a b = PairS (\f -> f a b)

newtype PairS' c a b = PairS'
  { runPairS'
      :: (a -> b -> c) -> c }

pair' :: a -> b -> PairS' c a b
pair' a b = PairS' (\f -> f a b)
```

Substituting all `Left`s with `left`, `Right`s with `right`, `case`s on `Either`s with `runEitherS`, pair constructions with `pair`, and `case`s on pairs with `runPairS` (and similarly for primed versions) does not change computational semantics of the transformed program in the sense that Scott-transformation of the original program's normal form coincides with the normal form of the Scott-transformed program.

Replacing a single datatype in a program with its Scott-encoding can be viewed as a kind of selective CPS-transform on those subterms of the program that use the datatype. The type of transformed functions changes the same way in both transformations, but Scott-encoding groups all continuation arguments, hides them behind a type alias and introduces a bunch of redundant beta reductions in constructors and eliminators.

The upside of CPS-transforming with Scott-encoding is that it supports partial applications, requires absolutely no thought to perform and no substantial changes to the bodies of the functions that are being transformed. It is also very useful for designing new languages and emulating datatypes in languages that do not support them[12] as it allows to use datatypes when none are supported by the core language.

The most immediate downside of this transformation is very poor performance on modern conventional CPUs. For instance, pattern matching on `Either` produces a simple short conditional `jmp` while for `runEitherS` the compiler, in general, cannot be sure about value of the arguments (it can be anything of the required type, not only `left` or `right`) and has to produce an indirect `jmp` (or `call` if

[12] For example, most instances of the *visitor* object-oriented (OOP) design pattern that are not simply emulating `Functor` instances usually emulate pattern matching with Scott-encoding.

it is not a tail call) and both `left` and `right` require another indirect `jmp`. This wastes address cache of CPU's branch predictor and confuses it[13] when instruction pointer jumps out of the stack frame.

For some classes of programs, however, it can increase performance significantly. For instance, in a "*case-tower*" like

```
doSomethingOn s = case internally s of
  Right a -> returnResult a
  Left b -> handeError b

internally s =
  case evenMoreInternally s of
    Right (a,s) -> doSomethingElse a s
    Left b -> Left b

doSomethingElse a s =
  case evenMoreInternally s of
    Right (a,s) -> Right a
    Left b -> Left b
```

(which is commonly produced by parser combinators) performing this selective CPS-transform followed by inlining and partial evaluation of the affected functions will replace all construction sites of `Left`s with direct calls to `handeError`, and `Right`s in `doSomethingElse` (and, possibly, the ones residing in `evenMoreInternally`) with `returnResult`.

In other words, rewriting this type of code using Scott-encoded datatypes is a way to apply deforestation [59] to it, but semi-manually as opposed to automatically, and with high degree of control. This fact gets used a lot in Hackage libraries, where, for example, most parser combinators (section 5.2) use Scott-encoded forms internally.

5.1.3 Cont

One of the roundabout ways to express pure values in Haskell is to wrap them with the `Identity Functor` (section 4.1.2) for which `Identity a`, logically, is just a pure type variable a. Applying generalized Kolmogorov's translation to this variable gives either of

$$\forall c.(a \to c) \to c$$

$$\lambda c.(a \to c) \to c$$

In Haskell the latter type is called `Cont`. It is defined in `Control.Monad.Cont` of `mtl` [52] as

```
newtype Cont r a = Cont
  { runCont :: (a -> r) -> r }
```

with the following `Monad` instance

```
instance Pointed (Cont r) where
  pure a = Cont $ \c -> c a

instance Monad (Cont r) where
  m >>= f  = Cont $ \c -> runCont m
                  $ \a -> runCont (f a) c
```

`Cont` has a transformer version defined in `Control.Monad.Trans.Cont` module of `transformers` [25] package as follows

[13] Note that this does not happen for the full CPS-transform of the previous subsection since that translation does no calls.

```haskell
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }

instance MonadTrans (ContT' r) where
  lift m = ContT (m >>=)
```

Interestingly, however, unlike `Identity` and `IdentityT` which have different `Monad` instances (see section 4.4.1), `Cont` and `ContT` have identical ones (equivalent to the one given above). Of particular note is the fact that the definition of `(>>=)` for `ContT` does not refer to the `Monad` operators of its argument `m`. This means that in cases when we do not need the `MonadTrans` instance (for which we have to have a `newtype` wrapper) we can redefine `ContT` as simply

```haskell
type ContT r m a = Cont (m r) a
```

The latter fact means that `ContT`, unlike other `MonadTrans`formers we saw before, is not a "`Monad` transformer" as it is not a functor on category of monads (it is always a `Monad` irrespective of the argument `m`). This property can be explained by the fact that, as we noted at the top of this section, `Cont Monad` is a kind of *"implicit-CPS"* form of computations. Since all it does is chain return addresses it does not care about types of computations those addresses point to.

5.1.4 Delimited callCC

Peirce's law states that

$$((a \to b) \to a) \to a$$

by applying generalized Kolmogorov's translation we get

$$\begin{gathered}\neg\neg(((a \to b) \to a) \to a)\\ \neg(\neg a \to \neg((a \to b) \to a))\\ \neg\neg((a \to b) \to a) \to \neg\neg a\\ (\neg\neg(a \to b) \to \neg\neg a) \to \neg\neg a\\ ((\neg\neg a \to \neg\neg b) \to \neg\neg a) \to \neg\neg a\end{gathered}$$

which can be encoded in Haskell as (note that this time we use $\forall$ variant of the translation)

```haskell
peirceCC :: ((Cont r a -> Cont r b) -> Cont r a)
         -> Cont r a
peirceCC f = Cont $ \c ->
  runCont (f (\ac -> Cont $ \_ -> runCont ac c)) c
```

This operator takes a function `f`, applies some magical subterm to it and then gives it its own return address. That is, in case the function `f` ignores its argument `peirceCC` is completely transparent. The magical argument `peirceCC` applies to `f` is itself a function that takes a computation producing value of the same type `f` returns as a result. The subterm then computes the value of the argument but ignores its own return address and continues to the return address given to `peirceCC` instead. In other words, `peirceCC` applies `f` with an *escape continuation* which works exactly like a `return` statement of conventional imperative languages (as opposed to `Monad`'s `pure` which should not be called "`return`", see section 4.2.1).

Note that when `ac` argument to the magical subterm is pretty boring: it is a computation that gets computed immediately. Hence, unless we require every subterm of our program to be written in *implicit-CPS* form we can simplify `peirceCC` a bit as follows

```haskell
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a
callCC f = Cont $ \c ->
  runCont (f (\a -> Cont $ \_ -> c a)) c
```

This operator bears a name of "delimited `call/cc` (`callCC`)" [27] and the escape continuation it supplies to `f` not only works but also looks exactly like an imperative `return` (in that it takes a pure value instead of a computation producing it).

5.1.5 Scheme's call/cc and ML's callcc

Note that delimited `callCC` is semantically different from similarly named operators of SML [60] and Scheme [26]. SML defines its operator as

```
type 'a cont
val callcc : ('a cont -> 'a) -> 'a
```

where `'a cont` type is the type of the *current global continuation* which is the computation till the end of the whole program, this type is a kind of technical alias for what, logically, should be $a \to b$, i.e. `callcc`'s type, logically, is non-Kolmogorov-translated Peirce's law.

The difference is that by applying Kolmogorov's translation to Peirce's law `callCC` gains intuitionistic witnesses (and, hence, purely functional implementations) and becomes *delimited* by the current `Cont` context instead of the whole program. Meanwhile, implementations of non-delimited `callcc` and `call/cc` require special support from the compiler/interpreter and Kiselyov [28] eloquently advocates that they simply should not exist as they are *less* useful than their delimited versions and their implementations introduce nontrivial trade-offs.

5.2 Monadic Parser Combinators

`Monad`ic parser combinators are not by themselves an error handling mechanism, but they have to handle failed parsing attempts and such computations can be seen as a kind of error handling.

Parser combinators can possess a wide variety of semantics and implementations, to mention just a few possible dimensions of the space:

- they can either automatically backtrack on errors or keep the state as is,
- they can distinguish not only successful and failed parsing attempts but also attempts that consumed none of the input and those that consumed at least one element of the input [29],
- they can support an impure state (e.g., make it a `Monad`),
- track position in the input stream,
- allow programmer-provided types in errors,
- provide `MonadTrans`former versions,
- encode their internals with Scott-encoding (section 5.1.2) for efficiency.

Discussing most of those features and their combinations is beyond the scope of this article. In the following subsections we shall only mention "backtrack vs. not" problem, in section 15 we shall also apply Scott-encoding to an almost identical structure. Detailed implementations of other features can be studied by following respective references.

The most popular parser combinator libraries for Haskell are Parsec [61], Attoparsec [62], and Megaparsec [63].

5.2.1 Simple stateful parser combinator

The simplest `Monad`ic parser combinator is just a composition of `StateT` (section 4.4.4) and `ExceptT` (section 4.4.3) `MonadTrans`formers with inner `Identity` (section 4.1.2)

```
type SParser s e = StateT s (ExceptT e Identity)
```

which can be β-reduced into

```
newtype SParser s e a = SParser
  { runSParser :: s -> Either e (a, s) }
```

with the following Monad instance

```
instance Pointed (SParser s e) where
  pure a = SParser $ \s -> Right (a, s)

instance Monad (SParser s e) where
  p >>= f = SParser $ \s ->
    case runSParser p s of
      Left x -> Left x
      Right (a, s') -> runSParser (f a) s'
```

5.2.2 ... with full access to the state

While the definition above is, in fact, exactly the definition used in `ponder` [64] parser combinator library, it provides no way to access the state of the parser on error, which makes it very inconvenient in practice. However, a simple modification of the type that moves Either into the tuple

```
newtype Parser s e a = Parser
  { runParser :: s -> (Either e a, s) }
```

which, of course, in isomorphic to[14]

```
newtype Parser s e a = Parser
  { runParser :: s -> Either (e, s) (a, s) }
```

solves this problem of access to state while keeping the definition of Monad identical to the above.

Theorem 1. *Parser satisfies Monad laws.*

Proof. By case analysis. Also see the next proof. □

The *choice* operator can be implemented by one of the two possible instances of Alternative. The first one rolls-back the state on error

```
instance Monoid e => Alternative (Parser s e) where
  empty = Parser $ \s -> Left (mempty, s)
  f <|> g = Parser $ \s -> case runParser f s of
    Right x -> Right x
    Left  (e, _) -> case runParser g s of
      Right x -> Right x
      Left  (f, _) -> Left (e `mappend` f, s)
```

while the second does not

```
instance Monoid e => Alternative (Parser s e) where
  empty = Parser $ \s -> Left (mempty, s)
  f <|> g = Parser $ \s -> case runParser f s of
    Right x -> Right x
    Left  (e, s') -> case runParser g s' of
      Right x -> Right x
      Left  (f, s'') -> Left (e `mappend` f, s'')
```

Theorem 2. *Both versions satisfy the laws of Alternative.*[15]

[14] The corresponding MonadTransformer stack is better left unspoken.

[15] Note, however, that SParser from section 5.2.1 can only do backtracking because, unlike Parser, it is asymmetric in its use of Either.

Proof. By case analysis.

Note that to convince yourself that `(<|>)` is associative it is enough to observe that in `a <|> b <|> c` for the above definitions

- `Right` is a zero,
- values of `e` always propagate to the right,
- while `s` is stays constant in the roll-back version, or always propagates in the no-roll-back version, but never both.

Which means that parentheses can't influence anything in either case.

The same idea can be used in similar proofs involving similar operators of `State` and `Parser`.

□

From the commonly used Haskell parser combinator libraries Attoparsec rolls-back while Parsec and Megaparsec do not, instead they implement backtracking with a separate combinator for which we could give the following type signature

```
try :: Parser s e a -> Parser s e a
```

5.2.3 Examples

The already given definitions allow us enough headroom to define some primitive parsers and a couple of examples. For instance, assuming `Alternative` rolls-back we can write

```
type Failures = [String]

eof :: Parser String Failures ()
eof = Parser $ \s -> case s of
  [] -> Right ((), s)
  _  -> Left  (["expected eof"], s)

char :: Char -> Parser String Failures ()
char x = Parser $ \s -> case s of
  []     -> Left  (["unexpected eof"], s)
  (c:cs) -> if (c == x)
    then Right ((), cs)
    else Left (["expected `" ++ [x] ++ "' got `" ++ [c] ++ "'"], s)

string :: String -> Parser String Failures ()
string [] = pure ()
string (c:cs) = char c >> string cs

parseTest = runParser (string "foo") "foo bar"
            == Right((), " bar")
         && runParser (string "abb" <|> string "abc") "aba"
            == Left (["expected `b' got `a'"
                     ,"expected `c' got `a'"], "aba")
```

To use the other implementation of `Alternative` we would need to wrap all calls to `string` on the left hand sides of `(<|>)` with `try`s.

Semantics-wise our `Parser` combines features of Attoparsec (backtracking) and Megaparsec (custom error types). Of course, it fits on a single page only because it has a minuscule number of features in comparison to either of the two. To make it practical we would need, at the very least, to implement tracking of the position in the input stream and a bunch of primitive parsers, which we leave as an exercise to the interested reader.

Interestingly, this exact implementation of handling of errors by accumulation via `Alternative` over a `Monoid` seems to be novel (although, pretty trivial). Megaparsec, however, does something very similar by accumulating errors in `Set`s instead of `Monoid`s.

`MonadTrans`former versions of these structures can be trivially obtained by adding `Monad`ic index `m` after the arrow in definition of `Parser` (i.e. by exposing the internal `Monad` of the original `MonadTrans` stack) and correspondingly tweaking base-level definitions and all type signatures.

5.3 Other variants of MonadCatch

Finally, worth mentioning are two lesser-known variants of structures similar to structures of section 4.8. The first one is defined in `Control.Monad.Exception.Catch` module of `control-monad-exception` [65] package as

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

and the second one in `Control.Monad.Catch.Class` module of `catch-fd` [66] package

```
class Monad m => MonadThrow e m | m -> e where
  throw :: e -> m a

class (MonadThrow e m, Monad n) => MonadCatch e m n | n e -> m where
  catch :: m a -> (e -> n a) -> n a
```

Note that `control-monad-exception` does not define a type class with a `throw` operator, that library provides a universal computation type `EM` (similar to `EIO` of section 15) with such an operator instead. Also note that the common point of those two definitions is that both `catch` operators change the type of computations from `m` to `n`.

6 The nature of an error

Lets forget for a minute about every concrete algebraic error-handling structure mentioned before and try to invent our own algebra of computations by reasoning like a purely pragmatic programmer who likes to make everything typed as precisely as possible.

We start, of course, by pragmatically naming our type of computations to be `C`. Then, we reason, it should be indexed by both the type of the result, which we shall pragmatically call `a`, and the type of exceptions `e`. We are not sure about the body of that definition, so we just leave it undefined

```
data C e a
```

Now, we know that `Monad`s usually work pretty well for the computation part (since we can as well just lift everything into `IO` which is a `Monad`), so we write

```
return :: a -> C e a

(>>=) :: C e a -> (a -> C e b) -> C e b
```

and expect these operators to satisfy `Monad` laws (section 4.2.1).

Meanwhile, pragmatically, an ”exceptional” execution path requires two conventional operators:

- a method of raising an exception; the type of this operator seems to be pretty straightforward

  ```
  throw :: e -> C e a
  ```

  as it simply injects the error into `C`,

- and a method to catch exceptions; the overly-general type for this operator is, again, pretty straightforward

  ```
  catch :: C e a -> (e -> C f b) -> C g c
  ```

  The only obvious requirement here is that the type the "handler" function (the second argument of `catch`) can handle should coincide with the type of errors the "computation" (the first argument) can `throw`.

Finally, we pragmatically expect all of those operators to obey conventional error handling rules, giving us the following definition.

Definition 2. ***Pragmatic error handling structure.*** *Structure `m :: * => * => *` with `return`, `(>>=)`, `throw`, and `catch` operators satisfying*

1. *`return` and `(>>=)` obey `Monad` laws (section 4.2.1),*
2. *`throw e >>= f == throw e` ("throwing of an error stops the computation"),*
3. *`throw e `catch` f == f e` ("throwing of an error invokes the most recent error handler"),*[16]
4. *`return a `catch` f == return a` ("`return` is not an error").*

7 The type of error handling operator

The first question to the structure of `C` is, of course, what is the precise type of `catch` operator.

```
catch :: C e a -> (e -> C f b) -> C g c
```

In other words, we would like to know which of the variables `f`, `g`, `b`, and `c` in this signature should have their own universal quantifier and which should be substituted with others. The answer comes by considering several cases.

- Firstly, let us consider the following expression.

  ```
  return a `catch` f
  ```

  The expected semantics of `catch` requires (by item 4 of definition 2)

  ```
  return a `catch` f == return a
  ```

  Note that the most general type for `return a` expression is `forall e . C e a` for `a : a`[17]. Moreover, we can assign the same type to any expression that does not `throw` since

  - both `a` and `e` in the type signify the potential to `return` and `throw` values of the corresponding types,
  - and an expression that does not `throw` any errors can be said to not-`throw` an error of any particular type, similarly to how bottom elimination rule works. Or, equivalently, any such computation can be said to `throw` values of an empty type and an empty type can always be replaced with any other type by bottom elimination.[18]

[16] Similarly to GHC's imprecise exceptions of section 4.5 dynamic dispatch can be implemented on top of such a structure. We shall do this in section 12.2.

[17] The reader might have noticed already that we abuse notation somewhat by assuming type variables and term variables use distinct namespaces. This expression happens to be the first and the only one that uses both at the same time, hence it looks like an exiting "type-in-type" kind of thing, but it is not, it is ordinarily boring.

[18] Implicitly or with `f `catch` bot-elim` which is extentionally equivalent to `f`.

- Now let us consider the following expression, assuming `e` and `f` are of different types (i.e. both the computation and the handler throw different exceptions).

  ```
  throw e `catch` (\_ -> throw f)
  ```

  The expected semantics of `catch` requires (by item 3 of definition 2)

  ```
  throw e `catch` (\_ -> throw f) == throw f
  ```

These two cases show that `g` should be substituted with `f` and `e` should be kept separate from `f` because

- if computation `throw`s then the type `f` in the handler "wins",
- but if it does not `throw` then `e` is an empty type and it can be substituted for any other type, including `f` (similarly to the type of `return` above)[19]
- these two cases are mutually exclusive.

That is, the type for `catch` is at most as general as

```
catch :: forall e f . C e a -> (e -> C f b) -> C f c
```

- Continuing, item 4 of definition 2 shows that `c` has to coincide with `a`.
- Similarly, item 3 requires

  ```
  throw e `catch` (\_ -> return a) == return a
  ```

  which shows that `c` has to coincide with `b`.

All these observations combine into the following.[20]

Theorem 3. *For any type `C :: * => * => *` obeying definition 2 the most general type for the `catch` operator is*

```
catch :: forall a e f . C e a -> (e -> C f a) -> C f a
```

Proof. By the above reasoning. That is, by simple unification of types of `return`, `throw`, `(>>=)` operators of definition 2 and the following equations that are consequences of rules of definition 2

```
return a `catch` f == return a
throw e `catch` (\_ -> return a) == return a
throw e `catch` (\_ -> throw f) == throw f
```

□

[19] The only nontrivial observation in this section.

[20] Spoilers! The reader is only supposed to notice the following after reading section 10.

Note that we could have written an equivalent up to names of operators sections 6 and 7 that explained why the type of `(>>=)` is the correct type for computations in `C` given that error handling should be done `Monad`ically. Which is another reason why we disagree with the conventional wisdom in footnote 2.

8 Conjoinedly Monadic algebra

After theorem 3 it becomes hard to ignore the fact that `throw` has the type of `return` and `catch` has the type of `(>>=)` in the "wrong" index for `C`. Moreover, item 3 of definition 2 looks exactly like a left identity law for `Monad` (section 4.2.1). While it is not as immediately clear that `catch` should be associative, it seems only natural to ask whenever the following conjoinedly `Monad`ic restriction of definition 2 has any instances.

Definition 3. ***Conjoinedly monadic error algebra.*** *A type* `m :: * => * => *` *for which*

- `m` *is a* `Monad` *in its second index (that is,* `m e` *is a* `Monad` *for all* `e`*)*
- `m` *is a* `Monad` *in its first index (that is,* `\e . m e a` *is a* `Monad` *for all* `a`*)*

and assuming

- *the names of* `Monad` *operators in the second index of* `m` *are* `return` *and* `(>>=)`
- *the names of* `Monad` *operators in the first index are* `throw` *and* `catch`

the following equations hold

1. `return x `catch` f == return x`,
2. `throw e >>= f == throw e`.

If we replace `Monad` in definition 3 with `MonadFish` (section 4.2.2), as usual, the latter two equations become a bit clearer.

Definition 4. ***Fishy conjoinedly monadic error algebra.*** *A type* `m :: * => * => *` *for which*

- `m` *is a* `MonadFish` *in its second index*
- `m` *is a* `MonadFish` *in its first index*

and assuming

- *the names of* `MonadFish` *operators in the second index are* `return` *and* `(>=>)`
- *the names of* `MonadFish` *operators in the first index are* `throw` *and* `handle`

the following equations hold

1. `return `handle` f == return`,
2. `throw >=> f == throw`.

On other words, definitions 3 and 4 define a structure that is a `Monad` (`MonadFish`) twice and for which `return` is a left zero for `catch` (`handle`) and `throw` is a left zero for `(>>=)` (`(>=>)`).

9 Instances: Either

Pragmatic programmer finally loses last bits of concentration realizing that `Either` type seems to match requirements of definition 3 and goes into sources to check whenever Haskell's standard library already has such a `catch`. Unfortunately, `Data.Either` module does not define such an operator. However, `catchE` and `throwE` of `ExceptT` (section 4.4.3) match. Of course, if we substitute `Identity` for `m`, `ExceptT` turns into `Either` and those operators can be simplified to

```
throwE' :: e -> Either e a
throwE' = Left

catchE' :: Either e a
        -> (e -> Either f a)
        -> Either f a
catchE' (Left e)  h = h e
catchE' (Right a) _ = Right a
```

Lemma 3. *For a given `Monad` `m` and a fixed argument `a`, `ExceptT` with `throwE` as `return` and `catchE` as `(>>=)` is a `Monad` in argument `e`.*

Proof. Any of the following

- **By brute force:** by case analysis, using the fact that `m` satisfies `Monad` laws.
- **Another way:** trivial consequence of section 10.

□

Lemma 4. *For `ExceptT` with the above operators the following equations hold*

1. `return x `catchE` f == return x`,
2. `throwE e >>= f == throwE e`.

Proof. By trivial case analysis.

□

Theorem 4. *`ExceptT` and, by consequence, `Either` satisfy definition 3.*

Proof. Consequence of lemma 3 and lemma 4.

□

10 Logical perspective

Note, that from a logical perspective most of the above is simply trivial. `Either a b` is just $a \vee b$ and so if $\lambda b.a \vee b$ is a `Monad` then $\lambda a.a \vee b$ must be a `Monad` too since $\vee$ operator is symmetric. Sections 6-8 simply generalize this fact with interactions between `Left`, `Right` and two `(>>=)` operators into definition 3.[20]

The main point of this article is that **this generalization is itself interesting** — the fact that we shall demonstrate in the following sections.

11 Encodings

Despite the noted triviality, these facts do not seem to be appreciated by the wider Haskell community. In particular:

- `ExceptT` does not get much use in Hackage packages in general,

- the equivalent of `catchE` for `ErrorT` has an overly-restricted type

```
catchError :: (Monad m)
           => ErrorT e m a
           -> (e -> ErrorT e m a)
           -> ErrorT e m a
m `catchError` h = ErrorT $ do
    a <- runErrorT m
    case a of
        Left  l -> runErrorT (h l)
        Right r -> return (Right r)
```

- no Monadic parsing combinator library from Hackage (most obvious beneficiaries of the observation) defines the would-be-Monad instance of `throwE` and `catchE`.

To our best knowledge, the only Hackage package that is explicitly aware of the fact that `Either` is a `Monad` twice is `errors` [67] and the only packages that seem to be aware that `throw` and `catch` in general need more general types than those given by `MonadCatch` of section 4.8 are those discussed in section 5.3 (but they miss the fact that their `catch` operators want to be Monadic `binds`). To our best knowledge, no Hackage package utilizes both facts. As to the question why had not anybody notice and start exploiting this yet we hypothesize that the answer is because Haskell cannot express these properties conveniently (not to mention less expressive mainstream languages which cannot express them at all).

The simplest possible encoding of definition 3 in Haskell is just

```
class ConjoinedMonads m where
  return :: a -> m e a
  (>>=)  :: m e a -> (a -> m e b) -> m e b

  throw  :: e -> m e a
  catch  :: m e a -> (e -> m f a) -> m f a
```

but it does not play too well with the rest of the Haskell ecosystem. In the ideal world, definition 3 would get encoded with the following pseudo-Haskell definition

Definition 5. ***Proper pseudo-Haskell definition.***

```
class (forall a . Monad (\e -> m e a)) -- `Monad` in `e`
     , forall e . Monad (m e) -- `Monad` in `a`
    => ConjoinedMonads m where
    -- and that's it
```

however, Haskell allows neither rank 2 types in type classes, nor lambdas in types, which brings us to the following "theorem".

"Theorem" 5. *Haskell cannot properly (equivalently to definition 5) define `ConjoinedMonads`.*

Proof. Proper definition of `ConjoinedMonads` requires rank 2 types in type class declaration, which is not possible in modern Haskell. There is no way to emulate rank 2 definition using only rank 1 constructions.

□

We call it a "theorem" because we do not really know if its proof really works out for Haskell as Haskell has an awful lot of language extensions (including future ones) and there might be some nontrivial combination of those that gives the desired effect. In particular, GHC version 8.6 released just before this article was finished introduced `QuantifiedConstraints` extension [68] allowing us to write

```haskell
data Swap r a e = Swap { unSwap :: r e a }

instance (forall e . Monad (r e)
        , forall a . Monad (Swap r a))
      => ConjoinedMonads r where
  -- ...
```

which, arguably, can be considered good enough, though not very convenient in practice.

The purposes of this article, however, is not to demonstrate that there is a convenient form of definition 3 in Haskell but to show what could be achieved if there were such a convenient definition. Which means that we can and, hence, shall completely ignore the question of the most elegant Haskell representation for definition 3 and just use the very first definition of `ConjoinedMonads` from above for simplicity.

As to the naming, it is, indeed, tempting to call this structure `BiMonad`, but that name is already taken by another structure from category theory. Then, since the structure consists of two `Monads` that are "dual" to each other via interaction laws it is tempting to call it `DualMonad` as a double-pun, but that "duality" is different from the usual duality of category theory. Which is why we opted into using the name "`ConjoinedMonads`" (in the sense of "conjoined twins", conjoined with left-zeroes).

12 Instances: constant Functors

In this section we discuss the relationship between `ConjoinedMonads` (and definition 3) and `MonadThrow`, `MonadCatch`, and `MonadError` from section 4.8.

12.1 MonadError

`MonadError` (section 4.8.1) relationship to `ConjoinedMonads` turns out to be pretty simple. Remember that `MonadError` is defined using functional dependencies

```haskell
class (Monad m) => MonadError e m
                 | m -> e where
```

This means that Haskell type system guarantees that for each `m` there exist unique `e` if `MonadError e m` is inhabited. This, in turn, means that substituting a constant `Functor r = \x a -> m a` over `Monad m` into the definition of `ConjoinedMonads` produces

```haskell
class ConjoinedMonads (\x a -> m a) where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b

  throw  :: e -> m a
  catch  :: m a -> (e -> m a) -> m a
```

The first two operators are just the definition of `Monad m`, the latter two match `MonadError`'s `throwError` and `catchError` exactly.

Theorem 6. *`MonadError` is a `ConjoinedMonads` that is constant in its first index.*

Proof. By the above argument.

□

12.2 MonadThrow and MonadCatch

For MonadThrow and MonadCatch (section 4.8.2) it is not the case that e is unique, since Exception e is a whole class of types. Moreover, operator catchM of MonadCatch, unlike catchError of MonadError, does dynamic dispatch by casting Exceptions to the type of its handler's argument and propagating errors when the cast fails. Note that, strictly speaking, purely from type perspective MonadCatch is not *required* but *allowed* to cast, but all the instances do actually cast. The latter fact means that we can distill that common computational pattern by redefining those structures using the technique used by imprecise exceptions of section 4.5 as follows

```
class Monad m => MonadThrowS m where
  throwS :: SomeException -> m a

class MonadThrow m => MonadCatchS m where
  catchS :: m a
         -> (SomeException -> m a) -> m a

throwM' :: (MonadThrowS m, Exception e)
        => e -> m a
throwM' = throwS . toException

handleOrAgain h e = case fromException e of
  Just f -> h f
  Nothing -> throwM e

catchM' :: (MonadCatchS m, Exception e)
        => m a -> (e -> m a) -> m a
catchM' ma = catchS ma . handleOrAgain
```

Note that MonadCatchS is, again, a constant ConjoinedMonads with error index fixed to SomeException. Also note that throwM' above is the only way to get an equivalent for throwM because toException is the only way to cast an arbitrary type to SomeException. On the other hand, catchM from MonadCatch, unlike catchM' above, allows for instances that can cheat. For example, catchM can give a constant SomeException to the handler every time instead of casting anything. We feel that this implies that MonadCatch is not a proper formal structure for error handling.

Definition 6. ***Proper MonadCatch instance.*** *We shall call an instance of MonadCatch* proper *when its catchM can be decomposed into catchS and handleOrAgain.*

Theorem 7. *Every proper instance of MonadCatch is a composition of ConjoinedMonads that is constant in its error index with toException in throwD and handleOrAgain in catchD. In particular, MonadThrow is a composition of Pointed in the error index with toException.*

Proof. By the above reasoning.

□

13 Instances: parser combinators

In this section we discuss the application of ConjoinedMonads and definition 3 to Monadic parser combinators discussed in section 5.2.

13.1 Inevitable definitions

To start off, let us continue using the definition of Parser type from section 5.2.2. The Monad instance in index e for this type is similarly easy to implement (by just trying all free functions of appropriate types) and it, too, has two possible implementations

```
throwP :: e -> Parser s e a
throwP e = Parser $ \s -> Left (e, s)

catchP :: Parser s e a -> (e -> Parser s f a) -> Parser s f a
catchP p f = Parser $ \s ->
  case runParser p s of
    Right x -> Right x
    Left (e, _) -> runParser (f e) s

catchP' :: Parser s e a -> (e -> Parser s f a) -> Parser s f a
catchP' p f = Parser $ \s ->
  case runParser p s of
    Right x -> Right x
    Left (e, s') -> runParser (f e) s'
```

with `catchP` doing backtracking on failures and `catchP'` proceeding to handling with the current state.

Theorem 8. *`Parser` is a `ConjoinedMonads` for both versions of `catchP`.*

Proof. `Monad` laws for `catchP'` follow from the corresponding laws for `(>>=)` of section 5.2.2.

The rest can be proven by trivial case analysis and/or by using the observation from the proof of theorem 2.

□

A curious consequence of the above theorem is that `(>>=)` of section 5.2.2 also has a roll-back version which satisfies `Monad` laws

```
bindP p f = Parser $ \s ->
  case runParser p s of
    Left x -> Left x
    Right (a, _) -> runParser (f a) s
```

Though, of course, a `Parser` that would use `bindP` in place of the usual `(>>=)` could not be called "parser" anymore.

13.2 The interesting parts

The first interesting fact is that `(<|>)` operator of the `Alternative` (section 4.1.3) type class is simply a type restricted version of `orElseP` which, in turn, is just `(>>)` operator for the `Monad` in index `e`

```
orElseP :: Parser s e a -> Parser s f a -> Parser s f a
orElseP f g = f `catchP` const g

instance Monoid e => Alternative (Parser s e) where
  empty = Parser $ \s -> Left (mempty, s)
  f <|> g = f `orElseP` g
```

Of even more interest is the fact that substituting `orElseP` instead of `(<|>)` into the definition of `many` operator produces `many` and `some` operators with types that show that `some` inherits error produced by its argument while `many` ignores them

```
someP :: Parser s e a -> Parser s e [a]
someP v = fmap (:) v <*> manyP v

manyP :: Parser s e a -> Parser s f [a]
manyP v = someP v `orElseP` pure []
```

This method of substituting `(<|>)` with `orElseP` extends to other similar combinators like `choice`, `optional`, `notFollowedBy` of all three aforementioned parser combinator libraries (Parser, Attoparsec, Megaparsec) and similar structures. The overall effect of this substitution is very useful in practice: it produces generic parser combinators that can be used to express parsers that are precise about errors they raise and handle. We can not emphasize this fact enough.

All of the above results of this section trivially generalize to their `MonadTrans` versions as usual.

14 Instances: conventional throw and catch via callCC

It is well-known fact that Emacs LISP-style `throw` and `catch` can be emulated with Scheme's `call/cc` and some mutable variables [69, 70]. Neil Mitchel used pretty much the same technique and Haskell's `callCC` in for Shake build system [71, 72] (however, at the time of writing Shake no longer uses that code). In this section we shall demonstrate that a structure with the same semantics can be implemented in pure Haskell without the use of mutable variables. In all the cases, as usual, C++/Java-style dynamic dispatch can be added on top using the same `cast`ing technique of sections 4.5 and 12.2. Hence without the loss of generality in this section we shall discuss only the most-recent-handler case.

14.1 Second-rank callCC

Remember the definition of `callCC` from section 5.1.4. The underappreciated fact about that function is that its type is not its most general type for its term. Note that variable b in Peirce's law

$$((a \to b) \to a) \to a$$

plays the same role as `r` plays in the definition of `Cont`: it is a generalization of the bottom $\bot$ constant. This, of course, means that we can generalize Peirce's law to

$$((\forall b.a \to b) \to a) \to a$$

and, by repeating the derivation in section 5.1.4, give the following second-rank type for `callCC`

```
callCCR2 :: ((forall b . a -> Cont r b) -> Cont r a) -> Cont r a
```

while keeping exactly the same implementation.

14.2 ThrowT MonadTransformer

Note that, in essence, `catch` maintains a stack of handler addresses and `throw` simply `jmp`s to the most recent one. Emulation of exceptions with `call/cc` works similarly [69, 70]. The main never explicitly stated observation in that translation is that the type of the handler in the type of

```
catch :: M -> (e -> M) -> M
```

matches the type of `throw :: e -> M` and the type of escape continuation when `M` is `ContT r m b`. In other words, we can simply assign

```
type Handler r e m = forall b . e -> ContT r m b
```

to be to type of our handler and since `callCC` provides an escape continuation directly to its argument `catch` can simply save it and `throw` can simply take the most recent one and escape into it

```
throwT :: e -> ThrowT r m e a
throwT e = ThrowT $ \currentThrow -> currentThrow e
```

Also note that since the stack `catch` maintains stays immutable between `catch`es and each state of the stack is bound to the computation argument of `catch`, in principle, we should be able to use a simple context (pure function, `Reader`) instead of a mutable variable as follows

```
type ThrowT r m e a =
  ReaderT (Handler r e m) -- for saving last handler
          (ContT r m)     -- for callCC
          a
```

which, after inlining all the definitions except pure `Cont` becomes

```
newtype ThrowT r m e a = ThrowT
  { runThrowT :: (forall b . e -> Cont (m r) b)
              -> Cont (m r) a }
```

Finally, since the escape continuation of delimited `callCC` escapes to the same address where the body of `callCC` normally returns, to emulate a single `catch` we need to chain two `callCC`s as follows

```
catchT :: ThrowT r m e a
       -> (e -> ThrowT r m f a)
       -> ThrowT r m f a
catchT m h = ThrowT $ \outerThrow ->
  callCC $ \normalExit -> do
    e <- callCCR2 $ \newThrow -> runThrowT m newThrow >>= normalExit
    -- newThrow escapes here
    runThrowT (h e) outerThrow
  -- normalExit escapes here
```

Note that this expression requires our second-rank `callCCR2` since our `Handler` is universally quantified by the variable `b`. However, if we fix `e` to a constant type then the conventional `callCC` will suffice.

Similarly to other uses of generalized Kolmogorov's translation we, too, can hide `r` parameter behind `forall`

```
newtype ThrowT' m e a = ThrowT'
  { runThrowT' :: forall r
                . (forall b . e -> Cont (m r) b)
               -> Cont (m r) a }

throwT' :: e -> ThrowT' m e a
catchT' :: ThrowT' m e a
        -> (e -> ThrowT' m f a)
        -> ThrowT' m f a
```

without any changes to the bodies of `throw` and `catch`.

Theorem 9. *For `Monad m` and any `r`, `ThrowT r m` and `ThrowT' m` are `ConjoinedMonads`s.*

Proof. For each index.

- In index `a`: `ThrowT` is a special case of `ReaderT` and `Cont` and `m` are `Monad`s.
- In index `e`: by substitution of the above definitions into the `Monad` laws, since the definitions of `throwT` and `throwT'` are, essentially, identity functions.

□

15 Instances: error-explicit IO

As we saw in section 4.5, `IO` is defined as a `State Monad` with some magical primitive operations.[21] Which means there is nothing prevents us from extending that `IO` signature with a type for errors.

```
newtype EIO e a
```

Similarly to parser combinators of section 13 there are several possible implementations of this `EIO` (including, in principle, the ones that do backtracking on errors, though, of course, that would be inconsistent with the semantics of the `RealWorld`). The simplest one matches a definition for non-backtracking parser combinator on `State# RealWorld` from section 5.2.2

```
newtype EIO e a = EIO
  { runEIO :: State# RealWorld
           -> (# Either e a, State# RealWorld #) }

instance Pointed (EIO e) where
  pure a = EIO $ \s -> (# Right a, s #)

instance Monad (EIO e) where
  m >>= f = EIO $ \s -> case runEIO m s of
    (# Left  a, s' #) -> (# Left a, s' #)
    (# Right a, s' #) -> runEIO (f a) s'

-- Note how symmetric this is with Pointed and Monad instances.
throwEIO :: e -> EIO e a
throwEIO e = EIO $ \s -> (# Left e, s #)

catchEIO :: EIO e a -> (e -> EIO f a) -> EIO f a
catchEIO m f = EIO $ \s -> case runEIO m s of
  (# Left  a, s' #) -> runEIO (f a) s'
  (# Right a, s' #) -> (# Right a, s' #)
```

Note that very similar structures were proposed before in [21] and `Control.Monad.Exception.Catch` module of `control-monad-exception` [65] discussed in section 5.3. Also note that the definition of GHC's `IO` before imprecise exceptions were introduced was similar to `EIO` above (but without the parameter `e`) and one of the primary motivations behind introduction of builtin exceptions into GHC mentioned in [40] was to make `IO` more efficient by allowing its `(>>=)` to be implemented without pattern-matching. But there are, of course, other ways to eliminate pattern matching. By moving `Either` in the definition of `EIO` out the parentheses using the technique from section 5.2.2 and then Scott-encoding the resulting type we can make the following definition

```
newtype SEIO e a = SEIO
  { runSEIO :: forall r
             . (e -> State# RealWorld -> r)
            -> (a -> State# RealWorld -> r)
            -> State# RealWorld
            -> r }

instance Pointed (SEIO e) where
  pure a = SEIO $ \err ok s -> ok a s
```

[21] Some of which actually break `Monad` laws, but as mentioned in remark 4 that is out of scope of this discussion.

```
instance Monad (SEIO e) where
  m >>= f = SEIO $ \err ok s -> runSEIO m err (\a -> runSEIO (f a) err ok) s

-- Note the same here.
throwSEIO :: e -> SEIO e a
throwSEIO e = SEIO $ \err ok s -> err e s

catchSEIO :: SEIO e a -> (e -> SEIO f a) -> SEIO f a
catchSEIO m f = SEIO $ \err ok s -> runSEIO m (\e -> runSEIO (f e) err ok) ok s
```

Theorem 10. *Both `EIO` and `SEIO` with the above operations are `ConjoinedMonads`s.*

Proof. Consequence of theorem 8 and the fact that Scott-encoding preserves computational semantics. □

16 Instances: conventional IO

Theorem 11. *`IO` is a composition of `ConjoinedMonads` that is constant in its error index with `toException` in `raiseIO#` and `handleOrAgain` in `catch#`.*

Proof. A consequence of of results of theorems 7 and 10 for `e == SomeException`. □

Note that, according to remark 4, the above works out only because `raiseIO#`/`throwIO`, unlike `raise#`/`throw`, are deterministic (see section 4.5).

Also note that in a dialect of Haskell with separate operators for imprecise exceptions (or without imprecise exceptions altogether) we can completely replace `IO` with `EIO` as defined above. We can not, however, apply that construction to GHC's Haskell dialect since it merges precise and imprecise `catch` (see remark 5).

17 Applicatives

Now let us once more turn our attention to the bodies of definitions 3, 4, and 5 (all of which define the same structure).

```
class (forall a . Monad (\e -> m e a))
     , forall e . Monad (m e)
    => ConjoinedMonads m where
```

Since `ConjoinedMonads` is simply a `Monad` × `Monad` with interaction laws between `pure` and `bind` operators (definition 3) it is natural to ask what would happen if we replace one or both of those `Monad`s with more general structures like `Applicative` and modify the interaction laws accordingly.

The two structures with `Applicative` in index `e` seem to be unusable for the purposes of this article since they lack conventional error handling operators. However, the structure with `Monad` in index `e` and `Applicative` in index `a` looks interesting.

```
class (forall a . Monad (\e -> m e a))
     , forall e . Applicative (m e)
    => MonadXApplicative m where
```

In this structure the `Monad`ic index gives conventional `throw` and `catch` operators, and the `Applicative` index can be treated as expressing generalized function application (see section 4.1.2) for structure `m`.

In other words, when the above structure preserves errors and pure values similarly to definition 3

```
throw e <*> a == throw e
pure a `catch` f == pure a
```

(and obeys the laws of `Applicative` and `Monad` for corresponding operators) then it can be used to express λ-calculus with exceptions by simply injecting all `pure` values and `lift`ing all pure functions into it.

In particular, since `ConjoinedMonads` is a special case of `MonadXApplicative`, all `ConjoinedMonads` instances from the previous sections can be used as a basis for such a formalism.

While it is not immediately clear how to make imprecise exceptions into an instance of `MonadXApplicative` (since they are non-deterministic, hence disobeying the above laws, and `throw` having a wrong type to be the identity element for `catch`, see remark 5), there are some interesting instances of `MonadXApplicative` that are not `ConjoinedMonads`s.

For instance, a folklore example of an `Applicative` that is not a `Monad` is "computations collecting failures in a `Monoid`", which can be defined as follows

```
newtype EA e a = EA { runEA :: Either e a }

instance Pointed (EA e) where
  pure = EA . Right

instance Monoid e => Applicative (EA e) where
  f <*> a = EA $ runEA f <**> runEA a where
    (Right f) <**> (Right a) = Right $ f a
    (Right f) <**> (Left  e) = Left e
    (Left  e) <**> (Right a) = Left e
    (Left e1) <**> (Left e2) = Left $ e1 `mappend` e2
```

Note, however, that this structure is a `Monad` in `e`

```
throwEA :: e -> EA e a
throwEA = EA . Left

catchEA :: EA e a -> (e -> EA f a) -> EA f a
(EA a) `catchEA` f = case a of
  Right a -> pure a
  Left  e -> f e
```

which means it is also an instance of `MonadXApplicative`. If we now remember that

- graded monads [22] also require `e` to be a `Monoid` and
- imprecise exceptions, too, can be though as producing a `Monoid` of possible errors with `catch` (including the implicit `catch` over `main`) "observing" one of its elements,

we come to a conclusion that in a calculus with `IO`-effects separated from non-determinism-effects, imprecise exceptions over non-deterministic `Applicative` computations, indeed, form a `Monad` (with equivalence defined up to raising the same set of exceptions, similarly to section 4 of [40]) over the `Monoid` of imprecise exceptions. That is, those, too, are examples of `MonadXApplicative`.

18 Conclusions and future work

We hope that with this article we pointed and then at least partially plugged an algebraic hole in the programming languages theory by showing that conventional `throw`/`try`/`catch`-exceptions are "conjoined" products of pairs of `Monad`s (or `Monad`s and `Applicative`s). This fact, in our opinion,

makes a lot of conventional programming "click into place" similarly to how plain Monads "click" imperative "semicolons".

Of particular note is the fact that everything in this paper, including EIO of section 15, follows the "marriage" framework of [45] of confining effects to monads, but ignores the question of any additional rules for type indexes in question. In other words, ad-hoc exception encoding constructions like that of error-explicit IO [21] or graded monads [22] are mostly orthogonal to our "conjoined" structures and can be used simultaneously.

Besides practical applications described in the body of the paper and observations already mentioned in section 1 (rereading said section about now is highly recommended) we also want turn your attention to the following observations.

1. Conventional error handling with `throw` and `catch` (but without dynamic dispatch) is "dual" to computation, a fact which, in our opinion, is interesting by itself (see the abstract and footnote 20).

2. Meanwhile, the "without dynamic dispatch" part above, in our opinion, provides an algebraic foundation for the argument against languages with dynamic dispatch of exception handlers (or extensively relying on that feature when programming in languages that have it) done by the primitive `catch`, a point which is commonly discussed in the folklore ("exceptions are evil") and was articulated by Hoare from programmer comprehension standpoint already in 1981 [73]. Not only dynamic dispatch of exceptions is, citing Hoare, "dangerous", but it also prevents programs from directly accessing the inherent Monadic structures discussed in this article.

3. We feel that the usual arguments against using Monads for error handling are moot.

   - The problem of syntactic non-uniformness between pure computations, Applicatives and Monads, in our view, is almost trivial to solve: common primitives like `map`/`mapM` should be expressed in terms of Applicatives (of which pure functions are trivial instance) instead of Monads. For instance, `mapM` for list[22] can be rewritten as

     ```
     mapAp :: Applicative f => (a -> f b) -> [a] -> f [b]
     mapAp f     [] = pure []
     mapAp f (a:as) = fmap (:) (f a) <*> mapAp f as
     ```

     Meanwhile, the uniform syntax for pure functions and Applicatives can be made by adding some more missing instances of the LISP macros into the compiler in question.[23] For instance, quasiquotation [74] is one conventional way do such a translation, Conal Elliot's "Compiling to Categories" [48] provides another categorically cute way to achieve similar results.

   - We feel that the problem of modularity as stated by Brady [32]

     > Unfortunately, useful as monads are, they do not compose very well. Monad transformers can quickly become unwieldy when there are lots of effects to manage, leading to a temptation in larger programs to combine everything into one coarse-grained state and exception monad.

     can be solved by applying graded monads to the Monad part of MonadXApplicative.

   In other words, we think that a programming language that

   - provides a primitive `catch` operator that does no dynamic dispatch,
   - provides quasi-quoting/compiling to categories for Applicatives,
   - distinguishes between IO-effects and non-determinism, and

[22] And, similarly, for Traversable which we shall continue to ignore for the purposes of this article.

[23] From a cynical LISP-evangelist point of view, all of "the progress" of the programming languages in the last 50 years can be summarized as "adopting more and more elements (especially meta-programming) from LISP while trying very hard not to adopt the syntax of LISP". From a less cynical perspective, "the progress", at least in typed languages, consists of well-typing said elements.

- uses a graded `MonadXApplicative` for a base type of computations

would provide all the efficiency of imprecise exceptions, simplicity of `Monad`s (doubled, in some sense, since error handling would stop being special), while having none of the usual arguments against said mechanisms applying to it.

We feel that the following future work directions on the topic would be of particular value:

- implementation of a practical "good-enough" (section 11) library for GHC Haskell, and, eventually, an implementation of a dialect of Haskell with a graded `MonadXApplicative` as a base type of computations,
- research into syntax and semantics of "marriages" between precise and imprecise exceptions in a single language, including, but not limited to, research into simpler semantic models for λ-calculus with Monads [45, 75],
- research into the question of whether multiplying more than two `Monad`s and `Applicative`s with non-trivial interaction laws produces interesting structures.[24]

All the practical results of this article except for `catchT` combinator of section 14 were born in 2014 in a course of a single week from observing the structure of a parser combinator `Monad` indexed by errors and values (and other things beyond the scope of this article, the original structure is also an indexed `State Monad` to allow parsing of arbitrary datatypes, not just streams) a very simplified version of which was presented in sections 5.2 and 13. The article itself was started in 2016 but then was rewritten from scratch four times before finally settling to the current presentation. The `catchT` combinator was discovered while writing section 5.1.

This article would have been impossible without the patience of Sergei Soloviev who read and meticulously commented numerous drafts of the paper, numerous people who encouraged me to write this after I described the general idea to them, and all contributors to Emacs and org-mode without whom neither the planning nor the writing of the actual text would have been manageable. The author is also grateful to Sergey Baranov for helpful discussions on related topics which steered the first half of this paper into its current form.

References

[1] Carsten Dominik et al. *Org mode for Emacs*. URL: https://orgmode.org/ (cit. on p. 4).

[2] Eric Schulte et al. "A Multi-Language Computing Environment for Literate Programming and Reproducible Research". In: *Journal of Statistical Software* 46.3 (Jan. 2012), pp. 1–24 (cit. on p. 4).

[3] *GHC: The Glasgow Haskell Compiler*. URL: https://www.haskell.org/ghc/ (cit. on p. 4).

[4] Simon Marlow, ed. *Haskell 2010. Language Report*. URL: https://haskell.org/definition/haskell2010.pdf (cit. on p. 4).

[5] Stephen Diehl. *What I Wish I Knew When Learning Haskell*. May 2016. URL: http://dev.stephendiehl.com/hask/ (cit. on p. 4).

[6] *Hackage: The base package, version 4.9.0.0*. 2016. URL: https://hackage.haskell.org/package/base-4.9.0.0 (cit. on pp. 4, 8–11, 13–15, 21, 22).

[7] Nick Benton and Andrew Kennedy. "Exceptional Syntax". In: *Journal of Functional Programming* 11 (July 2001), pp. 395–410 (cit. on p. 5).

[8] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. "HOPE: An experimental applicative language". In: *ACM Symposium on Lisp and Functional Programming (LFP)*. 1980, pp. 136–143 (cit. on p. 5).

[24] It is clear that one can have more than one index `e` conjoined to a single `a`, but such a construction doesn't seem to make much sense in presence of graded `Monad`s. However, that fact by itself does not exclude a possibility of existence of an interesting structure for which there are non-trivial interactions between different indexes `e`.

[9] R. Bailey. “A Hope Tutorial”. In: *Byte Magazine* 10.8 (Aug. 1985), pp. 235–255. ISSN: 0360-5280 (cit. on p. 5).

[10] J. Goguen, J. Thatcher, and E. Wagner. “An initial algebra approach to the specification, correctness, and implementation of abstract data types”. In: *Current Trends in Programming Methodology* 4 (1978). (also IBM Report RC 6487, Oct. 1976) (cit. on p. 5).

[11] M. Gogolla et al. “Algebraic and operational semantics of specifications allowing exceptions and errors”. In: *Theoretical Computer Science* 34.3 (Dec. 1984), pp. 289–313. ISSN: 0304-3975 (print), 1879-2294 (electronic) (cit. on p. 5).

[12] John B. Goodenough. “Exception Handling: Issues and a Proposed Notation”. In: *Communications of the ACM* 18.12 (Dec. 1975), pp. 683–696. ISSN: 0001-0782 (print), 1557-7317 (electronic) (cit. on p. 5).

[13] John B. Goodenough. “Exception handling design issues”. In: *ACM SIGPLAN Notices* 10.7 (July 1975), pp. 41–45. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/987305.987313 (cit. on p. 5).

[14] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0-201-11371-6 (cit. on pp. 5, 6).

[15] Andrew Koenig and Bjarne Stroustrup. “Exception Handling for C++ (revised)”. In: 1990, pp. 149–176 (cit. on p. 5).

[16] Bjarne Stroustrup. *The Design and Evolution of C++*. Reading, MA, USA: Addison-Wesley, 1994, pp. x + 461. ISBN: 0-201-54330-3 (cit. on pp. 5, 7).

[17] Eugenio Moggi. “Computational λ-Calculus and Monads”. In: *Logic in Computer Science (LICS)*. June 1989, pp. 14–23. URL: http://www.disi.unige.it/person/MoggiE/ftp/lics89.ps.gz (cit. on pp. 5, 6).

[18] Eugenio Moggi. “Notions of computation and monads”. In: *Information and Computation* 93.1 (1991). URL: http://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf (cit. on p. 5).

[19] Philip Wadler. “The essence of functional programming”. In: 1992, pp. 1–14 (cit. on pp. 5, 6).

[20] Wouter Swierstra. “Data Types à La Carte”. In: *Journal of Functional Programming* 18.4 (July 2008), pp. 423–436. ISSN: 0956-7968. DOI: 10.1017/S0956796808006758 (cit. on p. 5).

[21] Jose Iborra. “Explicitly Typed Exceptions for Haskell”. In: *PADL'10*. Jan. 2010 (cit. on pp. 5, 46, 49).

[22] Shin-ya Katsumata. “Parametric effect monads and semantics of effect systems”. In: *ACM SIGPLAN Notices* 49.1 (Jan. 2014). POPL '14 conference proceedings., pp. 633–645. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2578855.2535846 (cit. on pp. 5, 48, 49).

[23] Sheng Liang, Paul Hudak, and Mark Jones. “Monad transformers and modular interpreters”. In: 1995, pp. 333–343 (cit. on pp. 5, 16).

[24] Nick Benton, John Hughes, and Eugenio Moggi. “Monads and Effects”. In: *Lecture Notes in Computer Science* 2395 (2002), pp. 42–122. ISSN: 0302-9743 (print), 1611-3349 (electronic) (cit. on p. 5).

[25] Andy Gill and Ross Paterson. *Hackage: The transformers package, version 0.5.2.0*. 2016. URL: https://hackage.haskell.org/package/transformers-0.5.2.0 (cit. on pp. 5, 16, 17, 19, 30).

[26] Michael Sperber et al. *Revised^6 Report on the Algorithmic Language Scheme*. 1st. New York, NY, USA: Cambridge University Press, 2010. ISBN: 0521193990, 9780521193993 (cit. on pp. 5, 32).

[27] Kenichi Asai and Oleg Kiselyov. *Introduction to Programming with Shift and Reset*. Sept. 2011 (cit. on pp. 5, 32).

[28] Oleg Kiselyov. *An argument against call/cc*. 2012. URL: http://okmij.org/ftp/continuations/against-callcc.html (cit. on pp. 5, 7, 32).

[29] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. July 2001. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/ (cit. on pp. 5, 32).

[30] Kent M. Pitman. "Condition Handling in the Lisp Language Family". In: *Lecture Notes in Computer Science* 2022 (2001), 39–?? ISSN: 0302-9743 (print), 1611-3349 (electronic) (cit. on pp. 5, 6).

[31] Gordon D. Plotkin and Matija Pretnar. "Handlers of Algebraic Effects". In: 2009, pp. 80–94. DOI: 10.1007/978-3-642-00590-9_7 (cit. on p. 5).

[32] Edwin Brady. "Programming and reasoning with algebraic effects and dependent types". In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 133–144. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2544174.2500581 (cit. on pp. 5, 6, 49).

[33] Ohad Kammar, Sam Lindley, and Nicolas Oury. "Handlers in action". In: *ACM SIGPLAN Notices* 48.9 (Sept. 2013), pp. 145–158. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2544174.2500590 (cit. on p. 5).

[34] Oleg Kiselyov, Amr Sabry, and Cameron Swords. "Extensible effects: an alternative to monad transformers". In: *ACM SIGPLAN Notices* 48.12 (Dec. 2013). Haskell '14 conference proceedings., pp. 59–70. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2578854.2503791 (cit. on pp. 5, 6).

[35] Oleg Kiselyov and Hiromi Ishii. "Freer monads, more extensible effects". In: *ACM SIGPLAN Notices* 50.12 (Dec. 2015), pp. 94–105. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2887747.2804319 (cit. on p. 5).

[36] Aleksandar Nanevski. "Functional Programming with Names and Necessity". PhD thesis. Carnegie Mellon University, 2004. URL: http://software.imdea.org/~aleks/thesis/CMU-CS-04-151.pdf (cit. on pp. 5, 6).

[37] Aleksandar Nanevski. "A Modal Calculus for Exception Handling". In: *Intuitionistic Modal Logic and Applications Workshop (IMLA)*. A Logic in Computer Science Conference. Chicago, Illinois, USA, 2005. URL: http://software.imdea.org/~aleks/papers/effects/imla05.pdf (cit. on p. 5).

[38] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992) Open Group Technical Standard Base Specifications, Issue 6. New York, NY, USA: IEEE, 2001. ISBN: 1-85912-247-7 (UK), 1-931624-07-0 (US), 0-7381-3047-8 (print), 0-7381-3010-9 (PDF), 0-7381-3129-6 (CD-ROM) (cit. on p. 5).

[39] *Emacs Lisp Reference Manual: Catch and Throw*. URL: https://www.gnu.org/software/emacs/manual/html_node/elisp/Catch-and-Throw.html (cit. on p. 6).

[40] Simon Peyton Jones et al. "A Semantics for Imprecise Exceptions". In: *ACM SIGPLAN Notices* 34.5 (May 1999), pp. 25–36. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic) (cit. on pp. 6, 20, 46, 48).

[41] C++ FAQ. *Sequence Points*. URL: http://c-faq.com/expr/seqpoints.html (cit. on pp. 6, 20).

[42] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 28–38. ISBN: 0-89791-200-4. DOI: 10.1145/319838.319848 (cit. on p. 6).

[43] John M. Lucassen. "Types and effects, towards the integration of functional and imperative programming". PhD thesis. MIT Laboratory for Computer Science, Aug. 1987. URL: http://software.imdea.org/~aleks/thesis/CMU-CS-04-151.pdf (cit. on p. 6).

[44] John M. Lucassen and David K. Gifford. “Polymorphic effect systems”. In: *Principles of Programming Languages (POPL)*. Jan. 1988, pp. 47–57. URL: http://pag.lcs.mit.edu/reading-group/lucassen88effects.pdf (cit. on p. 6).

[45] Philip Wadler and Peter Thiemann. “The marriage of effects and monads”. In: *ACM Transactions on Computational Logic* 4.1 (Jan. 2003), pp. 1–32. URL: http://homepages.inf.ed.ac.uk/wadler/papers/effectstocl/effectstocl.ps.gz (cit. on pp. 6, 49, 50).

[46] Haskell Wiki Authors. *Typeclassopedia*. URL: https://wiki.haskell.org/Typeclassopedia (cit. on p. 9).

[47] John Hughes. “Generalising monads to arrows”. In: *Science of Computer Programming* 37.1–3 (2000), pp. 67–111. URL: http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf (cit. on p. 12).

[48] Conal Elliott. “Compiling to categories”. In: *Proc. ACM Program. Lang.* 1.ICFP (Sept. 2017). DOI: 10.1145/3110271. URL: http://conal.net/papers/compiling-to-categories (cit. on pp. 12, 49).

[49] M. P. Jones. “Functional Programming with Overloading and Higher-Order Polymorphism”. In: *Lecture Notes in Computer Science* 925 (1995), 97–?? ISSN: 0302-9743 (print), 1611-3349 (electronic) (cit. on p. 16).

[50] Simon Marlow. “An Extensible Dynamically-typed Hierarchy of Exceptions”. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*. Haskell ’06. Portland, Oregon, USA: ACM, 2006, pp. 96–106. ISBN: 1-59593-489-8. DOI: 10.1145/1159842.1159854 (cit. on p. 20).

[51] Edward Kmett. *Free Monads for Less (Part 3 of 3): Yielding IO*. 2011. URL: http://comonad.com/reader/2011/free-monads-for-less-3/ (cit. on p. 20).

[52] Andy Gill and Edward Kmett. *Hackage: The mtl package, version 2.2.1*. 2014. URL: https://hackage.haskell.org/package/mtl-2.2.1 (cit. on pp. 24, 30).

[53] Edward Kmett. *Hackage: The exceptions package, version 0.8.3*. 2015. URL: https://hackage.haskell.org/package/exceptions-0.8.3 (cit. on p. 25).

[54] John C. Reynolds. “The Discoveries of Continuations”. In: *Lisp and Symbolic Computation* 6.3/4 (Nov. 1993), pp. 233–248. ISSN: 0892-4635 (print), 1573-0557 (electronic) (cit. on p. 26).

[55] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992 (cit. on p. 26).

[56] Andrei Nikolaevich Kolmogorov. “On the principle of the excluded middle”. English. In: *From Frege to Gödel* (1971). Ed. by van Heijenoort, pp. 414–437 (cit. on p. 27). (Orig. pub. in 1925).

[57] Haskell Brooks Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic: Volume II*. Amsterdam, Holland: North-Holland, 1972 (cit. on p. 28).

[58] J. Steensgaard-Madsen. “Type Representation of Objects by Functions”. In: *ACM Transactions on Programming Languages and Systems* 11.1 (Jan. 1989), pp. 67–89. ISSN: 0164-0925 (print), 1558-4593 (electronic) (cit. on p. 28).

[59] Philip Wadler. “Deforestation: transforming programs to eliminate trees”. In: *Theoretical Computer Science* 73.2 (June 1990), pp. 231–248. ISSN: 0304-3975 (print), 1879-2294 (electronic) (cit. on p. 30).

[60] *Standard ML of New Jersey: The CONT signature*. URL: http://www.smlnj.org//doc/SMLofNJ/pages/cont.html#SIG:CONT.cont:TY (cit. on p. 32).

[61] Daan Leijen, Paolo Martini, and Antoine Latter. *Hackage: The Parsec package, version 3.1.11*. 2016. URL: https://hackage.haskell.org/package/parsec-3.1.11 (cit. on p. 32).

[62] Bryan O’Sullivan. *Hackage: The Attoparsec package, version 0.13.1.0*. 2016. URL: https://hackage.haskell.org/package/attoparsec-0.13.1.0 (cit. on p. 32).

[63] Mark Karpov et al. *Hackage: The megaparsec package, version 6.3.0*. 2017. URL: https://hackage.haskell.org/package/megaparsec-6.3.0 (cit. on p. 32).

[64] matt76k. *Hackage: ponder package, version 0.0.1*. 2014. URL: https://hackage.haskell.org/package/ponder-0.0.1 (cit. on p. 33).

[65] Pepe Iborra. *Hackage: The control-monad-exception package, version 0.11.2*. 2015. URL: http://hackage.haskell.org/package/control-monad-exception-0.11.2 (cit. on pp. 35, 46).

[66] Andy Sonnenburg. *Hackage: The catch-fd package, version 0.2.0.2*. 2012. URL: http://hackage.haskell.org/package/catch-fd-0.2.0.2 (cit. on p. 35).

[67] Gabriel Gonzalez and many others. *Hackage: The errors package, version 2.3.0*. 2018. URL: https://hackage.haskell.org/package/errors-2.3.0 (cit. on p. 40).

[68] Gert-Jan Bottu et al. "Quantified class constraints". In: *ACM SIGPLAN Notices* 52.10 (Oct. 2017), pp. 148–161. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/3156695.3122967. URL: https://homepages.inf.ed.ac.uk/wadler/papers/quantcc/quantcc.pdf (cit. on p. 40).

[69] Keunwoo Lee. *CSE341: Programming Languages: Scheme: Continuations*. 2004. URL: http://courses.cs.washington.edu/courses/cse341/04wi/lectures/15-scheme-continuations.html (cit. on p. 44).

[70] Wiki Books. *Scheme Programming: Continuations*. 2017. URL: https://en.wikibooks.org/w/index.php?title=Scheme_Programming/Continuations&oldid=3168913 (cit. on p. 44).

[71] Neil Mitchell. *Continuations and Exceptions*. 2014. URL: http://neilmitchell.blogspot.fr/2014/08/continuations-and-exceptions.html (cit. on p. 44).

[72] Neil Mitchell. *Shake*. URL: https://github.com/ndmitchell/shake (cit. on p. 44).

[73] Charles Anthony Richard Hoare. "The Emperor's Old Clothes". In: *Communications of the ACM* 24.2 (1981). This is the 1980 ACM Turing Award Lecture, delivered at ACM'80, Nashville, Tennessee, October 27, 1980., pp. 75–83. ISSN: 0001-0782 (print), 1557-7317 (electronic). DOI: 10.1145/358549.358561 (cit. on p. 49).

[74] Geoffrey Mainland. "Why It's Nice to Be Quoted: Quasiquoting for Haskell". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*. Haskell '07. Freiburg, Germany: ACM, 2007, pp. 73–82. ISBN: 978-1-59593-674-5. DOI: 10.1145/1291201.1291211 (cit. on p. 49).

[75] Andrzej Filinski. "Representing monads". In: 1994, pp. 446–457 (cit. on p. 50).