

Selective Applicative Functors

Andrey Mokhov, Georgy Lukyanov, **Simon Marlow**, Jeremie Dimino

Copenhagen, 26 April 2019

In the beginning...

In the beginning...

- There were no Monads

In the beginning...

- There were no Monads
 - (the less said about this era the better)

Then...

Philip Wadler: 1995, Glasgow

Monads for functional programming

Philip Wadler, University of Glasgow★

Department of Computing Science, University of Glasgow, G12 8QQ, Scotland
(wadler@dcs.glasgow.ac.uk)

Abstract. The use of monads to structure functional programs is described. Monads provide a convenient framework for simulating effects found in other languages, such as global state, exception handling, output, or non-determinism. Three case studies are looked at in detail: how monads ease the modification of a simple evaluator; how monads act as the basis of a datatype of arrays subject to in-place update; and how monads can be used to build parsers.

Monads provided a beautiful way to embed I/O in a purely functional language...

- Provide an abstract type `IO a` meaning
 - *computations that may do I/O and then return a value of type a*
- Then we need primitives, like

```
getLine :: IO String
putStrLn :: String -> IO ()
```

We need a way to compose I/O

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

Now we can write programs:

```
greeting :: IO ()  
greeting = getLine >>= \name -> putStrLn ("Hello " ++ name)
```


IO is not the only Monad...

- Monads abstract over different notions of computation
- Useful examples of different Monads:
 - Maybe (simple failure), or Either (exceptions)
 - State
 - Reader (environment, configuration)
 - Writer (output)
 - Lists (non-determinism, search)
 - Continuations (cooperative concurrency)

Generic Monads

In Haskell we abstract over Monads with a type class:

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

Which means we can write generic Monad combinators, e.g.

```
sequence :: Monad m => [m a] -> m [a]
filterM  :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

Motivating example

- “read a string, if it is ‘ping’ then print ‘pong’, otherwise do nothing”

```
pingPongM :: IO ()
pingPongM =
  getLine >>= \s ->
    if s == "ping" then putStrLn "pong" else pure ()
```

What if we want to analyse it?

```
pingPongM :: IO ()
pingPongM =
  getLine >>= \s ->
    if s == "ping" then putStrLn "pong" else pure ()
```

- Sometimes it's useful to be able to ask “what are all the effects this computation might have?”
- We could use this to
 - pre-allocate resources
 - speculate execution, parallelism
 - (examples coming later)

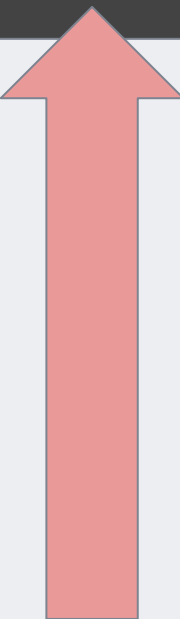
But we cannot do that here!

```
pingPongM :: IO ()  
pingPongM =  
  getLine >>= \s ->  
    if s == "ping" then putStrLn "pong" else pure ()
```

Only known at *runtime*

In general, Monad makes this impossible

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



- We cannot know **a** until we have performed **f a**
- So we cannot analyse the computation to find all its (potential) effects, we can only run it.

But let's take a simpler example

```
whenM :: Monad m => m Bool -> m () -> m ()
```

first execute this...

But let's take a simpler example

```
whenM :: Monad m => m Bool -> m () -> m ()
```

first execute this...

if it returned True,
execute this,
otherwise don't

Rewrite our example using whenM

```
whenM :: Monad m => m Bool -> m () -> m ()
```

We will need fmap:

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

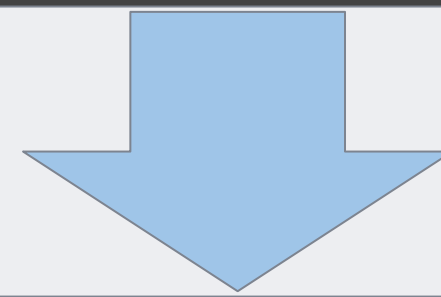
Now, to get IO Bool:

```
fmap (== "ping") getLine :: IO Bool
```

Rewrite our example using whenM

```
whenM :: Monad m => m Bool -> m () -> m ()
```

```
pingPongM :: IO ()  
pingPongM =  
  getLine >>= \s ->  
    if s == "ping" then putStrLn "pong" else pure ()
```



```
pingPongM :: IO ()  
pingPongM =  
  whenM (fmap (== "ping") getLine) (putStrLn "pong")
```

But why is this better?

- Look at the definition of whenM:

```
whenM :: Monad m => m Bool -> m () -> m ()  
whenM x y = x >>= \b -> if b then y else return ()
```

Still a *runtime* value, but
it only has two possible
values

- *We have some hope of statically analysing this code, because we can enumerate all the possibilities for b*

But why is this better?

- Look at the definition of whenM:

```
whenM :: Monad m => m Bool -> m () -> m ()  
whenM x y = x >>= \b -> if b then y else return ()
```

Still a *runtime* value, but
it only has two possible
values

- *We have some hope of statically analysing this code, because we can enumerate all the possibilities for b*
- *But we can't do it in this form, using >>=*

But wait...

- Don't we already have an abstraction that...
 - is weaker than Monad
 - admits static analysis

Applicative Functors: 2007, Nottingham/London

FUNCTIONAL PEARL

Idioms: applicative programming with effects

CONOR MCBRIDE

University of Nottingham

ROSS PATERSON

City University, London

Abstract

In this paper, we introduce **Idioms**—an abstract characterisation of an applicative style of effectful programming, weaker than **Monads** and hence more widespread. Indeed, it is the ubiquity of this programming pattern which drew us to the abstraction. We shall take the same course in this paper, introducing the applicative pattern by diverse examples, then abstracting it to define the **Idiom** type class and associated laws. We compare this abstraction with monoids, monads and arrows, and identify the categorical structure of idioms.

Applicative Functors

```
class Applicative f where
  pure   :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

✓ We can execute computations $f (a \rightarrow b)$ and $f a$ in parallel (if we like).

✓ All effects are statically visible and can be examined before execution.

✗ Computations must be independent, hence **no conditional execution**.

Ping-pong example: applicative functors

Task: Input a string, and if it equals “ping” then output “pong”.

Ping-pong example: applicative functors

x

Task: Input a string, and if it equals “ping” then output “pong”.

Ping-pong example: applicative functors

Task: Input a string, and if it equals “ping” then output “pong”.

```
pingPongA :: IO ()
```

```
pingPongA = fmap (\s -> id) getLine <*> putStrLn "pong"
```

$\text{IO } () \rightarrow ()$

$\text{IO } ()$

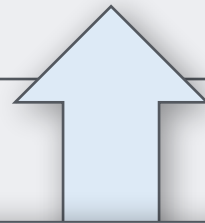
```
λ> pingPongA
ping
pong
```

```
λ> pingPongA
hello
pong
```

Towards a new intermediate abstraction

	Applicative functors	???	Monads

Towards a new intermediate abstraction

	Applicative functors	???	Monads
Independent effects & parallelism	✓		✗
 $(\times) :: f\ a \rightarrow f\ b \rightarrow f\ (a, b)$			

Towards a new intermediate abstraction

	Applicative functors	???	Monads
Independent effects & parallelism	✓		✗
Static visibility & analysis of effects	✓		✗

`getPure` `:: f a -> Maybe a`
`getEffects` `:: f a -> [f ()]`

Towards a new intermediate abstraction

	Applicative functors	???	Monads
Independent effects & parallelism	✓		✗
Static visibility & analysis of effects	✓		✗
Dynamic generation of effects	✗		✓
<pre>greeting = getLine >>= \name -> putStrLn ("Hello " ++ name)</pre>			

Towards a new intermediate abstraction

	Applicative functors	???	Monads
Independent effects & parallelism	✓		✗
Static visibility & analysis of effects	✓		✗
Dynamic generation of effects	✗		✓
Conditional execution of effects	✗		✓

pingPongM = whenM (fmap (=="ping") getLine) (putStrLn "pong")

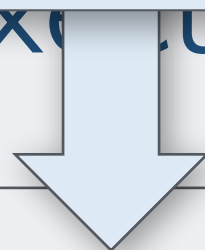
Towards an intermediate abstraction

	Applicative functors	???	Monads
Independent effects & parallelism	✓		✗
Conditional execution of effects	✗		✓
Speculative execution of effects	✗		✗

Ad-hoc speculative execution combinators from the Haxl library:

```
pAnd :: f Bool -> f Bool -> f Bool
```

```
pOr  :: f Bool -> f Bool -> f Bool
```



Towards an intermediate abstraction

	Applicative functors	Selective functors	Monads
Independent effects & parallelism	✓		✗
Static visibility & analysis of effects	✓		✗
Dynamic generation of effects	✗		✓
Conditional execution of effects	✗		✓
Speculative execution of effects	✗		✗

Towards an intermediate abstraction

	Applicative functors	Selective functors	Monads
Independent effects & parallelism	✓	✓	✗
Static visibility & analysis of effects	✓	✓	✗
Dynamic generation of effects	✗	✗	✓
Conditional execution of effects	✗	✓	✓
Speculative execution of effects	✗	✓	✗

Selective Applicative Functors

- Goal: an abstraction that allows
 - static analysis, parallelism, speculative execution
 - conditional effects

Selective Applicative Functors

- Goal: an abstraction that allows
 - static analysis, parallelism, speculative execution
 - conditional effects

```
class Applicative f => Selective f where  
  select :: f (Either a b) -> f (a -> b) -> f b
```

The first computation is used to *select what happens next*:

- Left *a*: you *must execute* the second computation to produce a *b*;
- Right *b*: you *may skip* the second computation and return the *b*.

Selective Applicative Functors

```
class Applicative f => Selective f where  
  select :: f (Either a b) -> f (a -> b) -> f b
```

- ✓ We can speculatively execute both computations in parallel (if we like).
- ✓ All effects are statically visible and can be examined before execution.
- ✓ A limited form of dependence, sufficient for conditional execution.

Why this particular formulation?

```
class Applicative f => Selective f where
  select :: f (Either a b) -> f (a -> b) -> f b
```

- Parametricity tell us what select can do
 - **whenM** can be implemented wrongly (**unlessM**)

But we love operators, so

```
(<*?) :: Selective f => f (Either a b) -> f (a -> b) -> f b  
(<*?) = select
```

Example

```
pingPongS :: IO ()
pingPongS = whenS (fmap (=="ping") getLine) (putStrLn "pong")

whenS :: Selective f => f Bool -> f () -> f ()
whenS x y = selector <*> effect
  where
    selector :: f (Either () ())
    selector = bool (Right ()) (Left ()) <$> x

effect :: f (() -> ())
effect = const <$> y
```


What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r = fmap (fmap Left) x <*> fmap (fmap Right) l <*> r
```

More combinators

```
ifs :: Selective f => f Bool -> f a -> f a -> f a
ifs x t e = branch (bool (Right ())) (Left ()) <$> x) (const <$> t) (const <$> e)
```

```
(<||>) :: Selective f => f Bool -> f Bool -> f Bool
a <||> b = ifs a (pure True) b
```

```
(<&&>) :: Selective f => f Bool -> f Bool -> f Bool
a <&&> b = ifs a b (pure False)
```

```
anyS :: Selective f => (a -> f Bool) -> [a] -> f Bool
anyS p = foldr ((<||>) . p) (pure False)
```

```
allS :: Selective f => (a -> f Bool) -> [a] -> f Bool
allS p = foldr ((<&&>) . p) (pure True)
```

Every Monad is Selective

- ```
selectM :: Monad m => m (Either a b) -> m (a -> b) -> m b
selectM x y = x >>= \e ->
 case e of
 Left a -> ($a) <$> y
 Right b -> return b
```

# Every Monad is Selective

```
selectM :: Monad m => m (Either a b) -> m (a -> b) -> m b
selectM x y = x >>= \e ->
 case e of
 Left a -> ($a) <$> y
 Right b -> return b
```

- In fact, `select = selectM` is the *definition of the semantics* of `select` for a **Monad**.
  - (rather like `<*> = ap` defines the semantics of `Applicative` for a `Monad`)

# Every Monad is Selective

```
selectM :: Monad m => m (Either a b) -> m (a -> b) -> m b
selectM x y = x >>= \e ->
 case e of
 Left a -> ($a) <$> y
 Right b -> return b
```

- Some Monads may choose to implement **select** more efficiently
  - e.g. Haxl uses parallelism for **<\*>**, speculation for **select**

# Every Applicative is Selective

```
selectA :: Applicative f => f (Either a b) -> f (a -> b) -> f b
selectA x y = (\e f -> either f id e) <$> x <*> y
```

Always executes y

- This is a valid implementation of **select**,
  - but may not be the only one.
- Summary:
  - **select** = **selectM** → conditional effects
  - **select** = **selectA** → unconditional effects

# Data validation example

```
data Validation e a = Failure e | Success a
```

The idea is that we can traverse a structure and report multiple errors

```
instance Semigroup e => Applicative (Validation e) where
 pure = Success
 Failure e1 <*> Failure e2 = Failure (e1 <> e2)
 Failure e1 <*> Success _ = Failure e1
 Success _ <*> Failure e2 = Failure e2
 Success f <*> Success a = Success (f a)
```



# Data validation example

```
data Validation e a = Failure e | Success a

instance Semigroup e => Selective (Validation e) where
 select (Success (Right b)) _ = Success b
 select (Success (Left a)) f = ($a) <$> f
 select (Failure e _) _ = Failure e
```

Accumulates errors in  
both computations

# Data validation example

```
data Validation e a = Failure e | Success a

instance Semigroup e => Selective (Validation e) where
 select (Success (Right b)) _ = Success b
 select (Success (Left a)) f = ($a) <$> f
 select (Failure e) _ = Failure e
```

Discard errors on the right if the condition failed

- Neither selectA nor selectM
- Cannot be a Monad!

```
mkAddress
```

```
 :: Selective f
```

```
=> f Street
```

```
-> f City
```

```
-> f PostCode
```

```
-> f Country
```

```
-> f Address
```

```
mkAddress street city postcode country =
```

```
 Address
```

```
 <$> street
```

```
 <*> city
```

```
 <*> ifS (hasPostCode <$> country) (Just <$> postcode) (pure Nothing)
```

```
 <*> country
```

# Laws

- There are identity, distributive and associative laws
- Non-laws:
  - `pure (Right x) <*> y == pure x`
  - `pure (Left x) <*> y == ($x) <$> y`
  - these would rule out **Validation**, and speculation
- But: Monads must satisfy `select = selectM`

Selective and Haxl

# What is Haxl?

- Solves the following problem:
  - I want to write code that works with remote data
  - I want data-fetching to happen in parallel where possible
  - automatically, without me having to do anything
- In use at scale at Facebook for writing anti-abuse code

# Example: a blog engine

```
getPostIds :: Hax1 [PostId]
getPostContent :: PostId -> Hax1 PostContent
```

I want to fetch all the content of all the posts:

```
getAllPostsContent :: Hax1 [PostContent]
getAllPostsContent = getPostIds >>= mapM getPostContent
```

- Just use standard monadic combinators
- `mapM getPostContent` should happen in parallel

# Batching

Indeed, not just parallel, but batching multiple requests where possible:

Unbatched

```
SELECT content FROM posts
WHERE postid = id1
```

```
SELECT content FROM posts
WHERE postid = id2
```

...

Batched

```
SELECT content FROM posts
WHERE postid IN {id1, id2, ...}
```



# Implementation

This is the  
result of a  
computation

```
data Result a
 = Done a
 | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

# Implementation

```
data Result a
 = Done a
 | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Done indicates  
that we have  
finished

# Implementation

```
data Result a
 = Done a
 | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Blocked indicates that the computation requires this data.

# Implementation

```
data Result a
 = Done a
 | Blocked (Seq BlockedRequest) (Haxl a)

newtype Haxl a = Haxl { unHaxl :: IO (Result a) }
```

Haxl is in IO,  
because we  
use IORefs to  
store results

```
instance Monad Haxl where
 return a = Haxl $ return (Done a)

Haxl m >>= k = Haxl $ do
 r <- m
 case r of
 Done a -> unHaxl (k a)
 Blocked br c -> return (Blocked br (c >>= k))
```

If  $m$  blocks with continuation  $c$ , the continuation for  $m \gg= k$  is  $c \gg= k$

# Haxl works by having a special Applicative instance

```
instance Applicative Haxl where
 pure = return

Haxl f <*> Haxl x = Haxl $ do
 f' <- f
 x' <- x
 case (f',x') of
 (Done g, Done y) -> return (Done (g y))
 (Done g, Blocked br c) -> return (Blocked br (g <$> c))
 (Blocked br c, Done y) -> return (Blocked br (c <*> return y))
 (Blocked br1 c, Blocked br2 d) -> return (Blocked (br1 <> br2) (c <*> d))
```

- when we use <\*> we get parallelism
- when we use >>= we get sequentiality

# Conditionals

- We found short-cutting “and” and “or” useful:

```
(.||), (.&&) :: Hax1 Bool -> Hax1 Bool -> Hax1 Bool
```

```
a .&& b = do
 x <- a
 if x then b else return False
```

- Particularly in cases like

```
if simpleCondition .&& complexCondition then .. else ..
```

- But sometimes it's not easy to know the best ordering  
`complexCondition && otherComplexCondition`
- ... especially when the number of conditions is large, and/or changes often
- We could do it in parallel:  
`and [complexCondition, otherComplexCondition]`
- But this leaves some performance on the table:
  - if either condition returns `False` early, we don't need to finish evaluating the other one.



# Parallel boolean operators

```
pAnd, pOr :: Hax1 Bool -> Hax1 Bool -> Hax1 Bool
```

- These are semantically the same as `(.&&)`, `(.||)`
  - but evaluate both arguments in parallel
  - and bail out early if the answer is known

# Direct implementation

```
pAnd :: Haxl Bool -> Haxl Bool -> Haxl Bool
pAnd (Haxl a) (Haxl b) = Haxl $ do
 x <- a
 case x of
 Done False -> return False
 Done True -> b
 Blocked bx cx -> do
 y <- a
 case y of
 Done False -> return False
 Done True -> return x
 Blocked by cy ->
 Blocked (bx <> by) (cx `pAnd` cy)
```

- When we say this is “parallel”, what do we mean?
  - data-fetches are done in parallel where possible
  - if both sides get blocked, we do their fetches together
  - *NOT* that we do the computation in parallel

# Using Selective

```
instance Selective Haxl where
 select (Haxl x) (Haxl f) = Haxl $ do
 rx <- x
 case rx of
 Done (Right b) -> return (Done b)
 Done (Left a) -> unHaxl (($a) <$> Haxl f)
 Blocked bx c -> do
 rf <- f
 case rf of
 Done f -> unHaxl (either f id <$> c)
 Blocked by d ->
 return (Blocked (bx <> by) (select c d))
```

- Now

```
pAnd = (<&&>)
pOr = (<||>)
```

- And the rest of the selective combinators will now work in parallel.

# But there's a subtle difference...

- .. between the direct implementation of **pAnd** and **<&&>**
- **select** will always execute its first argument to completion
- whereas **pAnd** might abort the first argument if the second argument returns **False**
  - e.g. **(someFetch >>= x) `pAnd` return False**
  - should never execute x

# Select is not precisely what we want

- But **branch** can be symmetric

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

- Solution: Add **branch** as a method in **Selective**
- Instances can override **branch** if they want

# Generalisation

We have:

```
ifs :: Selective f => f Bool -> f a -> f a -> f a
```

Alternatively:

```
bindBool :: Selective f => f Bool -> (Bool -> f a) -> f a
```



# Generalisation

We have:

```
ifs :: Selective f => f Bool -> f a -> f a -> f a
```

Alternatively:

```
bindBool :: Selective f => f Bool -> (Bool -> f a) -> f a
```

Moreover:

```
bindS
 :: (Selective f, Bounded a, Enum a, Eq a)
 => f a -> (a -> f b) -> f b
```

Look familiar?

# bindS

```
bindS
```

```
:: (Selective f, Bounded a, Enum a, Eq a)
=> f a -> (a -> f b) -> f b
```

- Implementation in terms of `select` could sequentially check all the possible values of `a`
- But for a monad, `bindS = (>>=)`
  - suggests that `bindS` should be a method

# More applications

- **Build systems:**
  - extract all build dependencies before execution, with conditional execution
- **Modelling processor instructions:**
  - Categorising instructions: Functor (e.g. increment), Applicative (arithmetic), Selective (branching), Monad (indirect memory access)
- **Parsing combinators:**
  - Use Selective instead of Alternative to avoid backtracking

# Conclusions

- Selective identifies a useful point in the design space between Applicative and Monad
- Combines the benefits of Applicative (static analysis, parallelism, speculation) with limited conditional support

# What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r = select x l
```

Would make `b == c`

# What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r =
```

```
 select
```

```
 (fmap (either Left (Right . Left)) x)
```

```
 (fmap (\f -> Right . f) l)
```

q = Either b c

# What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r =
 select (
 select
 (fmap (either Left (Right . Left)) x)
 (fmap (\f -> Right . f) l)
) r
```

q = Either b c

# What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r =
 select (
 select
 (fmap (either Left (Right . Left)) x) fmap (fmap Left)
 (fmap (\f -> Right . f) l)
) r
```



# What interesting combinators can we build?

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
```

Define branch in terms of select...

```
select :: Selective f => f (Either p q) -> f (p -> q) -> f q
```

```
branch x l r =
 select (
 select
 (fmap (either Left (Right . Left)) x) fmap (fmap Left)
 (fmap (\f -> Right . f) l) fmap (fmap Right)
) r
```