Extending Monads with Pattern Matching

Tomas Petricek
with Alan Mycroft and Don Syme

Special thanks to Phil Wadler and Dominic Orchard

Introduction

- Warm Fuzzy Things
  - Sequential composition
- There is more
  - Parallelism, concurrency
  - Additional operations

And more!

```
multiply :: Par Int -> Par Int -> Par int
multiply pa pb = do
    tok <- newCancelToken
    r <- forall' tok tree
    leftRes <- new
    rightRes <- new
    finalRes <- newBlocking
    forkWith tok (pa >>=
        completed leftRes rightRes finalRes)
    forkWith tok (pb >>=
        completed rightRes leftRes finalRes)
    r <- get finalRes
    cancel tok
    return r

  where
    completed varA varB fin resA = do
      put varA resA
      ( if not resA then put fin False
        else get varB >>=
            put fin . (&& resA) )
```

The Problem

- Practical monads have additional operations

```
spawn :: Par a -> Par (IVar a)
get   :: Ivar a -> Par a
```

  - Library-specific types
  - Library-specific names

- Are there common operations?
- Is there a nice notation?

docase notation

Get a GHC patch from https://github.com/tpetricek

Multiplying **Par** values

- **Par** values

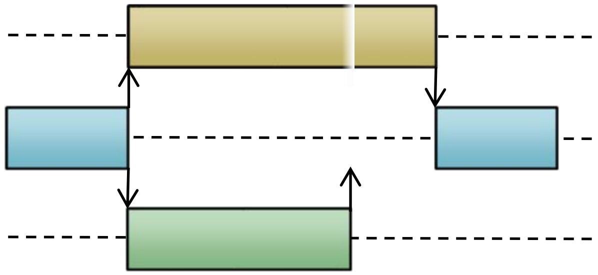
  Compute result
  in background

- Pattern matching
  - "a" waits for a value

Suspended computation

Run both

```
multiply f1 f2 =
  docase f1, f2 of
    a, b -> return $ a*b
```



Multiplying Par values

- **Par** values

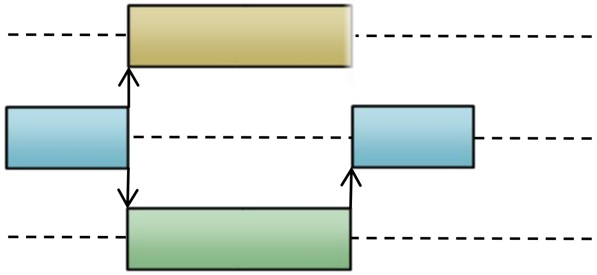
  Compute result
  in background

- Pattern matching
  - "a" waits for a value
  - "?" does not need
    a value to match
  - "o" waits for a
    specific value

Run both

```
multiply f1 f2 =
  docase f1, f2 of
    0, ? -> return 0
    ?, 0 -> return 0
    a, b -> return $ a*b
```

Choice



Multiplying **Par** values

- **Par** values

  Compute result
  in background

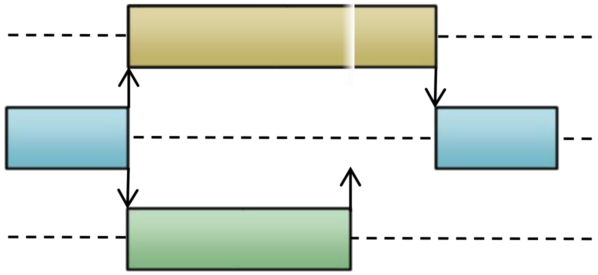
- Pattern matching
  - “a” waits for a value
  - “?” does not need
    a value to match
  - “o” waits for a
    specific value

Run both

```
multiply f1 f2 =
  docase f1, f2 of
    0, ? -> return 0
    ?, 0 -> return 0
    a, b -  return $ a*b
```

Choice

Fail



Joinad type classes

Monad with an additional near-semiring structure!

Three additional operations

- **MonadZip** (parallel composition)

```
mzip     :: m a -> m b -> m (a, b)
```

- **MonadOr** (monadic choice)

```
morelse :: m a -> m a -> m a
mzero   :: m a
```

- **MonadAlias** (aliasing of computations)

```
malias  :: m a -> m (m a)
```

Joinad Laws

- Intuition – **docase** is like **case**
  - Laws guarantee that **docase** equations hold
  - Implication is one way only (future work!)
- Nice algebraic structure

$\otimes$ means **mzip**, $\oplus$ means **morelse**, 0 means **mzero**

$$
\begin{gathered}
a \otimes 0 = 0 \\
a \oplus 0 = a \\
a \otimes b = b \otimes a \\
a \otimes (b \otimes c) = (a \otimes b) \otimes c \\
a \oplus (b \oplus c) = (a \oplus b) \oplus c \\
a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)
\end{gathered}
$$

More examples!

STM, Parsers, Communicating Haskell Processes, Orc monad

Summary

- Joinads capture common pattern

  Parallelism, concurrency, parsing, STM

  Always looking for more examples!

- **docase** notation is useful

  Comment on Hackage **Trac** and **glasgow -haskell-users**

- For more information
  - Info: http://tomasp.net/blog/docase-haskell.aspx
  - Now also GHC patch: https://github.com/tpetricek

Backup

Translation

```
docase ma, mb of
  0, ? -> return 0
  a, b -> return $ a*b
```

```
malias ma >>= \ma ->
malias mb >>= \mb ->
  ( (ma >>= \arg -> case arg of
       0 -> return (return 0)
       otherwise -> mzero) `morelse`
     (ma `mzip` mb >>= \arg -> case arg of
       (a, b) -> return (return $ a*b))) ) >>= id
```

Alternative type class

- More common structure capturing similar idea
  - Parallel composition and choice
  - Direct correspondence to **docase** syntax

```
class Functor f => Monoidal f where
  unit :: f ()
  (⋆)  :: f a -> f b -> f (a, b)

class Monoidal f => Alternative f where
  emtpy :: f a
  (◊)   :: f a -> f a -> f a
```

- How to keep additional (useful) features?

MonadAlias structure

- Two simple implementations
  - Run the effect later, when the computation is used

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

  - Run the effect now, then pass just a pure value

```
malias :: m a -> m (m a)
malias = liftM return
```

- Is it useful more generally? (we think so!)
- Is there nice formal background? (comonads!)

Parsing using **docase**

- Validating Cambridge phone numbers
  - Contain only digits
  - Consists of 10 characters
  - Start with a prefix “1223”

```
valid = docase many (satisfies isDigit),
               multiple 10 character,
               startsWith (string "1223")
        of str, _, _ -> return str
```

- MonadZip is *intersection* of languages
  - Returns results of all three parsers

Printing buffer using joins

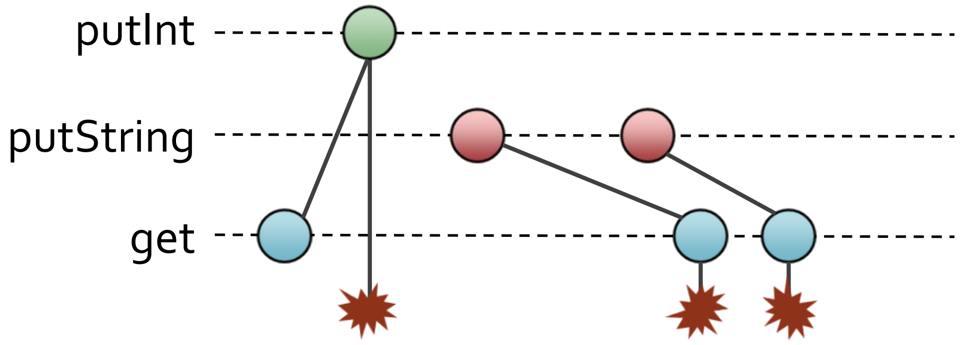
- Join calculus
  - Channels store values
  - Joins specify reactions
- Pattern matching
  - Use clauses to encode joins

```
buffer =
  docase get, putInt, putString of
    r, n, ? -> do
       reply r (intToString n)
    r, ?, s -> do
       reply r s
```

First clause

Second clause

Second clause


putInt
putString
get