

<https://drive.google.com/drive/folders/1oju5XL5sAlKe4n6T6dXfYpU1a1meVt6t?usp=sharing>

<https://drive.google.com/drive/folders/1qRhU05xbDVoXqgYLUuBbGYcFavCPDdgy?usp=sharing>

## 純遞迴 `Tree1.hs`

- 型別 → 用途 → 範例 → 策略 → 定義 → 測試 (Felleisen et al., 2018)
- 先盡量把 `sumTree` 跟 `productTree` 寫得相似，  
然後才把它們抽象成更一般的、可重複利用的模組

## 解譯器 `Arith1.hs`

- 隨機測試、property-based testing (Claessen and Hughes, 2000)
- 進階練習：定義變數 `Arith2.hs`
- 進階練習：用 `Expr` 的 fold 表達 `eval`

## 個別的副作用

---

## Accumulator passing

基本上副作用 (side effect) 就是一段程式除了把傳進來的引數變成傳回去的結果以外做的事情。

我們寫程式有時候會直觀想用副作用。印象最原始的是 state (狀態)：

$result := 0$

$sumTree (Leaf\ n) \quad = \quad result := result + n;$

$result$

$sumTree (Branch\ t_1\ t_2) = sumTree\ t_1;$

$sumTree\ t_2$

如此處理  $Branch (Leaf\ 3) (Branch (Leaf\ 5) (Leaf\ 2))$  的方法是  
 $((0 + 3) + 5) + 2$  還是  $3 + (5 + (2 + 0))$  還是  $3 + (5 + 2)$  ？

## Accumulator passing

基本上副作用 (side effect) 就是一段程式除了把傳進來的引數變成傳回去的結果以外做的事情。

我們寫程式有時候會直觀想用副作用。印象最原始的是 state (狀態)：

$result := 0$

$sumTree (Leaf\ n) \quad = \quad result := result + n;$

$result$

$sumTree (Branch\ t_1\ t_2) = sumTree\ t_1;$

$sumTree\ t_2$

如此處理  $Branch (Leaf\ 3) (Branch (Leaf\ 5) (Leaf\ 2))$  的方法是  
 $((0 + 3) + 5) + 2$

`TreeState1.hs` 用  $sumTree'$  定義  $sumTree$

## State threading

*next* := 0

*relabel* (*Leaf* *\_*)            = *next* := *next* + 1;  
                                      *Leaf next*

*relabel* (*Branch* *t*<sub>1</sub> *t*<sub>2</sub>) = *Branch* (*relabel* *t*<sub>1</sub>) (*relabel* *t*<sub>2</sub>)

TreeState2.hs 用 *relabel'* 定義 *relabel*

*seen* := *S.empty*

*unique* (*Leaf* *n*)            = **if** *S.member n seen* **then** *False*  
                                      **else** *seen* := *S.insert n seen*; *True*

*unique* (*Branch* *t*<sub>1</sub> *t*<sub>2</sub>) = *unique* *t*<sub>1</sub> && *unique* *t*<sub>2</sub>

用 *unique'* 定義 *unique* (其實也可以用 *unique''* 定義 *unique* , 那是比較不副作用、比較能平行化的作法)



講出來

把心目中的願望講出來，以便實現。所以如果心目中要的是副作用的話，就把副作用的意義講出來。

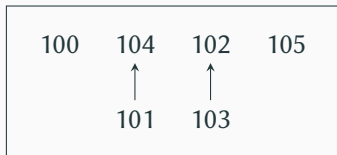
# Local vs global state

## UnionFind1.hs

```
testState :: State
testState = M.fromList
  [ (Key 100, Root 0 "A")
  , (Key 101, Link (Key 104))
  , (Key 102, Root 1 "C")
  , (Key 103, Link (Key 102))
  , (Key 104, Root 1 "E")
  , (Key 105, Root 0 "F") ]
```

```
fresh :: Info → Key
find  :: Key → (Key, Rank, Info)
union :: Key → Key → ()
```

Pointers, references, file system





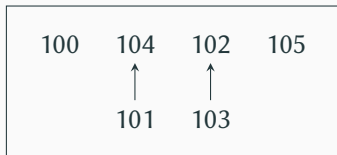
# Local vs global state

## UnionFind1.hs

```
testState :: State
testState = M.fromList
  [ (Key 100, Root 0 "A")
  , (Key 101, Link (Key 104))
  , (Key 102, Root 1 "C")
  , (Key 103, Link (Key 102))
  , (Key 104, Root 1 "E")
  , (Key 105, Root 0 "F") ]
```

```
fresh :: Info → State → (Key, State)
find  :: Key → State → (Key, Rank, Info, State)
union :: Key → Key → State → State
```

Pointers, references, file system

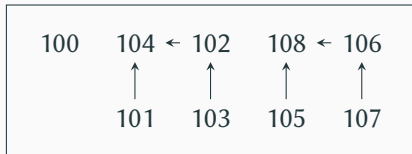
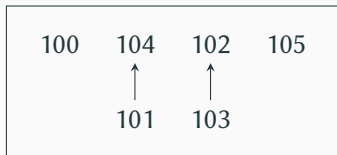


# Local vs global state

## UnionFind1.hs

```
testState :: State
testState' :: State
testState' = M.fromList
  [ (Key 100, Root 0 "A")
  , (Key 101, Link (Key 104))
  , (Key 102, Link (Key 104))
  , (Key 103, Link (Key 102))
  , (Key 104, Root 2 "E")
  , (Key 105, Link (Key 108))
  , (Key 106, Link (Key 108))
  , (Key 107, Link (Key 106))
  , (Key 108, Root 2 "I") ]
```

## Pointers, references, file system



# State-threading interpreter

ArithState1.hs

進階練習：調撥記憶體 ArithState2.hs

```
data Expr = Lit Int | Add Expr Expr | Mul Expr Expr
          | New Expr      -- 把 Expr 的結果存到一個新 allocate 的
                           -- memory cell, 傳回該 cell 的 address
          | Get Expr      -- 把 Expr 的結果當作一個 address,
                           -- 傳回該 cell 目前的內容
          | Put Expr Expr -- 把第一個 Expr 的結果當作一個 address,
                           -- 存入第二個 Expr 的結果並傳回

type State = [Int]      -- 記憶體內容
```

這怎麼會有用？

## Exception (*Maybe*)

把中途跳脫的意義講出來

```
data Maybe a = Nothing | Just a
```

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

TreeMaybe1.hs

- *decTree* 碰到非正數是錯誤
- *productTree* 碰到零有捷徑

ArithMaybe1.hs

- 除以零是錯誤

正常產生的 *Just* 需要 “threading”

## 二十一 點

每個數字遇到時都可以選擇要或是不要，但是一旦超過 21 就爆掉。  
最後得分有哪些可能？

$$11, -1, 11 \quad \rightarrow \quad \{-1, 0, 10, 11, 21\}$$

## TreeNondet1.hs

$$\begin{aligned} \text{blackjack}' &:: \text{Tree} \rightarrow \text{Int} \rightarrow [\text{Int}] \\ \text{blackjack}' (\text{Leaf } n) \quad \text{total} &= \text{if } \text{total} + n > 21 \text{ then } \text{total} \\ &\quad \text{else } \text{amb } [\text{total}, \text{total} + n] \\ \text{blackjack}' (\text{Branch } t_1 \ t_2) \ \text{total} &= \text{blackjack}' \ t_2 \ (\text{blackjack}' \ t_1 \ \text{total}) \end{aligned}$$

用 *blackjack'* 定義 *blackjack*

$$\begin{aligned} \text{concatMap} &:: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b] \\ \text{concatMap } f \text{ as} &= \text{concat } (\text{map } f \text{ as}) \end{aligned}$$



**Nondeterminism**

# Nondeterministic interpreter

ArithNondet1.hs

**data** *Expr* = ... | *Amb Expr Expr*

(McCarthy, 1963)

## 覆面算

$$\frac{x^2 + y^2}{z^2}$$

SEND  
+ MORE  

---

MONEY

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

$$\text{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$$

```
concatMap (λx → concatMap (λy → concatMap (λz → if x2 + y2 == z2
                                                    then [(x, y, z)] else []))
          [0..9]))
```

 $[0..9])$  $[0..9]$



# 覆面算

$$\begin{array}{r} X^2 \\ + Y^2 \\ \hline Z^2 \end{array}$$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

$\text{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$

$\text{concatMap } (\lambda x \rightarrow \text{concatMap } (\lambda y \rightarrow \text{concatMap } (\lambda z \rightarrow \text{if } x^2 + y^2 == z^2$   
 $\text{then } [(x, y, z)] \text{ else } [])$   
 $[0..9]))$   
 $[0..9])$   
 $[0..9]$

好像一個命令    好像一個命令    好像一個命令

# 覆面算

$$\begin{array}{r} X^2 \\ + Y^2 \\ \hline Z^2 \end{array}$$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

*concatMap* :: (*a* → [*b*]) → [*a*] → [*b*]

*concatMap* ( $\lambda d \rightarrow \text{concatMap } (\lambda e \rightarrow \text{concatMap } (\lambda y \rightarrow \text{if } \text{mod } (d + e) \text{ } 10 == y$   
**then ... else** [**]**)  
([0..9] \ [d,e]))  
([0..9] \ [d]))  
[0..9]

趁早檢查，免得做白工

# 覆面算

$$\begin{array}{r} X^2 \\ + Y^2 \\ \hline Z^2 \end{array}$$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

$\text{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$

```
concatMap (\d → concatMap (\e → concatMap (\y → if mod (d + e) 10 == y  
[0..9]                                     then ... else [])  
                                     ([0..9] \ [d,e]))  
([0..9] \ [d]))
```

好像一個命令

趁早檢查，免得做白工

# 覆面算

$$\begin{array}{r} X^2 \\ + Y^2 \\ \hline Z^2 \end{array}$$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

```
type Digit = Int      type Chosen = [ Digit]      -- Crypta1.hs
digit :: Chosen → [ (Digit, Chosen) ]

concatMap (λ(d, chosen) →
    concatMap (λ(e, chosen) →
        concatMap (λ(y, chosen) → if mod (d + e) 10 == y
            then ... else [ ]))
        (digit chosen))
    (digit chosen))
(digit chosen)
```

# 覆面算

$$\begin{array}{r} X^2 \\ + Y^2 \\ \hline Z^2 \end{array}$$

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

$$\begin{array}{r} \text{TO} \\ + \text{GO} \\ \hline \text{OUT} \end{array}$$

**type** *Digit* = *Int*      **type** *Chosen* = [ (*Char*, *Digit*) ]

*digit* :: *Char* → *Chosen* → [ (*Digit*, *Chosen*) ]

*concatMap* (λ(*carry*, *chosen*) → ...)   
            (*add* 'D' 'E' 'Y' ...)

-- *Crypta2.hs*

適合自資料檔讀取新題

# Monad

---

$eval\ (Lit\ v) =$	$eval\ (Add\ e_1\ e_2) =$	$eval\ (Neg\ e_1) =$
$\lambda s \rightarrow (v, s)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad (v_2, s_2) = eval\ e_2\ s_1$ $\quad\quad\quad \mathbf{in}\ (v_1 + v_2, s_2)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad \mathbf{in}\ (-v_1, s_1)$
$Just\ v$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow \mathbf{case}\ eval\ e_2\ \mathbf{of}$ $\quad\quad\quad Nothing \rightarrow Nothing$ $\quad\quad\quad Just\ v_2 \rightarrow Just\ (v_1 + v_2)$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow Just\ (-v_1)$
$[v]$	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 + v_2)$ $\quad\quad\quad (eval\ e_2))$ $\quad\quad\quad (eval\ e_1)$	$map\ (\lambda v_1 \rightarrow -v_1)$ $\quad\quad\quad (eval\ e_1)$

$eval (Lit\ v) =$	$eval (Mul\ e_1\ e_2) =$	$eval (If\ e_1\ et\ ef) =$
$\lambda s \rightarrow (v, s)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad (v_2, s_2) = eval\ e_2\ s_1$ $\quad\quad\quad \mathbf{in}\ (v_1 \times v_2, s_2)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad \mathbf{in}\ eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad\quad\quad\quad \mathbf{else}\ ef)\ s_1$
$Just\ v$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow \mathbf{case}\ eval\ e_2\ \mathbf{of}$ $\quad\quad\quad Nothing \rightarrow Nothing$ $\quad\quad\quad Just\ v_2 \rightarrow Just\ (v_1 \times v_2)$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad\quad\quad\quad \mathbf{else}\ ef)$
$[v]$	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 \times v_2)$ $\quad\quad\quad\quad\quad\quad (eval\ e_2))$ $\quad\quad\quad (eval\ e_1)$	$concatMap\ (\lambda v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad\quad\quad\quad \mathbf{else}\ ef))$ $\quad\quad\quad (eval\ e_1)$



# 把副作用抽象成 monad

(Moggi, 1990; Wadler, 1995)

$eval\ (Lit\ v) =$	$eval\ (Mul\ e_1\ e_2) =$	$eval\ (If\ e_1\ et\ ef) =$
<i>return</i> $v =$ $\lambda s \rightarrow (v, s)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $(v_2, s_2) = eval\ e_2\ s_1$ $\mathbf{in}\ (v_1 \times v_2, s_2)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\mathbf{in}\ eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\mathbf{else}\ ef)\ s_1$
<i>return</i> $v =$ $Just\ v$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $Nothing \rightarrow Nothing$ $Just\ v_1 \rightarrow \mathbf{case}\ eval\ e_2\ \mathbf{of}$ $Nothing \rightarrow Nothing$ $Just\ v_2 \rightarrow Just\ (v_1 \times v_2)$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $Nothing \rightarrow Nothing$ $Just\ v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\mathbf{else}\ ef)$
<i>return</i> $v =$ $[v]$	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 \times v_2)$ $(eval\ e_2))$ $(eval\ e_1)$	$concatMap\ (\lambda v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\mathbf{else}\ ef))$ $(eval\ e_1)$

# 把副作用抽象成 monad

(Moggi, 1990; Wadler, 1995)

$eval\ (Lit\ v) =$	$eval\ (Mul\ e_1\ e_2) =$	$eval\ (If\ e_1\ et\ ef) =$
$return\ v =$ <i>return :: a → s → (a, s)</i>	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad (v_2, s_2) = eval\ e_2\ s_1$ $\quad \dots$ $\quad (v_1 \times v_2, s_2)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad \mathbf{in}\ eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad \quad \mathbf{else}\ ef)\ s_1$
$return\ v =$ <i>return :: a → Maybe a</i>	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad \mathbf{Nothing} \rightarrow \mathbf{Nothing}$ $\quad \mathbf{Just}\ v_1 \rightarrow \mathbf{case}\ eval\ e_2\ \mathbf{of}$ $\quad \quad \mathbf{Nothing} \rightarrow \mathbf{Nothing}$ $\quad \quad \mathbf{Just}\ v_2 \rightarrow \mathbf{Just}\ (v_1 \times v_2)$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad \mathbf{Nothing} \rightarrow \mathbf{Nothing}$ $\quad \mathbf{Just}\ v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad \quad \mathbf{else}\ ef)$
$return\ v =$ <i>return :: a → [a]</i>	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 \times v_2)$ $\quad \quad \quad (eval\ e_2))$ $\quad (eval\ e_1)$	$concatMap\ (\lambda v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad \quad \mathbf{else}\ ef))$ $\quad (eval\ e_1)$

# 把副作用抽象成 monad

(Moggi, 1990; Wadler, 1995)

$eval\ (Lit\ v) =$	$eval\ (Mul\ e_1\ e_2) =$	$eval\ (If\ e_1\ et\ ef) =$
$return\ v =$ $\lambda s \rightarrow (v, s)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad (v_2, s_2) = eval\ e_2\ s_1$ $\quad\quad\quad \mathbf{in}\ (v_1 \times v_2, s_2)$	$\lambda s \rightarrow \mathbf{let}\ (v_1, s_1) = eval\ e_1\ s$ $\quad\quad\quad \mathbf{in}\ eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad \quad\quad\quad \mathbf{else}\ ef)\ s_1$
$return\ v =$ $Just\ v$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow \mathbf{case}\ eval\ e_2\ \mathbf{of}$	$\mathbf{case}\ eval\ e_1\ \mathbf{of}$ $\quad Nothing \rightarrow Nothing$ $\quad Just\ v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad \mathbf{else}\ ef)$
$return\ v =$ $[v]$	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 \times v_2)$ $\quad\quad\quad (eval\ e_1))$	$concatMap\ (\lambda v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\quad\quad\quad \mathbf{else}\ ef))$ $\quad\quad\quad (eval\ e_1)$

$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$   
 $map\ f = concatMap\ (return \circ f)$

$concatMap :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b]$

# 把副作用抽象成 monad

(Moggi, 1990; Wadler, 1995)

$eval\ (Lit\ v) =$	$eval\ (Mul\ e_1\ e_2) =$	$eval\ (If\ e_1\ et\ ef) =$
$return\ v =$ $\lambda s \rightarrow (v, s)$	$concatMap :: (a \rightarrow State \rightarrow (b, State)) \rightarrow (State \rightarrow (a, State)) \rightarrow (State \rightarrow (b, State))$ $concatMap\ f\ m = \lambda s \rightarrow \mathbf{let}\ (a, s_1) = m\ s\ \mathbf{in}\ f\ a\ s_1$ $concatMap\ f\ m = uncurry\ f \circ m$	
$return\ v =$ $Just\ v$	$concatMap :: (a \rightarrow Maybe\ b) \rightarrow Maybe\ a \rightarrow Maybe\ b$ $concatMap\ f\ Nothing = Nothing$ $concatMap\ f\ (Just\ a) = f\ a$	
$return\ v =$ $[v]$	$concatMap\ (\lambda v_1 \rightarrow map\ (\lambda v_2 \rightarrow v_1 \times v_2)$ $\hspace{15em} (eval\ e_2))$ $\hspace{10em} (eval\ e_1)$	$concatMap\ (\lambda v_1 \rightarrow eval\ (\mathbf{if}\ v_1\ \mathbf{then}\ et$ $\hspace{15em} \mathbf{else}\ ef))$ $\hspace{10em} (eval\ e_1)$

## 抽象完畢

```
eval :: Expr → M Int
eval (Lit v)      = return v
eval (Add e1 e2) = concatMap (λv1 → concatMap (λv2 → return (v1 + v2))
                                     (eval e2))
                                     (eval e1)
```

先做  $eval\ e_1$  這個動作，再拿結果  $u1$  去做另一個動作……

```
type M a = { State → (a, State)
            Maybe a
            [a]
```

```
return      :: a → M a           -- unit, pure, eta η
concatMap :: (a → M b) → M a → M b -- bind, ≪, .*
```

# Monad laws

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

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

$\text{return } a \gg= k = k a$

$m \gg= \text{return} = m$

$m \gg= \lambda a \rightarrow (k a \gg= l) = (m \gg= k) \gg= l$

檢查具體特例。 `Laws1.hs` 用 `Int` 以外的型別呢？`[]` 以外的 monad 呢？

**type**  $M a = [a]$

$a = 9 \quad :: Int$

$k = (\lambda n \rightarrow [1..n]) \quad :: Int \rightarrow M Int$

$m = [5, 3] \quad :: M Int$

$l = (\lambda n \rightarrow [n, n \times 10]) \quad :: Int \rightarrow M Int$

# Monad laws

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

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

$\text{return } a \gg= k = k a$

$m \gg= \text{return} = m$

$m \gg= \lambda a \rightarrow (k a \gg= l) = (m \gg= k) \gg= l$

原本的定義：

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

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

$\text{join} :: M (M a) \rightarrow M a$

# Type classes

---



## 動機：很重要所以只說一遍

$elem :: a \rightarrow [a] \rightarrow Bool$

$elem\ x\ [] = False$

$elem\ x\ (y:ys) = x == y \parallel elem\ x\ ys$

## 動機：很重要所以只說一遍

$elem :: a \rightarrow [a] \rightarrow Bool$

$elem\ x\ [] = False$

$elem\ x\ (y:ys) = x == y \parallel elem\ x\ ys$

$elemInt :: Int \rightarrow [Int] \rightarrow Bool$

$elemInt\ x\ [] = False$

$elemInt\ x\ (y:ys) = eqInt\ x\ y \parallel elemInt\ x\ ys$

$elemChar :: Char \rightarrow [Char] \rightarrow Bool$

$elemChar\ x\ [] = False$

$elemChar\ x\ (y:ys) = eqChar\ x\ y \parallel elemChar\ x\ ys$

## 動機：很重要所以只說一遍

$elem :: a \rightarrow [a] \rightarrow Bool$

$elem\ x\ [] = False$

$elem\ x\ (y:ys) = x == y \parallel elem\ x\ ys$

$elemInt :: Int \rightarrow [Int] \rightarrow Bool$

$elemInt\ x\ [] = False$

$elemInt\ x\ (y:ys) = eqInt\ x\ y \parallel elemInt\ x\ ys$

$elemChar :: Char \rightarrow [Char] \rightarrow Bool$

$elemChar\ x\ [] = False$

$elemChar\ x\ (y:ys) = eqChar\ x\ y \parallel elemChar\ x\ ys$

$elemBy :: (a \rightarrow a \rightarrow Bool) \rightarrow a \rightarrow [a] \rightarrow Bool$

$elemBy\ eq\ x\ [] = False$

$elemBy\ eq\ x\ (y:ys) = eq\ x\ y \parallel elemBy\ eq\ x\ ys$

## 模組的使用者

**type** *Eq* *a* = *a* → *a* → *Bool*

*lookupBy* :: *Eq* *a* → *a* → [(*a*, *b*)] → *Maybe* *b*

*lookupBy* *eq* *x* [] = *Nothing*

*lookupBy* *eq* *x* ((*y*, *b*) : *ybs*) = **if** *eq* *x* *y* **then** *Just* *b*  
**else** *lookupBy* *eq* *x* *ybs*

*nubBy* :: *Eq* *a* → [*a*] → [*a*]

*nubBy* *eq* *xs* = *nubBy'* *eq* *xs* []

*nubBy'* :: *Eq* *a* → [*a*] → [*a*] → [*a*]

*nubBy'* *eq* [] *seen* = []

*nubBy'* *eq* (*x* : *xs*) *seen* = **if** *elemBy* *eq* *x* *seen* **then** *nubBy'* *eq* *xs* *seen*  
**else** *x* : *nubBy'* *eq* *xs* (*x* : *seen*)

## 模組的提供者

**type** *Eq* *a* = *a* → *a* → *Bool*

*eqPair* :: *Eq* *a* → *Eq* *b* → *Eq* (*a*, *b*)

*eqPair* *eq\_a* *eq\_b* (*a*<sub>1</sub>, *b*<sub>1</sub>) (*a*<sub>2</sub>, *b*<sub>2</sub>) = *eq\_a* *a*<sub>1</sub> *a*<sub>2</sub> && *eq\_b* *b*<sub>1</sub> *b*<sub>2</sub>

*eqList* :: *Eq* *a* → *Eq* [*a*]

*eqList* *eq\_a* [] [] = *True*

*eqList* *eq\_a* (*x* : *xs*) (*y* : *ys*) = *eq\_a* *x* *y* && *eqList* *eq\_a* *xs* *ys*

*eqList* *eq\_a* \_ \_ = *False*

*eqList* (*eqPair* *eqInt* *eqChar*) :: *Eq* [(*Int*, *Char*)]

```
instance Eq Int where  
  (==) = eqInt
```

```
instance Eq Char where  
  (==) = eqChar
```

**class** *Eq* *a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

**instance** *Eq* *Int* **where**

$(==) = \text{eqInt}$

$(==) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$

**instance** *Eq* *Char* **where**

$(==) = \text{eqChar}$

$(==) :: \text{Char} \rightarrow \text{Char} \rightarrow \text{Bool}$

## 使用 `method` 時生成 `constraint` 累積成 `context` (Wadler and Blott, 1989)

**class** *Eq a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

**instance** *Eq Int* **where**

$(==) = \text{eqInt}$

**instance** *Eq Char* **where**

$(==) = \text{eqChar}$

*elem* :: (*Eq a*)  $\Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$

*elem*  $x [] = \text{False}$

*elem*  $x (y : ys) = x == y \parallel \text{elem } x \text{ } ys$

*lookup* :: (*Eq a*)  $\Rightarrow a \rightarrow [(a, b)] \rightarrow \text{Maybe } b$

*lookup*  $x [] = \text{Nothing}$

*lookup*  $x ((y, b) : ybs) = \text{if } x == y \text{ then Just } b$   
 $\text{else } \text{lookup } x \text{ } ybs$



**class** *Eq a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

**instance** *Eq Int* **where**

$(==) = \text{eqInt}$

**instance** *Eq Char* **where**

$(==) = \text{eqChar}$

**instance** (*Eq a, Eq b*)  $\Rightarrow$  *Eq (a, b)* **where**

$(a_1, b_1) == (a_2, b_2) = a_1 == a_2 \ \&\& \ b_1 == b_2$

**instance** (*Eq a*)  $\Rightarrow$  *Eq [a]* **where**

$[] == [] = \text{True}$

$(x : xs) == (y : ys) = x == y \ \&\& \ xs == ys$

$\_ == \_ = \text{False}$

**class** *Eq a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

**instance** *Eq Int* **where**

$(==) = \text{eqInt}$

**instance** *Eq Char* **where**

$(==) = \text{eqChar}$

**newtype** *Set a* = *MkSet* [*a*]

**instance** (*Eq a*)  $\Rightarrow$  *Eq (Set a)* **where**

$\text{MkSet } xs == \text{MkSet } ys = \text{all } (\lambda x \rightarrow \text{elem } x \text{ } ys) \text{ } xs \ \&\&$   
 $\text{all } (\lambda y \rightarrow \text{elem } y \text{ } xs) \text{ } ys$

$(==) :: \text{Set } a \rightarrow \text{Set } a \rightarrow \text{Bool}$

## Default method implementation

**class** *Eq a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

$x /= y = \text{not } (x == y)$

$x == y = \text{not } (x /= y)$

## Class contexts (superclasses)

**class** *Eq a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

$x /= y = \text{not } (x == y)$

$x == y = \text{not } (x /= y)$

**class** (*Eq a*)  $\Rightarrow$  *Ord a* **where**

$(<), (\leq), (>), (\geq) :: a \rightarrow a \rightarrow \text{Bool}$

$x < y = \text{not } (x == y) \ \&\& \ (x \leq y)$

$x > y = \text{not } (x == y) \ \&\& \ \text{not } (x \leq y)$

$x \geq y = (x == y) \ \parallel \ \text{not } (x \leq y)$

...

## There's no type class like *Show* type class

**class** *Eq* *a* **where**

$(==) :: a \rightarrow a \rightarrow \text{Bool}$

$(/=) :: a \rightarrow a \rightarrow \text{Bool}$

$x /= y = \text{not } (x == y)$

$x == y = \text{not } (x /= y)$

**class** (*Eq* *a*)  $\Rightarrow$  *Ord* *a* **where**

$(<), (\leq), (>), (\geq) :: a \rightarrow a \rightarrow \text{Bool}$

$x < y = \text{not } (x == y) \ \&\& \ (x \leq y)$

$x > y = \text{not } (x == y) \ \&\& \ \text{not } (x \leq y)$

$x \geq y = (x == y) \ \parallel \ \text{not } (x \leq y)$

...

**class** *Show* *a* **where** *show* ::  $a \rightarrow \text{String} \dots$

**class** *Monad* *m* **where**

*return* ::  $a \rightarrow m\ a$

$(\gg=)$  ::  $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

**newtype** *State* *s* *a* = *MkState* { *runState* ::  $s \rightarrow (a, s)$  }

**instance** *Monad* (*State* *s*) **where**

*return* *a* = *MkState* ( $\lambda s \rightarrow (a, s)$ )

$m \gg= k$  = *MkState* ( $\lambda s \rightarrow$  **let** (*a*, *s'*) = *runState* *m* *s*  
**in** *runState* (*k* *a*) *s'*)

至於 *Maybe* 與 `[]` 的 *Monad* instances 則已有內建

**class** *Monad* *m* **where**

*return* :: *a* → *m a*

(*>>=*) :: *m a* → (*a* → *m b*) → *m b*

*MkState* :: (*s* → (*a*, *s*)) → *State s a*

**newtype** *State s a* = *MkState* { *runState* :: *s* → (*a*, *s*) }

**instance** *Monad* (*State s*) **where**

*return a* = *MkState* (*λs* → (*a*, *s*))

*m >>= k* = *MkState* (*λs* → **let** (*a*, *s'*) = *runState m s*  
**in** *runState (k a) s'*)

*runState* :: *State s a* → (*s* → (*a*, *s*))

至於 *Maybe* 與 *[]* 的 *Monad* instances 則已有內建

**class** *Monad* *m* **where**

*return* :: *a* → *m a*

(*>>=*) :: *m a* → (*a* → *m b*) → *m b*

*MkState* :: (*s* → (*a*, *s*)) → *State s a*

**newtype** *State s a* = *MkState* { *runState* :: *s* → (*a*, *s*) }

**instance** *Monad* (*State s*) **where**

*return a* = *MkState* (*λs* → (*a*, *s*))

*m >>= k* = *MkState* (*λs* → **let** (*a*, *s'*) = *runState m s*  
**in** *runState (k a) s'*)

*runState* :: *State s a* → (*s* → (*a*, *s*))

*return* :: *a* → *State s a*

(*>>=*) :: *State s a* → (*a* → *State s b*) → *State s b*

至於 *Maybe* 與 *[]* 的 *Monad* instances 則已有內建



## 輕鬆實作 superclasses

**class** *Functor* *m* **where**

*fmap* ::  $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$

**class** (*Functor* *m*)  $\Rightarrow$  *Applicative* *m* **where**

*pure* ::  $a \rightarrow m\ a$

$(\langle * \rangle)$  ::  $m\ (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$

**class** (*Applicative* *m*)  $\Rightarrow$  *Monad* *m* **where**

*return* ::  $a \rightarrow m\ a$

$(\gg=)$  ::  $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

**instance** *Functor* (*State* *s*) **where** *fmap* = *liftM*

**instance** *Applicative* (*State* *s*) **where** *pure* = *return*;  $(\langle * \rangle) = ap$

# 來寫範例吧！

ArithMonad1.hs

ArithMonad2.hs

ArithMonad3.hs

# Imperative programming

---

ArithIO1.hs 「輸入」、「輸出」是什麼意思呢？

適合用什麼 monad 來表達呢？

ArithIO1.hs 「輸入」、「輸出」是什麼意思呢？

適合用什麼 monad 來表達呢？

```
data IO a = Return a
          | Input (Int → IO a)
          | Output Int (IO a)
```

對程式而言，外界是一個抽象的 monad

*A value of type `IO a` is an “action” that, when performed, may do some input/output, before delivering a value of type `a`.*

```
type IO a = World → (a, World)
```

*(Peyton Jones, 2001)*

```
int main() {  
    return putchar(toupper(getchar()));  
}
```

可譯為

$$main = getChar \gg \lambda c \rightarrow putChar (toUpper c)$$

```
int main() {  
    return putchar(toupper(getchar()));  
}
```

可譯為

$$main = \underbrace{getChar}_{getChar :: IO Char} \gg \lambda c \rightarrow \underbrace{putChar}_{putChar :: Char \rightarrow IO ()} (toUpper c) \quad :: \quad ???$$
  
$$\underbrace{toUpper}_{toUpper :: Char \rightarrow Char}$$

$$\frac{}{\{E[putChar\ c]\} \xrightarrow{!c} \{E[return\ ()]\}} \text{ PUTC}$$

$$\frac{}{\{E[getChar]\} \xrightarrow{?c} \{E[return\ c]\}} \text{ GETC}$$

$$\frac{}{\{E[return\ N \gg M]\} \rightarrow \{E[M\ N]\}} \text{ LUNIT}$$

$$\frac{\llbracket M \rrbracket = \llbracket V \rrbracket \quad M \not\equiv V}{\{E[M]\} \rightarrow \{E[V]\}} \text{ FUN}$$

Semantics 以 labeled transition 在外、denotation 在內

$$main = getChar \gg \lambda c \rightarrow putChar\ (toUpper\ c)$$



$$\frac{}{\{E[putChar\ c]\} \xrightarrow{!c} \{E[return\ ()]\}} \text{ PUTC}$$

$$\frac{}{\{E[getChar]\} \xrightarrow{?c} \{E[return\ c]\}} \text{ GETC}$$

$$\frac{}{\{E[return\ N \gg M]\} \rightarrow \{E[M\ N]\}} \text{ LUNIT}$$

$$\frac{\llbracket M \rrbracket = \llbracket V \rrbracket \quad M \not\equiv V}{\{E[M]\} \rightarrow \{E[V]\}} \text{ FUN}$$

Semantics 以 labeled transition 在外、denotation 在內

$$\begin{aligned}
 & \{getChar \gg \lambda c \rightarrow putChar\ (toUpper\ c)\} \\
 \xrightarrow{?'w'} & \{return\ 'w' \gg \lambda c \rightarrow putChar\ (toUpper\ c)\} && \text{(GETC)} \\
 \rightarrow & \{(\lambda c \rightarrow putChar\ (toUpper\ c))\ 'w'\} && \text{(LUNIT)} \\
 \rightarrow & \{putChar\ 'W'\} && \text{(FUN)} \\
 \xrightarrow{!'W'} & \{return\ ()\} && \text{(PUTC)}
 \end{aligned}$$

為何滿足 preservation? progress?

## Do notation

$$\text{main} = \text{getChar} \gg \lambda c \rightarrow \\ \text{putChar } (\text{toUpper } c)$$
$$\text{main} = \mathbf{do} \ c \leftarrow \text{getChar} \\ \text{putChar } (\text{toUpper } c)$$

## Do notation

$$\begin{aligned} \text{main} = & \text{getChar} \gg \lambda c_1 \rightarrow \\ & \text{getChar} \gg \lambda c_2 \rightarrow \\ & \text{putChar} (\text{toUpper } c_1) \gg \lambda () \rightarrow \\ & \text{putChar} (\text{toLower } c_2) \end{aligned}$$
$$\begin{aligned} \text{main} = & \mathbf{do} \ c_1 \leftarrow \text{getChar} \\ & \ c_2 \leftarrow \text{getChar} \\ & \ () \leftarrow \text{putChar} (\text{toUpper } c_1) \\ & \ \text{putChar} (\text{toLower } c_2) \end{aligned}$$

有哪些 labeled transition sequences?

## Do notation

```
main = getChar >>= \c1 →  
      getChar >>= \c2 →  
      putChar (toUpper c1) >>  
      putChar (toLower c2)
```

$(\gg) :: (\text{Monad } m) \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$   
 $m \gg n = m \gg= \backslash\_ \rightarrow n$

```
main = do c1 ← getChar  
        c2 ← getChar  
        putChar (toUpper c1)  
        putChar (toLower c2)
```

## Do notation

$main = \text{getChar} \gg= \lambda c_1 \rightarrow$

$\text{getChar} \gg$

$\text{putChar} (\text{toUpper } c_1) \gg$

$\text{putChar} (\text{toLower } c_1)$

$(\gg) :: (\text{Monad } m) \Rightarrow m\ a \rightarrow m\ b \rightarrow m\ b$   
 $m \gg n = m \gg= \backslash\_ \rightarrow n$

$main = \text{do } c_1 \leftarrow \text{getChar}$

$\text{getChar}$

$\text{putChar} (\text{toUpper } c_1)$

$\text{putChar} (\text{toLower } c_1)$

## Do notation 用用看

---

把這個 interpreter...	用這個 monad...	在這裡寫成 do notation:
--------------------	--------------	--------------------

---

ArithMonad1.hs	<i>State Int</i>	→ ArithDo1.hs
----------------	------------------	---------------

ArithMonad2.hs	<i>Maybe</i>	→ ArithDo2.hs
----------------	--------------	---------------

ArithMonad3.hs	<i>[]</i>	→ ArithDo3.hs
----------------	-----------	---------------

ArithIO1.hs	<i>IO</i>	→ ArithDo4.hs
-------------	-----------	---------------

---

## Do notation 用用看

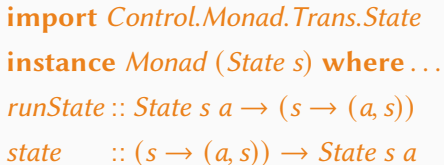
---

把這個 interpreter...    用這個 monad...    在這裡寫成 do notation:

---

ArithMonad1.hs	<i>State Int</i>	→ ArithDo1.hs
ArithMonad2.hs	<i>Maybe</i>	→ ArithDo2.hs
ArithMonad3.hs	<i>[]</i>	→ ArithDo3.hs
ArithIO1.hs	<i>IO</i>	→ ArithDo4.hs

---



```
import Control.Monad.Trans.State
instance Monad (State s) where ...
runState :: State s a → (s → (a, s))
state    :: (s → (a, s)) → State s a
```

## Do notation 表達了 monad laws 的 imperative 直覺

### Left identity

$$\text{return } a \gg= \lambda x \rightarrow k \ x \ = \ k \ a$$

$$\begin{array}{l} \text{do } x \leftarrow \text{return } a \\ \quad k \ x \end{array} \ = \ k \ a$$

### Right identity

$$m \gg= \lambda x \rightarrow \text{return } x \ = \ m$$

$$\begin{array}{l} \text{do } x \leftarrow m \\ \quad \text{return } x \end{array} \ = \ m$$

### Associativity

$$m \gg= \lambda a \rightarrow (k \ a \gg= \lambda b \rightarrow l \ b) \ = \ (m \gg= \lambda a \rightarrow k \ a) \gg= \lambda b \rightarrow l \ b$$

$$\begin{array}{l} \text{do } a \leftarrow m \\ \quad b \leftarrow k \ a \\ \quad \quad l \ b \end{array} \ = \ \begin{array}{l} \text{do } b \leftarrow \text{do } a \leftarrow m \\ \quad \quad \quad k \ a \\ \quad \quad \quad l \ b \end{array}$$



## 單一程式可以應用於各種 monad

```
traverse :: (Monad m) => (a -> m b) -> [a] -> m [b]  -- 又名 mapM
traverse f []      = return []
traverse f (a : as) = do b <- f a
                      bs <- traverse f as
                      return (b : bs)
```

有什麼用呢？

```
renumber "hello" = [0, 1, 2, 3, 4]
choices  [2, 3]   = [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2]]
dec      [2, 5, 3] = Just [1, 4, 2]
dec      [2, 0, 3] = Nothing
```

再多找一些用途！[Traverse1.hs](#)

## 單一程式可以應用於各種 monad

```
data Tree = Leaf Int | Branch Tree Tree
```

```
deriving (Eq, Show)
```

```
traverseTree :: (Monad m) => (Int -> m Int) -> Tree -> m Tree
```

```
traverseTree f (Leaf n)      = do n' <- f n  
                               return (Leaf n')
```

```
traverseTree f (Branch t1 t2) = do t'1 <- traverseTree f t1  
                               t'2 <- traverseTree f t2  
                               return (Branch t'1 t'2)
```

有什麼用呢？[Traverse1.hs](#)

很多資料結構只要提供 `traverse` 就是用途很廣的 API 了。

## Loops1.hs

1.  $\text{forever } \text{action} = \text{action} \gg \text{forever } \text{action}$  型別為何？
2. 用  $\text{forever}$  寫一個一直讀一行（用  $\text{getLine}$ ）然後馬上寫出（用  $\text{putStrLn}$ ）的程式。
3.  $\text{iterateM}_f x = f x \gg= \text{iterateM}_f$  型別為何？
4. 用  $\text{iterateM}_f$  寫一個一直讀數字然後顯示累計總和的程式。
5.  $\text{forever } (\text{getChar} \gg= \text{putChar})$  有哪些 labeled transition sequences?
6.  $\text{iterateM}_f (\lambda c \rightarrow \text{putChar } c \gg \text{getChar}) 'X'$  有哪些 labeled transition sequences?

## Loops2.hs

1. 定義  $\text{replicateM}_\_ :: (\text{Monad } m) \Rightarrow \text{Int} \rightarrow m\ a \rightarrow m\ ()$  使得  $\text{replicateM}_\_ n\ \text{action}$  的意思是把  $\text{action}$  重複  $n$  遍。有什麼用？
2. 定義  $\text{for} :: (\text{Monad } m) \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int} \rightarrow m\ a) \rightarrow m\ ()$  使得  $\text{for}\ \text{from}\ \text{to}\ f$  的意思是做從  $f\ \text{from}$  到  $f\ \text{to}$  的一系列動作。有什麼用？
3. 定義  $\text{while} :: (\text{Monad } m) \Rightarrow m\ \text{Bool} \rightarrow m\ a \rightarrow m\ ()$  使得  $\text{while}\ \text{cond}\ \text{action}$  的意思是重複做  $\text{action}$  直到  $\text{cond}$  的結果成為  $\text{False}$  為止。有什麼用？

## 兩種 monad 的定義可以互相轉換

Join1.hs

$\text{return} :: a \rightarrow m\ a$

$\text{fmap} :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$

$\text{join} :: m\ (m\ a) \rightarrow m\ a$

用  $\text{fmap}$  和  $\text{join}$  定義  $\gg=$       用  $\text{return}$  和  $\gg=$  定義  $\text{fmap}$  和  $\text{join}$



$\text{return} :: a \rightarrow m\ a$

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

## 組合副作用

---

## 邊 state 邊 IO

**newtype** *StateIO* s a = *MkStateIO* { *runStateIO* :: s → IO (a, s) }

StateIO1.hs

- 完成 **instance** *Monad* (*StateIO* s)
- 新語法：**do** 的中間是可以穿插 **let** 的

StateIO2.hs

- 提供 *change* 這個 operation 以便 state 動作
- 提供 *lift* 這個 operation 以便 IO 動作
- 新語法：**(+n)** 就是  $\lambda s \rightarrow s + n$  的意思

## 邊 state 邊 exception

StateMaybe1.hs = StateMaybe2.hs

- *puzzle1* 和 *puzzle2* 應該怎樣？
- 定義 **newtype** *M a* 並完成 **instance** *Monad M*
- 提供 *get* 和 *put* 這兩個 operation 以便 state 動作
- 提供 *divide* 這個 operation 以便 exception 動作
- 找兩組不同的解法！



## 邊 state 邊 nondeterminism

StateNondet1.hs = StateNondet2.hs

- *puzzle1* 和 *puzzle2* 應該怎樣？
- 定義 **newtype** *M a* 並完成 **instance** *Monad M*
- 提供 *get* 和 *put* 這兩個 operation 以便 state 動作
- 提供 *amb* 這個 operation 以便 nondeterminism 動作
- 找兩組不同的解法！

Crypta3.hs 邁向 logic programming (Fischer et al., 2011)

**type** *Parser a = String*  $\rightarrow [(a, \text{String})]$

*parse (many1 number) "12345"*

*Parsing1.hs* 把 *tree* 這個 *Tree* 的 parser 寫完

*Parsing2.hs* 寫一個 parser combinator 把「某種項目的 parser」以及「某種隔間的 parser」組合成為「一串項目的 parser」，叫做 *sepby1*

*Parsing3.hs* 把 *expr* 這個 parser 加上減法的 syntactic sugar

“A parser for things  
is a function from strings  
to lists of pairs  
of things and strings.”

**type** *Prob* *a* = [ (*a*, *Float*) ]

**Prob1.hs** 定義 *die* 擲骰子

**Prob2.hs** 用 *M* (也就是 *Data.Map* 模組) 裡的 *M.toList* 以及 *M.fromListWith* 定義 *coalesce*

**Prob3.hs** 在 *countL* 或 *countR* 內部呼叫 *coalesce*，使得 *coalesce (countL 100 0.5)* 或 *coalesce (countR 100 0.5)* 變得很快

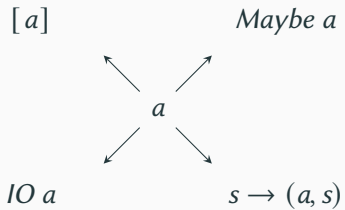
**Prob4.hs** 不僅用 *coalesce*，也得用 **let** 或 **where**，使得 *coalesce (countL 100 0.5)* 和 *coalesce (countR 100 0.5)* 都很快

$$\begin{aligned} \mathbb{P}(A, B) &= \text{do } a \leftarrow \mathbb{P}(A) &= \text{do } b \leftarrow \mathbb{P}(B) \\ &\quad b \leftarrow \mathbb{P}(B \mid A = a) &\quad a \leftarrow \mathbb{P}(A \mid B = b) \\ &\quad \text{return } (a, b) &\quad \text{return } (a, b) \end{aligned}$$

Prob5.hs 丟一枚一元銅板、一枚五元銅板。

1. 看到頭像的機率為何？  
若看到頭像，則一元銅板是頭像的機率為何？
2. 沒有頭像的機率為何？  
若沒有頭像，則一元銅板是頭像的機率為何？

## 有無窮多種 monad



# 有無窮多種 monad

$$s \rightarrow [(a, s)]$$

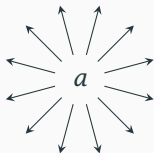
$$[Maybe\ a]$$

$$s \rightarrow (Maybe\ a, s)$$

$$[a]$$

$$Maybe\ a$$

$$[IO\ a]$$



$$s \rightarrow Maybe\ (a, s)$$

$$IO\ a$$

$$s \rightarrow (a, s)$$

$$s \rightarrow (IO\ a, s)$$

$$s \rightarrow IO\ (a, s)$$

$$s \rightarrow IO\ (Maybe\ a, s)$$

哪兩個不行？

## Monad transformers (Liang et al., 1995)

- 把任一個 monad 「 $m$ 」 加一層功能，變成另一個 monad 「 $t\ m$ 」
- 例如  $t = \text{StateT Int}, \text{MaybeT}, \dots$

## Monad transformers (Liang et al., 1995)

- 把任一個 monad 「 $m$ 」 加一層功能，變成另一個 monad 「 $t\ m$ 」
- 例如  $t = \text{StateT } \text{Int}, \text{MaybeT}, \dots :: (\text{Type} \rightarrow \text{Type}) \rightarrow (\text{Type} \rightarrow \text{Type})$
- 不一定 commutative

## Monads

- 把任一個 type 「 $a$ 」 加上副作用，變成「產生  $a$  結果的 computation/action」的 type 「 $m\ a$ 」
- 例如  $m = \text{State } \text{Int}, \text{Maybe}, [], \text{IO}, \dots :: \text{Type} \rightarrow \text{Type}$
- 其他 type constructors 例如  $(,), (\rightarrow) :: \text{Type} \rightarrow \text{Type} \rightarrow \text{Type}$

## Types

- 有 value 進駐 (inhabit) 的
- 例如  $\text{Int}, \text{Bool}, \text{Char}, \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}, \dots :: \text{Type}$



## Composing monad transformers

$StateT :: Type \rightarrow (Type \rightarrow Type) \rightarrow (Type \rightarrow Type)$

$StateT\ s\ m\ a = s \rightarrow m\ (a, s)$

$StateT\ S\ IO\ s = s \rightarrow IO\ (a, s)$

$StateT\ s\ []\ a = s \rightarrow [(a, s)]$

$StateT\ s\ Identity\ a = s \rightarrow (a, s)$

$Identity\ a = a$

## Composing monad transformers

$StateT :: Type \rightarrow (Type \rightarrow Type) \rightarrow (Type \rightarrow Type)$

$StateT\ s\ m\ a = s \rightarrow m\ (a, s)$

$StateT\ S\ IO\ s = s \rightarrow IO\ (a, s)$

$StateT\ s\ []\ a = s \rightarrow [(a, s)]$

$StateT\ s\ Identity\ a = s \rightarrow (a, s)$

$Identity\ a = a$

$MaybeT :: (Type \rightarrow Type) \rightarrow (Type \rightarrow Type)$

$MaybeT\ m\ a = m\ (Maybe\ a)$

$StateT\ s\ (MaybeT\ Identity)\ a = ???$

$MaybeT\ (StateT\ s\ Identity)\ a = ???$

## Composing monad transformers

$StateT :: Type \rightarrow (Type \rightarrow Type) \rightarrow (Type \rightarrow Type)$

$StateT\ s\ m\ a = s \rightarrow m\ (a, s)$

$StateT\ S\ IO\ s = s \rightarrow IO\ (a, s)$

$StateT\ s\ []\ a = s \rightarrow [(a, s)]$

$StateT\ s\ Identity\ a = s \rightarrow (a, s)$

$Identity\ a = a$

**newtype**  $StateT\ s\ m\ a = MkStateT\ \{ runStateT :: s \rightarrow m\ (a, s) \}$

**newtype**  $MaybeT\ m\ a = MkMaybeT\ \{ runMaybeT :: m\ (Maybe\ a) \}$

**class**  $MonadTrans\ t$  **where**

$lift :: (Monad\ m) \Rightarrow m\ a \rightarrow t\ m\ a$

# Monad transformers 用用看

## StateIO3.hs

- 使用共用的 *lift*
- 使用共用的 *modify* 和 *get* 來定義 *change*

## StateMaybe3.hs

- 使用共用的 *empty* 或 *lift* 來定義 *divide*

## StateMaybe4.hs

- 使用共用的 *lift*
- 使用共用的 *empty* 來定義 *divide*

## StateNondet3.hs

- 使用共用的 *empty* (或 *lift*) 以及  $\langle | \rangle$  來定義 *amb*

## StateNondet4.hs

- 使用共用的 *lift*
- 使用共用的 *empty* 以及  $\langle | \rangle$  來定義 *amb*

# 自動微分

---

學習 Linear.hs →

學習一個神經元 Perceptron.hs →

學習神經網路 Network.hs

為了自動調整參數們，所以需要把 loss 相對於參數們微分

1. 把 `loss` 表達為 *Expr*，其中每個參數是一個 *Var*  
`Diff1.hs` 把 `eval` 寫完，增加 *Pow* 與 *Exp* 的 cases
2. 用 dual number 進行 forward automatic differentiation  
(為每個參數都得重跑一遍)  
用 **instance** *Arbitrary Expr* 產生隨機 *Expr* 用來測試 (Claessen and Hughes, 2000)  
用 `traverse` 產生隨機 *Params* 用來起始學習過程  
`randomParams perceptronLoss >>= iterateM_ (optimize 999 perceptronLoss)`  
`randomParams networkLoss >>= iterateM_ (optimize 999 networkLoss)`  
`Diff2.hs` 把 `eval2` 寫完，增加 *Pow* 與 *Exp* 的 cases

3. 增加慣性 (*Inertia*) 加速學習

`Diff3.hs` 把 *randomParams*, *stepParams*, *optimize* 裡處裡慣性的部份寫完

4. 一次對所有參數微分： $(Float, Delta) = (Float, M.Map\ Name\ Float)$

`Diff4.hs` 用 *dScale* 把 *eval3* 寫完，增加 *Pow* 與 *Exp* 的 cases

5. 把 *Delta* 用一個有 *DLet* 的語法樹表示

用 state monad 以 preserve sharing!

`Diff5.hs` 用 *deltaLet* 和 *DScale* 把 *eval4* 寫完，增加 *Pow* 與 *Exp* 的 cases



## References i

- Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00: Proceedings of the ACM International Conference on Functional Programming* (Montréal, Québec, Canada) (*ACM SIGPLAN Notices*, Vol. 35(9)). ACM Press, New York, 268–279. <https://doi.org/10.1145/351240.351266>
- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs* (second ed.). MIT Press, Cambridge. <http://www.htdp.org/2018-01-06/Book/>
- Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely Functional Lazy Nondeterministic Programming. *Journal of Functional Programming* 21, 4–5 (2011), 413–465. <https://doi.org/10.1017/S0956796811000189>
- Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8, 4 (July 1998), 437–444. <https://doi.org/10.1017/S0956796898003050>

## References ii

- Mark P. Jones. 1995a. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, Johan Jeuring and Erik Meijer (Eds.). Number 925 in Lecture Notes in Computer Science. Springer, Berlin, 97–136. <http://web.cecs.pdx.edu/~mpj/pubs/springschool.html>
- Mark P. Jones. 1995b. A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism. *Journal of Functional Programming* 5, 1 (Jan. 1995), 1–35. <https://doi.org/10.1017/S0956796800001210>
- Faustyna Krawiec, Simon Peyton Jones, Neel Krishnaswami, Tom Ellis, Richard A. Eisenberg, and Andrew Fitzgibbon. 2022. Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 48:1–48:30. <https://doi.org/10.1145/3498710>

## References iii

- Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL '95: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (San Francisco, CA). ACM Press, New York, 333–343.  
<https://doi.org/10.1145/199448.199528>
- John McCarthy. 1963. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Number 35 in Studies in Logic and the Foundations of Mathematics. Elsevier Science, Amsterdam, 33–70.  
<http://jmc.stanford.edu/articles/basis/basis.pdf>
- Eugenio Moggi. 1990. *An Abstract View of Programming Languages*. Technical Report ECS-LFCS-90-113. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.  
<http://www.disi.unige.it/person/MoggiE/ftp/abs-view.ps.gz>

## References iv

- Simon L. Peyton Jones. 2001. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. In *Engineering Theories of Software Construction (NATO Science Series: Computer and Systems Sciences, 180)*, Tony Hoare, Manfred Broy, and Ralf Steinbruggen (Eds.). IOS Press, Amsterdam, 47–96. <https://www.microsoft.com/en-us/research/publication/tackling-awkward-squad-monadic-inputoutput-concurrency-exceptions-foreign-language-calls-haskell/> Presented at the 2000 Marktoberdorf Summer School..
- Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *POPL '02: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (Portland, OR). ACM Press, New York, 154–165. <https://www.cs.tufts.edu/~nr/pubs/pmonad-abstract.html>
- Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (Paris). ACM Press, New York, 130–144.

Philip L. Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming: 1st International Spring School on Advanced Functional Programming Techniques*, Johan Jeuring and Erik Meijer (Eds.). Number 925 in Lecture Notes in Computer Science. Springer, Berlin, 24–52.  
<https://homepages.inf.ed.ac.uk/wadler/topics/monads.html#marktoberdorf>

Philip L. Wadler and Stephen Blott. 1989. How to Make *Ad-Hoc* Polymorphism Less *Ad Hoc*. In *POPL '89: Conference Record of the Annual ACM Symposium on Principles of Programming Languages* (Austin). ACM Press, New York, 60–76.  
<https://doi.org/10.1145/75277.75283>