Monads and all that…
III – Applicative Functors

John Hughes
Chalmers University/Quviq AB

Recall our expression parser…

```
expr = do a <- term
          exactly '+'
          b <- term
          return (a+b)
       `mplus`
       term

term = do a <- factor
          exactly '*'
          b <- factor
          return (a*b)
       `mplus`
       factor
```

```
factor = number
         `mplus`
         do exactly '('
            a <- expr
            exactly ')'
            return a

exactly t =
  satisfy (==t)
```

Wouldn't it be nice to use **`liftM3`** here?

```
liftM3 (\a _ b -> a*b) ?

liftM3x_x (*) ?
```

An Applicative Interface

- Let's *build* **`liftM3`** from simpler parts!

```
(<*>) :: Monad m => m (a -> b) -> m a -> m b
f <*> x = liftM2 ($) f x
```

- Then…

```
liftM  f x     = return f <*> x
liftM2 f x y   = return f <*> x <*> y
liftM3 f x y z = return f <*> x <*> y <*> z
…
```

left associative,
like application

Ignoring Values

- Variations on **(<*>)** that ignore one argument

```
(<*) :: Monad m => m a -> m b -> m a
a <* b = return const <*> a <*> b
```

```
(*>) :: Monad m => m a -> m b -> m b
a *> b = return (const id) <*> a <*> b
```

- All the *effects* happen left to right, but some *values* are discarded

Revisiting our expression parser…

```
expr =
   return (+) <*> term <* exactly '+' <*> term
   `mplus` term

term =
   return (*) <*> factor <* exactly '*' <*> factor
  `mplus` factor

factor =
   number
   `mplus` exactly '(' *> expr <* exactly ')'

exactly t = satisfy (==t)
```

More concise

More ”applicative” in feel

Another Problem

- Backtracking is inefficient!

```
instance MonadPlus m =>
           MonadPlus (StateT s m) where
  m `mplus` m' =
    StateT (\s ->
      runStateT m s `mplus` runStateT m' s)
```

Even if the first parser m succeeds…
…we must keep the *entire input* and the *other parser* in memory, in case we should ever need to backtrack

A Solution?

- Compute *static information* about each parser, and use to optimise
  - Possible *starter symbols*
  - Can it match the empty string?
- In **`m `mplus` m'`**, if **`m'`**
  1. Cannot match the empty string
  2. Cannot match the next symbol
  - Then it can safely be discarded

Attaching Static Information

- Let parsers be a *pair*, of
  - Static information
  - A dynamic parsing function (as before)

We can't know **`?`** until we see the (dynamic) input!

- But what about **`(>>=)`** ?
  - **`m >>= f`** matches "" ⇔ **`m`** matches "" *and* **`f ?`** matches ""
  - starters(**`m >>= f`**) = starters **`m`** ++ starters (**`f ?`**) **if** **`m`** matches ""

Hmm…

- (**>>=**) is an obstacle to computing static into
- But (**<*>**) makes (**>>=**) less necessary… can we do without (**>>=**) sometimes?

```
expr =
  return (+) <*> term <* exactly '+` <*> term
  `mplus` term
```

- Computing static info for **`f <*> m`** is unproblematic ☺

Applicative Functors

- An alternative interface…

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

- Every Monad is Applicative

```
newtype WrappedMonad m a = Wrap {unWrap :: m a}

instance Monad m => Applicative (WrappedMonad m)
where
  pure a             = Wrap (return a)
  Wrap f <*> Wrap x = Wrap (liftM2 ($) f x)
```

Not every Applicative is a Monad!

- Can a parser match the empty string?

```
newtype Empty a = Empty Bool

instance Applicative Empty where
  pure _                = Empty True
  Empty f <*> Empty x = Empty (f && x)
```

  - A ”parser” that can’t parse—just tell us if it matches ””!
- A generally useful kind of non-monadic Applicative: collect information using a monoid

But every Applicative is a Functor

- We can always define **`fmap`** like this…

```
fmap :: (a->b) -> f a -> f b
fmap f a = pure f <*> a
```

- (We can't write a *general* instance, because the type-checker would use it too often, but for any *specific* **`f`** the definition works)

```
instance Applicative f => Functor f
where
   fmap f a = pure f <*> a
```

Applicative vs Monad

- Consider a *conditional* function

```
cond :: m Bool -> m a -> m
```

Effects depend on *value* of **`m`**

- Monadic:

```
cond m f g = do bool <- m
                if bool then f else g
```

All effects happen anyway

- Applicative:

```
cond m f g =
  pure (\b t e->if b then t
    <*> m <*> f <*> g
```

OK for parsing CFGs!

Applicatives are more composable!

- We can *pair* any two Applicatives:

```
data Prod f g a = Prod (f a) (g a)

instance (Applicative f, Applicative g) =>
                       Applicative (Prod f g) where
  pure x = Prod (pure x) (pure x)
  Prod f g <*> ~(Prod x y) =
    Prod (f <*> x) (g <*> y)
```

Applicatives are more composable!

- We can *compose* Applicatives:

```
newtype Compose f g a = Comp (f (g a))

instance (Applicative f, Applicative g) =>
                    Applicative (Compose f g) where
  pure x = Comp (pure (pure x))
  Comp f <*> Comp x = Comp (pure (<*>) <*> f <*> x)
```

- Even monads which *don’t* compose can be wrapped and composed as Applicatives!

Making Choices

- We need an analogue of **`MonadPlus`**

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

- Of course, wrapping a MonadPlus gives an Alternative

```
instance MonadPlus m =>
           Alternative (WrappedMonad m) where
  empty = Wrap mzero
  Wrap a <|> Wrap b = Wrap (a `mplus` b)
```

Making Empty an Alternative

- Can we define an Alternative instance for Empty?
  - When does a choice betw… rsers match …?

Matches no strings, so definitely not the empty string

```
instance Alternative Empty where
  empty = Empty False
  Empty f <|> Empty g = Empty (f || g)
```

Compare <*>, which used &&

Some and Many revisited ☺

- Now we can define **some** and **many** for *any* Alternative functor!

```
some f = s where
  s = (:) <$> f <*> (s <|> pure[])
```

- Even generic *optional* values!

```
optional f =      Just <$> f
        <|> pure Nothing
```

watch that laziness!

Where are we now?

- Wrapping our Parser monad gives us an Alternative functor
  - With pure, <*>, empty, <|>, <*, *>, some, many…
  - Almost everything we need to write parsers!

```
newtype Monadic a =
    Monadic (WrappedMonad (StateT String Maybe) a)
  deriving (Functor, Applicative, Alternative)
```

- We just need to add **`exactly`**

The Parser Class

- Because we want *multiple* representations of parsers, define a class

```
class Alternative p => Parser p where
  exactly :: Char -> p Char
```

- Monadic implementation:

```
instance Parser Monadic where
  exactly t = Monadic (WrapMonad (do
    ts <- get
    case ts of
      [] -> mzero
      t':ts' -> do
        guard (t==t')
        put ts'
        return t))
```

Our Example, Applicatively

```
number, expr, term, factor ::
  Parser p => p Integer

number = read <$> some (anyof ['0'..'9'])

expr = (+) <$> term <* exactly '+' <*> term
   <|> term

term = (*) <$> factor <* exactly '*' <*> factor
   <|> factor

factor = number
     <|> exactly '(' *> expr <* exactly ')'
```

```
*Parser> runMonadic expr "1+2*3"
Just (7,"")
```

defined using exactly and <|>

Empty Parser

- Can **`exactly t`** match the empty string?

```
instance Parser Empty where
  exactly _ = Empty False
```

Examples

```
*Parser> runEmpty expr
False
*Parser> runEmpty (many expr)
True
```

We can *execute* and *analyse* the same code

What tokens can a parse start with?

```
newtype Starts a = Starts [Char]

instance Functor Starts where
  fmap f x = pure f <*> x

instance Applicative Starts where
  pure x = Starts []


instance Alternative Starts where
  empty = Starts []
  Starts ts <|> Starts ts' = Starts (nub (ts++ts'))

instance Parser Starts where
  exactly t = Starts [t]
```

Of course this doesn't work…

```
*Parser> runStarts (exactly 'x' <|> exactly 'y')
"xy"

*Parser> runStarts (some (exactly 'x'))
"*** Exception: No instance nor default method for
class operation Control.Applicative.<*>
```

- As soon as we use something needing **`<*>`**, we crash

Let’s compute Empty and Starts together

- Just form their *product*

```
newtype Static a = Static (Prod Starts Empty a)
  deriving (Functor,Applicative,Alternative,Parser)
```

- We’ll need to make **`Prod`** a **`Parser`**

```
instance (Parser f, Parser g) => Parser (Prod f g)
where
  exactly t = Prod (exactly t) (exactly t)
```

- Of course, it still doesn’t work!

```
*Parser> runStatic (exactly 'x' <|> exactly 'y')
("xy",False)
*Parser> runStatic (some (exactly 'x'))
("*** Exception: No instance nor default method for
class operation Control.Applicative.<*>
```

Replace <*> just for Static!

- Derive everything except **`Applicative`**

```
newtype Static a = Static (Prod Starts Empty a)
  deriving (Functor,Alternative,Parser)
```

```
instance Applicative Static where
  pure x = Static (pure x)

  Static (Prod (Starts ts) (Empty e)) <*>
    ~(Static (Prod (Starts ts') (Empty e')))
    = Static (Prod (Starts (ts++if e then ts' else []))
                   (Empty e<*>Empty e'))
```

Now it works!

- **Examples:**

```
*Parser> runStatic (some (exactly ' ') *> exactly 'x')
(" ",False)

*Parser> runStatic (many (exactly ' ') *> exactly 'x')
(" x",False)

*Parser> runStatic expr
("0123456789(",False)
```

(Truth in Advertising)

- It should work, but it doesn’t
- I have to explicitly declare the Alternative instance too, and work around a bug in ghc’s strictness analyser (?)

Optimizing <|>

- Choice is inefficient in backtracking parsers
- Let's *pair* the Static and Monadic parsers

```
newtype OptParser a = Opt (Prod Static Monadic a)
  deriving (Functor, Applicative, Parser)
```

- Define an Alternative instance that optimizes <|> based on the starter tokens and the next character
- Could not be done with monads

What else can we do?

- Let's try Applicative randomness!

```
newtype Random a = Random (WrappedMonad RandomM a)
  deriving (Functor, Applicative, Choice)
```

- We need a class for choose

```
class Applicative f => Choice f where
  choose :: Int -> Int -> f Int
```

```
instance Choice (WrappedMonad RandomM) where
  choose m n | m <= n =
    WrapMonad (do x <- generate
                  return (m + (x `mod` (n-m+1))))
```

Random Alternatives

- We make the choice *in the monad* to avoid generating both alternatives always

```
instance Alternative Random where
  Random (WrapMonad m) <|> Random (WrapMonad m') =
    Random (WrapMonad (do
      x <- generate
      if even x then m else m'))
```

BUT

- No sensible definition of empty

Bounded lists

- Bounded lists are easy to define with <|>:

```
blist 0 g           = pure []
blist n g | n > 0 =
    shorter <|>
    (:) <$> g <*> shorter
  where shorter = blist (n-1) g
```

```
*Random> runRandom (blist 30 (choose 1 10 <|> pure 33))
[33,5,33,5,33,33,9,4,33,7,3,33,1,10]
```

- But do we really want 33 so often?

Cardinality

- How many possibilities are we choosing from?

```
newtype Card a = Card {runCard :: Integer}
```

```
instance Applicative Card where
  pure _ = Card 1
  Card m <*> Card n = Card (m*n)

instance Alternative Card where
  empty = Card 0
  Card m <|> Card n = Card (m+n)

instance Choice Card where
  choose m n = Card (fromIntegral $ n-m+1)
```

Use Cardinality to Guide Choice

- Compose Card and Random into a product

```
newtype Uniform a =
    Uniform (Prod Card (WrappedMonad RandomM) a)
  deriving (Functor, Applicative, Choice)
```

```
instance (Choice f, Choice g) =>
                Choice (Prod f g) where
  choose m n = Prod (choose m n) (choose m n)
```

- Define Alternative Uniform to use cardinalities as weights!

```
empty = Uniform (Prod empty undefined)
```

That's Better!

- Here's the old test

```
*Random> runRandom (blist 30 (choose 1 10 <|> pure 33))
[33,5,33,5,33,33,9,4,33,7,3,33,1,10]
```

  - Lots of 33s!

- Here's the new one

```
*Random> runUniform (blist 30 (choose 1 10 <|> pure 33))
[5,33,2,6,7,3,3,7,10,7,1,10,4,10,9,4,3,6,4,6,10,3,33,5,3,
33,9,1,4]
```

What else can we do?

- ZipLists!
  - `[f,g,h] <*> [x,y,z]` → `[f x,g y,h z]`
- Think of a *sequence of steps*

- Lists are already Applicative (all combinations), so we need a new type

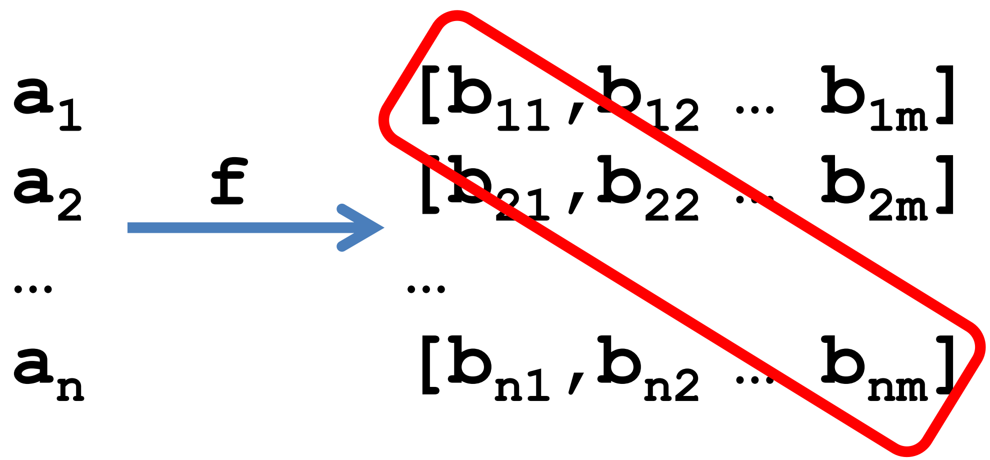
Applicative ZipLists

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs =
    ZipList (zipWith ($) fs xs)
```

- It makes sense that pure repeats x infinitely… it's available at every step

A ZipList Monad?

- Consider `[a1,a2…an] >>= f` ($[a_1, a_2 \ldots a_n]$ >>= f)



- The 3rd monad law fails
  - (if f returns lists of different lengths)
  - Would also be very inefficient

Functional Reactive Programming

- Describes changing behaviours over time
  - Behaviour a   Time -> a
- Naturally applicative!
  - Behaviour (a->b) -> Behaviour a -> Behaviour b
- Inefficient as a monad!
  - Behaviour a -> (a -> Behaviour b) -> Behaviour b

Terrible for GC!

Construct a Behaviour b from a n at each Time, then sample it at one point!

Html (nano-)Formlets

- Example:

Name: John Hughes
Age: 54
Gender: male

- Generated by:

```
Name:   <input type="text" name="name"> <br>
Age:    <input type="text" name="age">  <br>
Gender: <input type="text" name="gender">
```

- Data returns to the application as

```
[("name","John Hughes"),
 ("age","54"),
 ("gender","male")]
```

Names must be unique

Names must match

Using Formlets

```
data Person = Person String Integer Gender
  deriving Show
data Gender = Male | Female
  deriving (Read, Show)
```

```
person =
  Person
    <$  html "Name: "
    <*> input
    <*  html "<br>\nAge: "
    <*> (read <$> input)
    <*  html "<br>\nGender: "
    <*> (read <$> input)
```

Generate HTML

Accept and process input

The features we need

- Generation of unique names
- Collection of generated HTML
- Evaluation of results given field values

in this order!

```
newtype Formlet a =
    Formlet (Compose NameGen (Compose Html Eval) a)
  deriving (Functor, Applicative)
```

`NameGen (Html (Eval a))`

Staged effects

Name Generation

- We use a state monad to carry a counter

```
newtype NameGen a =
    NameGen (WrappedMonad (State Integer) a)
  deriving (Functor, Applicative)
```

- Generate a name by incrementing it

```
nextName :: NameGen String
nextName = NameGen (WrapMonad (do
  n <- get
  put (n+1)
  return ("input_"++show n)))
```

Collecting Html

pairing with a monoid gives an Applicative

- Collect a string of HTML as the effect

```
newtype Html a = Html (String,a)
  deriving (Functor, Applicative)
```

- Basic operation generates some text

```
text :: String -> Html ()
text s = Html (s,())
```

- Generating a named input field

```
inputField name =
  text $ "<input type=\"text\"name=\""++name++"\">"
```

Evaluation of fields

adding an argument gives an Applicative

- Pass in list of fields implicitly

```
newtype Eval a = Eval ([(String,String)] -> a)
  deriving (Functor, Applicative)
```

- An operation to look up the value of a named field

```
field :: String -> Eval String
field name = Eval (fromJust . lookup name)
```

Formlets: Generating HTML

```
newtype Formlet a =
    Formlet (Compose NameGen (Compose Html Eval) a)
  deriving (Functor, Applicative)
```

```
html :: String -> Formlet ()
html s = Formlet (Comp (pure (Comp (pure <$> text s))))
```

```
                                              Html ()
                                      Html (Eval ())
                                Compose Html Eval ()
                      NameGen (Compose Html Eval ())
              Compose NameGen (Compose Html Eval) ()
```

Formlets: Input Fields

- Combine effects in all three Applicatives!

```
input :: Formlet String
input = Formlet (Comp (
   (\name -> Comp ((pure (field name))
                   <*
                   inputField name)
                  )
   <$> nextName
   ))
```

- Key: **NameGen** **Html** **Eval**

Running it…

- Run the person Formlet…

```
*Formlet> let (output,fun) = runFormlet person
```

- Print the HTML

```
*Formlet> putStrLn output
Name: <input type="text" name="input_1"><br>
Age: <input type="text" name="input_2"><br>
Gender: <input type="text" name="input_3">
```

- Evaluate on corresponding inputs

```
*Formlet> runEval fun [("input_1","John Hughes"),
("input_2","54"), ("input_3","Male")]
Person "John Hughes" 54 Male
```

Conclusions

- Applicative functors are…
  - Less *powerful* than monads—less expressive
  - More *general* than monads—more instances
- More *composable* than monads
  - Prod and Compose
  - No need for ”Applicative transformers”
- Enjoy a simple interface—a ”sweet spot” in common interfaces
- Have lots of applications