

# Mathematically Structuring Programming Languages

*Cambridge Programming Research Group Lectures 2010*  
*Dominic Orchard*

# Plan

- Introduce a simple arithmetic language
- Programs as semantics & interpreter
- Use a *monad* structure to abstract/simplify
- Warm up: a simple mathematical structure

# Warm up... Monoids

A simple mathematical structure



Not the one-eyed aliens from 1960s 'Dr. Who'

# Monoids

$(S, \bullet, e)$

where

$$\begin{aligned}\bullet &: S \times S \rightarrow S \\ e &: S\end{aligned}$$

and

$$\forall x . x \bullet e = x = e \bullet x$$

**identity**

$$\forall x y z . x \bullet (y \bullet z) = (x \bullet y) \bullet z$$

**associativity**

e.g.

$$(\mathbb{N}, +, 0)$$

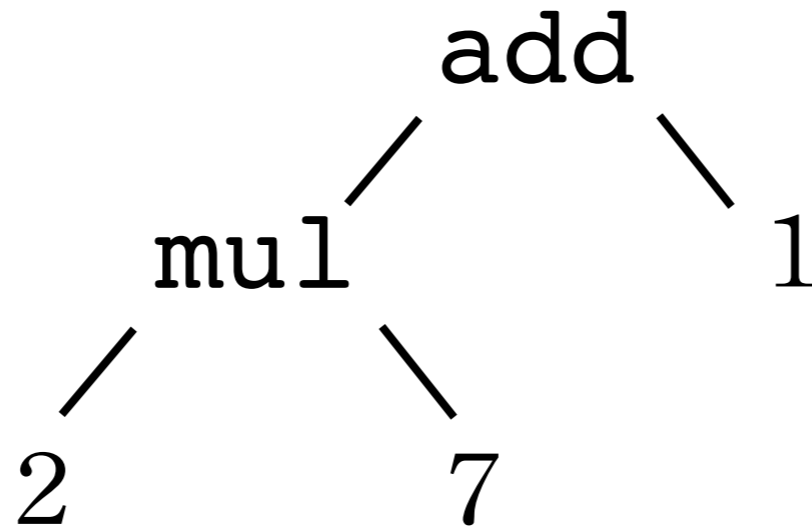
$$(\mathbb{N}, *, 1)$$

$$([a], ++, [])$$

$$\del{(\mathbb{N}, -, 0)}$$

# Simple Arithmetic Language $E$

$E := \text{add } E_1 E_2 \mid \text{sub } E_1 E_2 \mid \text{div } E_1 E_2 \mid \text{mul } E_1 E_2 \mid \mathbb{Z}$



# Semantics of $E$

$$[-] : E \rightarrow \mathbb{Z}$$

$$[\text{add } e1 \ e2] = [e1] + [e2]$$

$$[\text{sub } e1 \ e2] = [e1] - [e2]$$

$$[\text{div } e1 \ e2] = \frac{[e1]}{[e2]} \longrightarrow \frac{[e1]}{0} = ?$$

$$[\text{mul } e1 \ e2] = [e1] * [e2]$$

$$[n] = n$$

# Shallow EDSL in Haskell

- Haskell as meta language
- Terminals in grammar = functions
- Semantic definition + interpreter for **free**

```
add x y = x + y
sub x y = x - y
div x y = x `Prelude.div` y
mul x y = x * y
```

# The types ensure well-formed programs

```
add :: Int -> Int -> Int
add x y = x + y
```

```
sub :: Int -> Int -> Int
sub x y = x - y
```

```
div :: Int -> Int -> Int
div x y = x `Prelude.div` y
```

```
mul :: Int -> Int -> Int
mul x y = x * y
```

E.g.

```
mul (add 3 4) 6
```

⇓

```
42
```

```
add (mul 4) 5
```

⇓

Couldn't match expected type 'Int'  
against inferred type 'Int -> Int'

# Adding Exceptions (I)

```
div :: Int -> Int -> Int
div x y = x `Prelude.div` y
```

```
div 1 0
```



**\*\*\* Exception: divide by zero**

# Adding Exceptions (2)

- Return *either* an *Int* **or** an exception string:  
*Either Int String*
- **Sum type:** *Either a b*  $\cong$   $a + b$

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

*Left :: a → Either a b*

*Right :: b → Either a b*

# Adding Exceptions (3)

```
div :: Int -> Int -> Int  
div x y = x 'Prelude.div' y
```



```
div :: Int -> Int -> Either Int String  
div (x, y) = if y==0 then Right "Divide by zero"  
             else Left (x 'Prelude.div' y)
```

# Adding Exceptions (4)

```
add :: Int -> Int -> Int  
add x y = x + y
```



```
add :: Int -> Int -> Either Int String  
add x y = Left (x + y)
```

# Now the types don't match

```
add 1 (div 1 0)
```



Couldn't match expected type 'Int'  
against inferred type 'Either Int String'  
In the second argument of 'add', namely '(div 1 0)'

# Extending each operation to pass exceptions through is bad

- Tedious
- Complicates semantics/code

```
add :: Either Int String -> Either Int String -> Either Int String
add x y = case x of
  Right e -> Right e
  Left x' -> case y of
    Right e' -> Right e'
    Left y' -> Left (x' + y')
```

- x4 (sub, div, and mul too)

# Question

- Can we write a higher-order function to abstract the checking of exceptions in the parameters to `add`, `sub`, `mul`, `div`?

```
add :: Either Int String -> Either Int String -> Either Int String
add x y = case x of
  Right e -> Right e
  Left x' -> case y of
    Right e' -> Right e'
    Left y'  -> Left (x' + y')
```

# Possible Solution

```
handle :: (a -> a -> Either String a) -> Either String a
        -> Either String a
        -> Either String a
handle f x y = case x of
    Right e -> Right e
    Left x' -> case y of
        Right e' -> Right e'
        Left y'  -> f x' y'
```

# Monads

- *Monads* provide machinery for composing functions which return extra structure

$$f : a \rightarrow b$$

$$f' : a \rightarrow M b$$

e.g.

$$f : a \rightarrow M b$$

$$g : b \rightarrow M c$$

$$g \circ f : a \rightarrow Mb$$

~~$$b \rightarrow Mc$$~~

# Monads

$(M, \eta, \mu)$

where

- $M$  is a *functor*

like a parameterised data type e.g.

data M a = ...

with

$$M_{map} : (a \rightarrow b) \rightarrow M a \rightarrow M b$$

- and...

$$\eta : a \rightarrow M a$$

like an identity element

$$\mu : M(M a) \rightarrow M a$$

like the monoid's • operation

# Monads as “boxes”

$$M_{map} : (a \rightarrow b) \rightarrow M a \rightarrow M b$$

Change contents, but not box

$$\eta : a \rightarrow M a \quad \text{(note: } \forall a \text{)}$$

Put something into a (simple) box

$$\mu : M(M a) \rightarrow M a \quad \text{(note: } \forall a \text{)}$$

Fold nested structures together into one

# Monads

- Going to use monad operations for composing functions that return  $M$  structures:

$$f : a \rightarrow M b \qquad g : b \rightarrow M c$$

$$M_{map} g : M b \rightarrow M(M c)$$

$$\mu \circ (M_{map} g) : M b \rightarrow M c$$

$$\mu \circ (M_{map} g) \circ f : a \rightarrow M c$$

“**Extension**” form:

$$extend : (a \rightarrow M b) \rightarrow M a \rightarrow M b$$

$$extend f = \mu \circ (M_{map} f)$$

# Monads in FP

- A structure (data type):  $\text{data } M a = \dots$

- Inject values into the monad ( $\eta$ )

$$\text{unit} :: a \rightarrow M a$$

- Lift functions to work on the monad

$$\text{extend} :: (a \rightarrow M b) \rightarrow M a \rightarrow M b$$

- We will implement *extend* directly, not using  $\mu$

# Exception Monad

- **Structure:** *Either a String*  $\cong$   $a + \text{String}$

- **Unit operation:**

*unit* :  $a \rightarrow \text{Either a String}$

*unit* = *Left*

# Exception Monad (2)

- Extend operation:

$$\text{extend} : (a \rightarrow \text{Either } b \text{ String}) \rightarrow \text{Either } a \text{ String}$$
$$\rightarrow \text{Either } b \text{ String}$$
$$\text{extend } f \text{ (Left } x) = f x$$
$$\text{extend } f \text{ (Right } e) = \text{Right } e$$

Handles the exception in the first argument of  $f$

# Intuition of *extend*

$$\textit{extend} :: (a \rightarrow M b) \rightarrow M a \rightarrow M b$$

- *extend* deconstructs a monad
- ... but it doesn't really. When derived, uses `Mmap`, lifting a function to `Ma -> M(M b)` whose result is combined back into `M b`
- combines incoming info. with outgoing
- no need to duplicate code for handling effects in parameters

# *E* with monadic exceptions

```
add :: Int -> Int -> Either Int String
add x y = unit (x + y)
```

```
sub :: Int -> Int -> Either Int String
sub x y = unit (x - y)
```

```
mul :: Int -> Int -> Either Int String
mul x y = unit (x * y)
```

```
div :: Int -> Int -> Either Int String
div x y = if y==0 then Right "Divide by zero"
           else unit (x 'Prelude.div' y)
```

# $E$ with monadic exceptions

## Exercise

$extend (\lambda a. extend (\lambda b. mul\ a\ b) (add\ 2\ 5)) (add\ 3\ 4)$

=

?

# ... however

*extend* complicates expressions in  $E$

## Compare

*extend* ( $\lambda a . \textit{extend} ( $\lambda b . \textit{mul}  $a$   $b$ ) (*add* 2 4)) (*add* 4 3)$$

with

*mul* (*add* 2 4) (*add* 3 4)

# Haskell can make *extend* implicit with the *do* notation

*extend* ( $\lambda a . \text{extend} (\lambda b . \text{mul } a \ b) (\text{add } 2 \ 4)) (\text{add } 4 \ 3)$



*do*  $a \leftarrow \text{add } 4 \ 3$   
 $b \leftarrow \text{add } 2 \ 4$   
 $\text{mul } a \ b$

**Rule:**

$\text{do } y \leftarrow f \ x \quad \equiv \quad \text{extend} (\lambda y . \dots) (f \ x)$   
 $\dots$

# Haskell *do* notation is parameterised by a monad

- “Overloading” via type classes
- A type class specifies a set of methods
- Types are made instances of a class by defining all of the methods in the class

# User-defined Monads as an instance of a type class

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

```
instance Monad (Either a String) where
  return x = Left x
  (Right e) >>= f = Right e
  (Left x) >>= f = f x
```

# User-defined Monads as an instance of a type class

- *unit* = `return`
- *extend* = `>>=` (with parameters flipped)

$extend :: (a \rightarrow M b) \rightarrow M a \rightarrow M b$

$>>= :: M a \rightarrow (a \rightarrow M b) \rightarrow M b$

# Correct monad instance for *do* notation is chosen based on the types

$\vdash \text{Int} \rightarrow \text{Int} \rightarrow \text{Either Int String}$

*do*  $a \leftarrow \text{add } 4 \ 3$   
 $b \leftarrow \text{add } 2 \ 4$   
 $\text{mul } a \ b$

Selects our *Either* instance of the monad for the *do* notation

# Other Monads: State, IO

```
do  putStrLn "Enter a number:"  
    x <- getLine  
    double_x <- return ((read x) * 2)  
    putStrLn ("Double your number = "  
              ++ (show double_x))
```

- Sequences input/output effects

*putStrLn* :: *String* → IO ()

*getLine* :: IO *String*

- IO is abstract so cannot be deconstructed, only by *extend* of IO monad

# Conclusion (I)

- Monads can be used to structure effectful computations: exceptions, state, non-determinism, I/O

$$(M, \eta, \mu) \quad \eta : a \rightarrow M a$$

$$\mu : M(M a) \rightarrow M a$$

- Extension form useful for functions
- FP interpretation: data type + 2 operations

$$\textit{unit} :: a \rightarrow M a$$

$$\textit{extend} :: (a \rightarrow M b) \rightarrow M a \rightarrow M b$$

- sum-type (Either) monad describes exceptions

# Conclusion (2)

- Haskell as a meta language, describing another language
- Executable semantics
- Monads used to simplify semantics considerably
- Can apply monads for any kind of computational effect



# Monads are monoids!

- Monads are monoids in a different category to where we usually use them (monoids in the “Set” category)
- Identity and associativity? (see course notes)

$$\forall x . x \bullet e = x = e \bullet x \quad \text{identity (monoid)}$$

$$\forall x y z . x \bullet (y \bullet z) = (x \bullet y) \bullet z \quad \text{associativity (monoid)}$$

$$\mu \circ (M\eta x) = x = \mu \circ (\eta_M x) \quad \text{identity (monad)}$$

$$\mu \circ M\mu = \mu \circ \mu_M \quad \text{associativity (monad)}$$