On the Expressive Power of Indexed Applicative and Monadic Structures

Jan Malakhovski

IRIT, University of Toulouse-3 and ITMO University

Version: 1.1; October 15, 2019

https://oxij.org/thesis/PhD/

General Introduction

History of Computing Hardware

Programs as linear sequences of instructions

- Colossus (1943-1945), early revisions of ENIAC (1945-1955): programmed using plugboards and mechanical switches.
- IBM 650 (1953-1969), the first mass-produced computer: each instruction's operation code (opcode) had to explicitly specify the address of the next instruction (similarly to how `jump` instructions of modern Assembly languages do).

  (Fun fact: theoretically speaking, modern CPUs implicitly incrementally transform programs they execute into continuation passing style (CPS), IBM 650 does not.)

Recursive subroutines

- PDP-11 (1970-1990): the first computer with proper hardware support for subroutine calls of non-fixed-level nesting depth (that is, supporting recursion and arbitrary modularity).

Together

Immediately after: UNIX, the C programming language, and the rest of modern computing.

History of Programming Languages

The first two high-level programming languages:

- FORTRAN: initially developed by John Backus at IBM (for IBM 704) around 1956 (first compiler in 1957)
- LISP initially developed by John McCarthy at MIT around the same time (first specified in 1958, first universal interpreter implemented by Steve Russell for IBM 704 around 1960, first compiler written in LISP in 1962).

(Fun fact: `car` and `cdr` of LISP are names of IBM 704 assembly language macros that were used to implement those operations.)

History of Programming Languages

These gave birth to two language families:

- Imperative compiled strictly-typed languages (FORTRAN, ALGOL, C, etc).
  - Straightforward attempts to make universal high-level "Assembly" languages.
  - Designed to be efficiently compiled into programs efficiently computable on real hardware.
- Functional interpreted dynamically-typed languages (LISP, Common LISP, Scheme, etc).
  - Adaptations of λ-calculus to practical programming and meta-programming.
  - Designed to have as much expressive power as possible, computational efficiency was a secondary concern.

History of Programming Languages

Descendants of those "evolved" by mutation and exchange of ideas. Two very successful "cross-bred" new language families were produced as the result:

- Imperative (usually) interpreted dynamically-typed languages (Smalltalk, Python, Ruby, JavaScript, etc).
- Functional (usually) compiled statically-typed languages (ML, OCaml, SML, Haskell, Idris, Rust, etc).

The recent rise of popularity of the latter family of languages means that the theory, finally, gets adopted into mainstream practice.

History of Programming Languages

Among those, the sequence LISP $\rightarrow$ ML $\rightarrow$ Miranda $\rightarrow$ Haskell is rather interesting:

- LISP $\rightarrow$ ML replaced dynamic typing with a polymorphic type system and infix syntax at the cost of losing both special forms and reflection,
- ML $\rightarrow$ Miranda switched to lazy evaluation by default ($\sim$special forms),
- Miranda $\rightarrow$ Haskell added type classes ($\sim$dynamic types), reintroduced reflection, among many other things.

In other words, Haskell was designed to conveniently express things commonly discussed in Programming Languages Theory (PLT).

Summing It Up

- Programs represented using function calls and linear sequences of instructions gave birth to modern computing.
- It is well-known that first-class functions (functions are values) of functional programming produce a lot of useful expressive power.
- Generalizing those things even further might give even more expressive power.
- Haskell was designed to conveniently express such things. It makes sense to discuss those generalizations using Haskell.
- As we shall see below, Haskell already has those things generalized into `Applicative` and `Monad` type classes. Those are well-known among Haskell programmers, and programs in Haskell use these generalizations ubiquitously.

Short Abstract

Programmer in Haskell

It is conventionally argued that many commonly useful things can't be expressed using `Applicative` and `Monad` type classes and require special attention (e.g. `throw`/`catch`, tree transformations, generalized patter matching).
This work aims to show that many of those examples can, in fact, be expressed by reusing those well-known structures with minor (if any) modifications.

Researcher in PLT

This work aims to apply the

KISS (Keep It Stupid Simple)

principle to programming language design.

Technical Introduction

Syntax

Math

$$f(a, b, n, m) \mapsto a\ sin(2\alpha) + b\ cos(\alpha) + mod(n, m)$$

$$(f\ .\ g)(x) \mapsto f(g(x))$$

Haskell

```
f a b n m = a * sin (2 * alpha) + b * cos alpha
          + fromInteger (mod n m)

f . g = \x -> f (g x)
-- or
(.) f g = \x -> f (g x)
-- or
(.) f g x = f (g x)
```

Haskell Syntax: Types

```
f :: Rational -> Rational -> Integer -> Integer
  -> Rational
f a b n m = a * sin (2 * alpha) + b * cos alpha
          + fromInteger (mod n m)

(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Haskell Syntax: Data Types

```haskell
type Const a b = a

data Struct a b c
  = NothingOfValue
  | SomeFields
    { firstField :: a
    , secondField :: b
    , thirdField :: c
    }

newtype FunArrow a b
  = FunArrow { runFunArrow :: a -> b }
```

Haskell Syntax: Type Classes

```haskell
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c

data Category cat = Category
  { id :: cat a a
  , (.) :: cat b c -> cat a b -> cat a c
  }
```

Haskell Syntax: Type Class Instances

```
instance Category (->) where
  id x = x
  (.) f g x = f (g x)

arrCategory = Category
  { id = \x -> x
  , (.) = \f g x -> f (g x)
  }

comp3 :: Category cat => cat c d -> cat b c
      -> cat a b -> cat a d
comp3 f g h = f . g . h
```

Why are type classes useful?

Mathematician

They can describe common algebraic structures. They allow to reason about programs in terms of those structures.

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

instance Monoid Integer where
  mempty = 0
  mappend = (+)
```

Programmer

They are “interfaces” in OOP-speak. They allow one to write generic functions applicable to many different types. That is, they are a mechanism of ad-hoc polymorphism.

What is an Applicative? Monad?

- A bit unconventionally defined hierarchy

```haskell
-- Inject a pure value into computations over `f`
class Pointed f where
  pure :: a -> f a

-- Of no consequence for this work
class Functor f where
  fmap :: (a -> b) -> f a -> f b

-- Generalized application
class (Pointed f, Functor f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b -- "apply"

-- Generalized semicolon operator
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b -- "bind"
```

(<*>) is a generalized application operator

- Take normal application and wrap everything with `f`

```
($) :: (a -> b) -> a -> b
f $ x = f x

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

(>>=) is a generalized semicolon operator

```
something_t main () {
  x = foo();
  y = bar(x);
  return baz(x, y);
}
```

- Assume everything is pure (without side-effects).
- Then, in Haskell/ML syntax this becomes

```
let x = foo in
let y = bar x in
baz x y
```

(>>=) is a generalized semicolon operator

- Remember that, in general,

```
let x = f args in rest
```

- can be encoded in pure λ-calculus as follows

```
(\x -> rest) (f args)
```

(>>=) is a generalized semicolon operator

- In other words,

```
x = foo();
y = bar(x);
baz(x, y);
```

- can be encoded as

```
(\x -> (\y -> baz x y) (bar x)) foo
```

- which can be encoded as

```
let andThenContinueTo x f = f x in

foo   `andThenContinueTo` (\x ->
bar x `andThenContinueTo` (\y ->
baz x y))
```

What is the type of andThenContinueTo?

```
foo `andThenContinueTo` (\x -> rest)
```

- Obviously, this

```
andThenContinueTo :: a -> (a -> b) -> b
andThenContinueTo x f = f x
```

- Thus, (>>=) is just a generalization of this type!

```
class Applicative f => Monad f where
  (>>=) :: f a -> (a -> f b) -> f b
```

- Note (a -> f b), not f (a -> b), the variable a plays a variable binding here, not an argument to a function over f.
- Obviously

```
type Identity a = a
instance Monad Identity where
  (>>=) = andThenContinueTo
```

Consider this

```
main =
  foo   >>= (\x ->
  bar x >>= (\y ->
  baz x y))
```

- For `Identity`

```
instance Monad Identity where
  (>>=) :: a -> (a -> b) -> b
  (>>=) = andThenContinueTo

foo :: a
bar :: a -> b
baz :: a -> b -> c
```

- Thus

```
main :: c
```

But there are other options, e.g.

```
main =
  foo   >>= (\x ->
  bar x >>= (\y ->
  baz x y))
```

- For `(->) r`

```
instance Monad ((->) r) where
  (>>=) :: (r -> a) -> (a -> r -> b) -> (r -> b)
  (>>=) ra f r = f (ra r) r

foo :: r -> a
bar :: a -> r -> b
baz :: a -> b -> r -> c
```

- Thus

```
main :: r -> c
```

Consider what we just did

- We took an "imperative" program made of assignments

```
main =
  foo   >>= (\x ->
  bar x >>= (\y ->
  baz x y))
```

- and made a constant environment `r` propagate into all sub-expressions

```
instance Monad ((->) r) where
  (>>=) :: (r -> a) -> (a -> r -> b) -> (r -> b)
  (>>=) ra f r = f (ra r) r
```

- by simply requesting an instance of `main` of a different type. Thus, effectively, we changed semantics of `main` without actually changing a single line of code in `main`!

Side-note: syntax sugar

- For expressions like this

```
main =
  foo   >>= (\x ->
  bar x >>= (\y ->
  baz x y))
```

- Haskell has the following syntax sugar

```
main = do
  x <- foo
  y <- bar x
  baz x y
```

Side-note: Pointed, Monad $\mapsto$ Functor, Applicative

```
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
```

Also, Pointed, Applicative $\rightarrow$ Functor

```
apfmap :: Applicative f => (a -> b) -> m a -> m b
apfmap f ma = pure f <*> ma
```

Why are `Applicative` and `Monad` useful?

- They can be used to express generic operators. E.g.

```
mapM :: Monad f => (a -> f b) -> [a] -> f [b]
mapM f     [] = pure []
mapM f (a:as) = do
  b <- f a
  bs <- mapM f as
  pure (b:bs)
```

- Using generic operators reduces boilerplate somewhat by allowing for generic combinators.

Why are Applicative and Monad useful?

- More importantly, those structures provide a convenient level of abstraction that hides irrelevant details.

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

instance Pointed (Either e) where
  pure = Right

instance Monad (Either e) where
  Left  e >>= _ = Left e
  Right a >>= f = f a

dossier name = do
  uid  <- getUidByName name
  birth<- getDateOfBirthByUid uid
  addr <- getAddressByUid uid
  pure (name, birth, addr)
```

Why are `Applicative` and `Monad` useful?

In other words, they allow to define "functional" and "imperative" domain specific languages (DSLs) in Haskell.

- Example: parser combinator libraries.

What are parsers combinators?

- Embedded domain specific languages (eDSLs) that allow one to describe parsers.
- That is, they are eDSLs to parse stuff without any special tools like Yacc, GNU Bison, etc.
- Usually they parse PEG [Ford, 2004].

Example: A Tiny Parser Combinator Library

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

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 e -> Left e
      Right (a, s') -> runSParser (f a) s'

f <|> g = SParser $ \s -> case runSParser f s of
  Right x -> Right x
  Left  e -> case runSParser g s of
    Right x -> Right x
    Left  e' -> Left (e `mappend` e')
```

Example: A Tiny Parser Combinator Library

```
type Failures = [String]

type Parser a = SParser String Failures a
```

Example: A Tiny Parser Combinator Library

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

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

string :: String -> Parser ()
string [] = pure ()
string (c:cs) = char c >>= \_ -> string cs
```

Example: A Tiny Parser Combinator Library

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

- Fun fact: This file was compiled from Org-Mode to TeX using Pandoc, which is implemented using a similar parser combinator library.

Exceptionally Monadic Error Handling

Exceptions

Python

```
try: expression1
except Type as e: expression2
```

C++/Java/etc

```
try { expression1 } catch (type_t e) { expression2 }
```

Haskell/etc

```
foo = expression1 `catch` (\e :: Type -> expression2)
```

Contribution #1

`catch` without dynamic dispatch is `Monad`ic `(>>=)`!

Proof: Pragmatic programmer's view

- Let us say, we have

```
data C e a
```

- let us assume, we know it is a `Monad`

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

- we want to add

```
throw :: e -> C e a
catch :: C e a -> (e -> ?) -> ?
```

- what should be the type of `catch`? Generally,

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

Proof: Unification

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

- Unify the types in

```
pure a `catch` f == pure a
-> c == a
throw e `catch` (\_ -> pure a) == pure a
-> c == b (== a)
throw e `catch` (\_ -> throw f) == throw f
-> g == f
```

- Which gives

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

Proof: `e == f`?

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

- No:
  - If computation `throws` 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`,
  - these two cases are mutually exclusive.
- Thus, `catch` has exactly the type of `(>>=)` in index `e`.

Monad laws

But having the same type is not enough to be a proper `Monad`. It needs to satisfy the `Monad`ic laws.

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

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

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

Definition: ConjoinedMonads

We shall say that a type `m` with two indexes is an instance of `ConjoinedMonads` iff it is a `Monad` separately in each index

```
class ConjoinedMonads m where
  pure  :: 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
```

it satisfies the `Monad`ic laws in both, and the following additional equations hold

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

So, are there any instances?

Yes, lots.

Instance: Either

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

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

Proof

Since `r = \a -> Either e a` is a `Monad`,
`r = \e -> Either e a` is also a `Monad`. `Either` is $\vee$, which is symmetric. Additional equations are easy to check.

Instances: All instances of `MonadError`

This one is easy to prove.

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

Proof

The above implies `e` is constant for each given `m`. Which means that if we substitute `r = \e a -> m a` into an instance of `ConjoinedMonads` we'll get an instance of `MonadError`. (Since it already requires `Monad m`.)

Instance: Parser combinators

- The `Monad` in `e`

```
throwSP :: e -> SParser s e a
throwSP e = SParser $ \_ -> Left e

catchSP :: SParser s e a
        -> (e -> SParser s f a) -> SParser s f a
catchSP p f = SParser $ \s ->
  case runSParser p s of
    Right x -> Right x
    Left e -> runSParser (f e) s
```

Instance: Parser combinators

```
-- zero or more `p`
many :: ConjoinedMonads m => m e a -> m f [a]
many p = some p `catch` \_ -> pure []
    -- instead of <|>

-- one or more `p`
some :: ConjoinedMonads m => m e a -> m e [a]
some p = fmap (:) p <*> many p
```

Instance: More

- Other parser combinators, similarly (easy).
- All `MonadThrow` and `MonadCatch` instances without dynamic dispatch (somewhat involved).
- Exception emulation with continuations (somewhat involved).
- And many others.

See the thesis for details.

Contribution #2

`ConjoinedMonads` is not the only useful Cartesian product of type classes.

For instance (see the thesis for details)

- `Monad` × `Applicative` adds throw/catch error handling to `Applicative` expressions.
- Thus, throw/catch error handling × pure functions is an instance of `Monad` × `Applicative`.
- With some modifications to the language this can work even GHC's imprecise exceptions × pure functions.

Other parts of the work also make interesting Cartesian products.

Contribution #2.5

- Our results adhere to the general structure of the "marriage" framework of [Wadler and Thiemann, 2003].
- Therefore, all results applicable there are applicable here.
- For instance, one can take a Cartesian product with some graded `Monad` of [Katsumata, 2014], thus producing an "effect system".

Transforming Trees with Generalized `Applicative` Expressions

The Observation

Applicative parsers are cute!

```
data Device = Device
  { block :: Bool
  , major :: Int
  , minor :: Int }

exampleDevice :: Device
exampleDevice = Device False 19 1

class Parsable a where
  parse :: Parser a

instance Parsable Device where
  parse = pure Device <*> parse <*> parse <*> parse
```

The Idea

Why can't we do the same for other kinds transformations?

```
parseDevice
  = pure Device <*> parse <*> parse <*> parse

-- pretty-printing
showDevice
  = depure unDevice <**> show <**> show <**> show

-- "mapping" between values
mapDevice
  = depure unDevice <**> not <**> (+ 100) <**> (+ 200)
```

The Idea Formalized: `Applicative`-like

- The `Applicative` operator

```
(<*>) :: f (a -> b) -> f a -> f b
```

- follows the following pattern

```
plug :: f full -> g piece -> f fullWithoutThePiece
```

- contrast to the usual composition

```
compose  :: f fullWithoutThePiece -> g piece -> f full
-- or
compose' :: g piece -> f fullWithoutThePiece -> f full
```

- The main point is to decide on all data types and the way to assemble results *first* and then delegate handing of parts to subcomputations.
- Such operators we shall call "`Applicative`-like".

Can we use the same idea for other kinds of transformations?

Yes, we can!

- "Destruct" the data type

```
unDeviceLISP :: Device -> (Bool, (Int, (Int, ())))
unDeviceLISP (Device b x y) = (b, (x, (y, ())))
```

- Invent a plug

```
chop :: (s, (a, b)) -> (s -> a -> t) -> (t, b)
chop (s, (a, b)) f = (f s a, b)

chopR :: (r -> (s, (a, b)))
      -> (s -> a -> t) -> (r -> (t, b))
chopR o f r = chop (o r) f
```

Pretty-printer: Done!

```
showDevice :: Device -> ([String], ())
showDevice = depureShow unDeviceLISP `showa` show
                                     `showa` show
                                     `showa` show

depureShow :: (r -> b) -> r -> ([String], b)
depureShow f r = ([], f r)

showa :: (r -> ([String], (a, b)))
      -> (a -> String)
      -> (r -> ([String], b))
showa st f = chopR st (\s a -> (f a):s)

testShowDevice :: String
testShowDevice = runShow $ showDevice exampleDevice
  -- == "False 19 1"
```

Then, this can be trivially extended to

- maps between data types

```
mapDevice :: Device -> (Device, ())
mapDevice = depureMap Device unDeviceLISP
  `mapa` not
  `mapa` (+ 100)
  `mapa` (+ 200)

testMapDevice :: Device
testMapDevice = runMap $ mapDevice exampleDevice
  -- == Device True 119 201
```

Then, this can be trivially extended to

- `zip`s between data types

```
zipDevice :: Device -> Device -> (Device, (), ())
zipDevice = depureZip Device unDeviceLISP unDeviceLISP
  `zipa` (&&)
  `zipa` (+)
  `zipa` (+)

testZipDevice :: Device
testZipDevice = runZip
  $ zipDevice exampleDevice testMapDevice
  -- == Device False 138 202
```

- and similarly for more structures

Then, this can be trivially extended to

- Generalized “stack”-machine computations (with arbitrary data types for “stacks”)

```
remapDevice :: Device -> (Device, ())
remapDevice = depureMap Device unDeviceLISP
  `andThen` pop
  `push` True
  `mapa` id
  `andThen` pop
  `andThen` dup
  `mapa` id
  `mapa` id

testRemapDevice :: Device
testRemapDevice = runMap $ remapDevice exampleDevice
  -- == Device True 1 1
```

Scott-encoding

- "Destruction" to LISP is not the only possible way, this also works with Scott-encoding

```
unDeviceScott :: Device
  -> (Bool -> Int -> Int -> c) -> c
unDeviceScott (Device b x y) f = f b x y

chopS :: ((s -> a -> b) -> c)
      -> (s -> a -> t)
      -> ((t -> b) -> c)
chopS i f o = i $ \s a -> o (f s a)
```

- Examples look exactly the same, only the types change.
- But `chops` that `fold` more than one type at the same time are a bit involved. See the thesis.

Formal Account

- `Applicative` is generalized λ-application.
- Our structures are instances of generalized dependently typed λ-application!

```
class ApplicativeLike f where
  type C f a b :: * -- type of arrow under `f`
  type G f a :: *   -- type of argument dependent on `f`
  type F f b :: *   -- type of result dependent on `f`
  (<**>) :: f (C f a b) -> G f a -> F f b
```

Contributions

#3

`Applicative` can be naturally generalized into a dependently typed structure we call `ApplicativeLike`. (Which happens to be both simpler and more general than superapplicatives of [Bracker and Nilsson, 2018].)

#4

It has a bunch of practically useful instances: pretty-printing, `mapping`, `zipping`, etc for simple data types of a single constructor.

Contribution #5

We can make instances of Cartesian products of `Monad` and `ApplicativeLike`.

```
chopE :: Either e (s, (a, b))
      -> (s -> a -> Either e t)
      -> Either e (t, b)
chopE i f = do
  (s, (a, b)) <- i
  fsa <- f s a
  pure (fsa, b)

chopES :: ((Either e s -> a -> b) -> c)
       -> (s -> a -> Either e t)
       -> ((Either e t -> b) -> c)
chopES i f o = i $ \s a -> o (s >>= \s' -> f s' a)
```

Transforming Trees with Indexed Monads

The Idea

Monadic parser combinators are cute!

Can't we make them "parse" arbitrary data types/trees, not just Streams (Strings)?

- The problem

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

- The solution

```
newtype IxSParser e i j a = IxSParser
  { runIxSParser :: i -> Either e (a, j) }
```

Indexed Monads

```
class IxPointed m where
  ipure :: a -> m i i a

class IxFunctor f where
  ifmap :: (a -> b) -> f i j a -> f i j b

class (IxPointed m, IxFunctor m) => IxApplicative m where
  (<*+>) :: m i j (a -> b) -> m j k a -> m i k b

class IxApplicative m => IxMonad m where
  (>>=+) :: m i j a -> (a -> m j k b) -> m i k b
```

Yes, we can!

- For instance

```
newtype IxSParser e i j a = IxSParser
  { runIxSParser :: i -> Either e (a, j) }

instance IxPointed (IxSParser e) where
  ipure a = IxSParser $ \i -> Right (a, i)

instance IxMonad (IxSParser e) where
  p >>=+ f = IxSParser $ \i ->
    case runIxSParser p i of
      Left x -> Left x
      Right (a, j) -> runIxSParser (f a) j
```

Yes, we can!

- The other index

```
throwIxSP :: e -> IxSParser e i j a
throwIxSP e = IxSParser $ \_ -> Left e

-- Note that this keeps indices as is,
-- since it is a `Monad`,
-- not `IxMonad` in `e`
catchIxSP :: IxSParser e i j a
          -> (e -> IxSParser f i j a)
          -> IxSParser f i j a
catchIxSP m f = IxSParser $ \i ->
  case runIxSParser m i of
    Right x -> Right x
    Left e -> runIxSParser (f e) i
```

Cartesian Product

```
class MonadXIxMonad m where
  icpure  :: a -> m e i i a
  icbind  :: m e i j a -> (a -> m e j k b) -> m e i k b

  icthrow :: e -> m e i j a
  iccatch :: m e i j a -> (e -> m f i j a) -> m f i j a
```

Interesting Parts

- It can actually "parse" arbitrary data types/trees. (See the thesis.)
- Combinators `many` and `some` are as interesting as before.
- But now also

```
-- zero of more `p` separated by `sep`
sepBy :: MonadXIxMonad m
      => m e i i a -> m e i i b -> m f i i [a]
sepBy p sep = _

-- one or more `p` separated by `sep`, that is
-- `p` followed by zero or more `sep >> p`
sepBy1 :: MonadXIxMonad m
       => m e i j a -> m e j i b -> m e i j [a]
sepBy1 p sep = _

exampleSepBy1 = sepBy1 integer comma
```

But there is more!

```
class IxMonadXIxMonad m where
  iicpure  :: a -> m e i u i a
  iicbind  :: m e i u j a
           -> (a -> m e j u k b) -> m e i u k b

  iicthrow :: e -> m e i i u a
  iiccatch :: m e i j u a
           -> (e -> m f j k u a) -> m f i k u a
```

- And this also has an instance.

Contributions

#6

`Monad`ic parser combinators can be generalized into indexed `Monad`ic "parser" combinators for arbitrary data types/trees.

#7

They are instances of `MonadXIxMonad` and `IxMonadXIxMonad` Cartesian products.

Conclusions

In Terms of Related Works

- The first part, essentially, extends the work of [Wadler, 1992] by showing that `Monad`s can also be used for proper error handling (and not just "hiding errors from the higher-level interpreter"), the observation which we formalized into `ConjoinedMonads` structure.

  Similarly to how Wadler's `Monad` instances influenced the design of modern Haskell, our instances also hint at new language design opportunities.

- The second part extends the work of [McBride and Paterson, 2008] on `Applicative`s by showing other interesting structures that follow the same general form of expressions but allow for more sophisticated transformations, a structure which we formalized into the `ApplicativeLike` type class.

- The third part, essentially, extends works on parser combinators, most notably the work of [Leijen and Meijer, 2001], to "parsing" arbitrary data types/trees.

The Main Points

- `catch` without dynamic dispatch is `Monad`ic `(>>=)`. Lots of instances. (But mostly informative, hard to apply to practice at the moment.)
- `Applicative` can be generalized into dependent types to produce `ApplicativeLike`. Lots of immediately practically useful instances.
- `Monad`s when generalized into indexed `Monad`s have "parser" combinators that can "parse" data types/trees as instances.
- Many interesting structures can be represented as Cartesian products of type classes where one part represents error handling, while the other represents normal computation.

Questions?

Publications

- Sergei Soloviev, Jan Malakhovski. Automorphisms of Types and Their Applications. Zapiski nauchnykh seminarov POMI. Volume 468, 2018.
- Sergei Soloviev, Jan Malakhovski. Automorphisms of Types and Their Applications. Journal of Mathematical Sciences Volume 240, Issue 5, August 2019, Pages 692-706.
- Jan Malakhovski. Exceptionally Monadic Error Handling. Journal of Functional Programming. Submitted January 2019, positive reaction (August 2019), to be resubmitted after a re-edit with reviewer's comments.
- Jan Malakhovski, Sergei Soloviev. Programming with Applicative-like Expressions. Journal of Functional Programming. Under consideration, submitted August 2019.


Bracker, Jan and Henrik Nilsson (2018). "Supermonads and superapplicatives". In: *Journal of Functional Programming*. Under consideration, submitted 12 December 2017 (cit. on p. 63).


Ford, Bryan (Jan. 2004). "Parsing expression grammars: a recognition-based syntactic foundation". In: *ACM SIGPLAN Notices* 39.1, pp. 111–122. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic) (cit. on p. 31).


Katsumata, Shin-ya (Jan. 2014). "Parametric effect monads and semantics of effect systems". In: *ACM SIGPLAN Notices* 49.1. POPL '14 conference proceedings., pp. 633–645. ISSN: 0362-1340 (print), 1523-2867 (print), 1558-1160 (electronic). DOI: 10.1145/2578855.2535846 (cit. on p. 51).


Leijen, Daan and Erik Meijer (July 2001). *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. URL: https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/ (cit. on p. 75).


McBride, Conor and Ross Paterson (2008). "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1, pp. 1–13. URL: http://www.soi.city.ac.uk/~ross/papers/Applicative.pdf (cit. on p. 75).


Wadler, Philip (1992). "The essence of functional programming". In: *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*. Ed. by ACM. ACM order number 54990. New York, NY, USA: ACM Press, pp. 1–14. ISBN: 0-89791-453-8 (cit. on p. 75).


Wadler, Philip and Peter Thiemann (Jan. 2003). "The marriage of effects and monads". In: *ACM Transactions on Computational Logic* 4.1, pp. 1–32. URL: http://homepages.inf.ed.ac.uk/wadler/papers/effectstocl/effectstocl.ps.gz (cit. on p. 51).