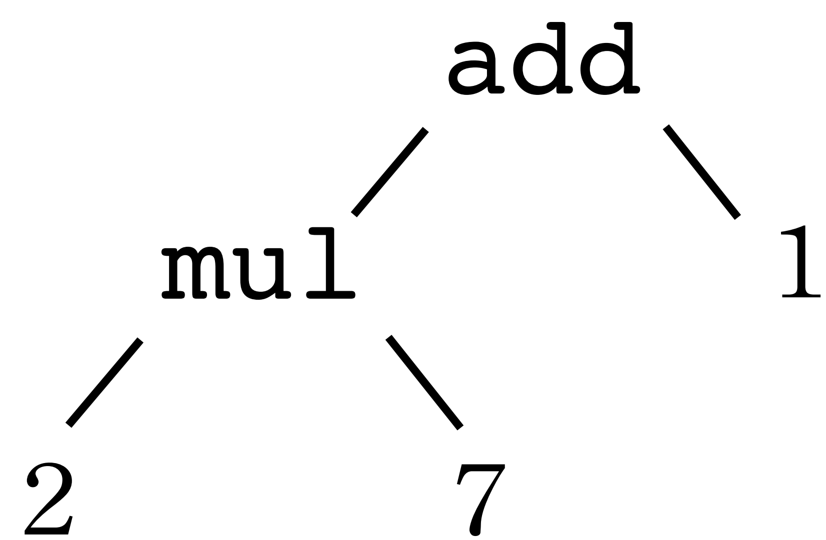
Mathematically Structuring Programming Languages

Cambridge Programming Research Group Lectures 2010

Dominic Orchard

Simple Arithmetic Language *E*

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



Semantics of *E*

$$[\![\_]\!] : E \to \mathbb{Z}$$

$$[\![\texttt{add e1 e2}]\!] = [\![e1]\!] + [\![e2]\!]$$

$$[\![\texttt{sub e1 e2}]\!] = [\![e1]\!] - [\![e2]\!]$$

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

$$[\![\texttt{mul e1 e2}]\!] = [\![e1]\!] * [\![e2]\!]$$

$$[\![n]\!] = n$$

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 guarantee 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
```

$\Downarrow$

```
42
```

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

$\Downarrow$

Couldn’t match expected type ‘Int’ against inferred type ‘Int -> Int’

Adding Exceptions (1)

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

```
div 1 0
```

$$\Downarrow$$

*** Exception: divide by zero

Adding Exceptions (2)

- We want to return *either* an Int or an exception string: $Either\ Int\ String$
- Sum type: $Either\ a\ b \cong a + b$

$$Left :: a \to Either\ a\ b$$

$$Right :: b \to Either\ a\ b$$

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)
```

Now the types don't match

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

$\Downarrow$

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)

Exercise

- 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'
```

Exceptions are a kind of *effect*

$f : a \to b$

Function

$f' : a \to M\ b$

Function + effect M

- Many kinds of effect: state, input/output, errors, exception, traces, non-determinism
- *Monads*, from category theory, can be used to structure effects
- Allow us to compose functions with effects

Warm up... Monoids

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

Warm up... Monoids

$$(S, \bullet, e)$$

where

$$\bullet : S \times S \to S$$
$$e : S$$

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.

$$(\mathbb{N}, +, 0) \quad (\mathbb{N}, *, 1) \quad ([a], ++, []) \quad \cancel{(\mathbb{N}, -, 0)}$$

Monads

(M, η, μ)

where (M is a functor):

$M : C \to C$

like a data type e.g.

```
data M a = ...
```

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

and (monad part):

$\eta : a \to Ma$ like an identity element

$\mu : M(Ma) \to Ma$ like the monoid's • operation

Monads

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

$$\forall x \,.\, x \bullet e = x = e \bullet x$$ identity (monoid)

$$\forall x\, y\, z \,.\, x \bullet (y \bullet z) = (x \bullet y) \bullet z$$ associativity (monoid)

$$\mu \circ (M\eta\ x) = x = \mu \circ (\eta_M\ x)$$ identity (monad)

$$\mu \circ M\mu = \mu \circ \mu_M$$ associativity (monad)

Monads

- Going to use monad operations for composing functions that return monads:

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

$$M_{map}\ g : Mb \rightarrow M(Mc)$$

$$\mu \circ (M_{map}\ g) : Mb \rightarrow Mc$$

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

$$extend : (a \rightarrow Mb) \rightarrow Ma \rightarrow Mb$$

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

Monads in FP

- A structure (data type): data $M\, a = \ldots$

  - Inject values into the monad (null effect) (η)

    $$unit :: a \to M\ a$$

  - Lift functions to work on the monad

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

- We will implement *extend* directly, not using μ

Exception Monad

- Structure: $Either\ a\ String \cong a + String$

- Unit operation:

$$unit : a \rightarrow Either\ a\ String$$
$$unit = Left$$

Exception Monad (2)

- Extend operation:

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

Handles the exception in the first argument of f

Intuition of *extend*

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

- *extend* deconstructs a monad for us
- but it doesn't really when derived it uses Mmap, lifting a function to Ma -> M(M b) which is combined back into M b
- combines incoming effects 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

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

=

?

49

... however

extend complicates expressions in *E*

Compare

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

with

$$mul\ (add\ 2\ 4)\ (add\ 3\ 4)$$

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

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

$$\downarrow$$

$$\begin{aligned} do \quad & a \leftarrow add\,4\,3 \\ & b \leftarrow add\,2\,4 \\ & mul\,a\,b \end{aligned}$$

Rule:

$$\begin{aligned} do \quad & y \leftarrow f\ x \\ & \ldots \end{aligned} \qquad \equiv \qquad extend\,(\lambda y\ .\ \ldots)\ (f\ x)$$

Haskell *do* notation is parameterised by a monad

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

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

$$\texttt{>>=} :: M\ a \to (a \to M\ b) \to M\ b$$

User-defined Monads as an instance of a type class

- 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
```

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

$\vdash Int \rightarrow Int \rightarrow Either\ Int\ String$

$$
\begin{aligned}
do \quad & a \leftarrow add\ 4\ 3 \\
& b \leftarrow add\ 2\ 4 \\
& mul\ a\ b
\end{aligned}
$$

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

Other Monads: State, IO

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

- Sequences input/output effects

$$putStr :: String \rightarrow \texttt{IO}\ ()$$

$$getLine :: \texttt{IO}\ String$$

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

Conclusion (1)

- Monads can be used to structure effectful computations

$$(M, \eta, \mu) \qquad \eta : a \rightarrow Ma$$
$$\mu : M(Ma) \rightarrow Ma$$

- FP interpretation: data type + 2 operations

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

- 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