Monad 與副作用

單中杰

2022-08

暖身

純遞迴 `Tree-1.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 `Arith-1.hs`

- 隨機測試、property-based testing (Claessen and Hughes, 2000)
- 進階練習：定義變數 `Arith-2.hs`

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）：

$$
\begin{aligned}
&\mathit{result} := 0 \\
&\mathit{sumTree}\ (\mathit{Leaf}\ n) &&= \mathit{result} := \mathit{result} + n; \\
& && \quad\ \mathit{result} \\
&\mathit{sumTree}\ (\mathit{Branch}\ t_1\ \ t_2) &&= \mathit{sumTree}\ t_1; \\
& && \quad\ \mathit{sumTree}\ t_2
\end{aligned}
$$

如此處理 $\mathit{Branch}\ (\mathit{Leaf}\ 3)\ (\mathit{Branch}\ (\mathit{Leaf}\ 5)\ (\mathit{Leaf}\ 2))$ 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）：

$$
\begin{aligned}
&\mathit{result} := 0\\
&\mathit{sumTree}\ (\mathit{Leaf}\ n) &&= \mathit{result} := \mathit{result} + n;\\
& &&\quad\ \mathit{result}\\
&\mathit{sumTree}\ (\mathit{Branch}\ t_1\ \ t_2) &&= \mathit{sumTree}\ t_1;\\
& &&\quad\ \mathit{sumTree}\ t_2
\end{aligned}
$$

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

`TreeState-1.hs` 用 $\mathit{sumTree}'$ 定義 $\mathit{sumTree}$

State threading

$$
\begin{aligned}
&\mathit{next} := 0\\
&\mathit{relabel}\ (\mathit{Leaf}\ \_) &&= \mathit{next} := \mathit{next} + 1;\\
& && \ \mathit{Leaf}\ \mathit{next}\\
&\mathit{relabel}\ (\mathit{Branch}\ t_1\ t_2) &&= \mathit{Branch}\ (\mathit{relabel}\ t_1)\ (\mathit{relabel}\ t_2)
\end{aligned}
$$

`TreeState-2.hs` 用 *relabel′* 定義 *relabel*

$$
\begin{aligned}
&\mathit{seen} := \mathit{S.empty}\\
&\mathit{unique}\ (\mathit{Leaf}\ n) &&= \mathbf{if}\ \mathit{S.member}\ n\ \mathit{seen}\ \mathbf{then}\ \mathit{False}\\
& && \ \mathbf{else}\ \mathit{seen} := \mathit{S.insert}\ n\ \mathit{seen};\ \mathit{True}\\
&\mathit{unique}\ (\mathit{Branch}\ t_1\ t_2) &&= \mathit{unique}\ t_1\ \&\&\ \mathit{unique}\ t_2
\end{aligned}
$$

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）

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。

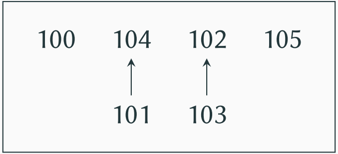
Local vs global state

UnionFind-1.hs

```haskell
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") ]
```

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

Pointers, references, file system



Local vs global state

UnionFind-1.hs

Pointers, references, file system

```haskell
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") ]
```

$$\begin{tikzcd} 100 & 104 & 102 & 105 \\ & 101 \arrow[u] & 103 \arrow[u] & \end{tikzcd}$$

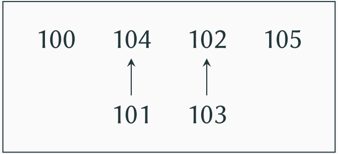
```haskell
fresh :: Info → State → (Key, State)
find  :: Key → State → (Key, Rank, Info, State)
union :: Key → Key → State → State
```

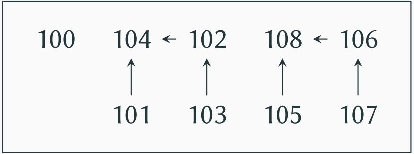
Local vs global state

UnionFind-1.hs

```haskell
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

`ArithState-1.hs`

進階練習：調撥記憶體 `ArithState-2.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

$$\mathbf{data}\ \mathit{Maybe}\quad a = \mathit{Nothing} \mid \mathit{Just}\quad a$$

$$\mathbf{data}\ \mathit{Either}\ b\ a = \mathit{Left}\ b \quad \mid \mathit{Right}\ a$$

`TreeMaybe-1.hs`

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

`ArithMaybe-1.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？

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

TreeNondet-1.hs

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

用 $\mathit{blackjack'}$ 定義 $\mathit{blackjack}$

$$\begin{array}{l} \mathit{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b] \\ \mathit{concatMap}\ f\ \mathit{as} = \mathit{concat}\ (\mathit{map}\ f\ \mathit{as}) \end{array}$$

Nondeterminism

覆面算

```
   X²         SEND        TO
 + Y²       + MORE      + GO
 ----       ------      ----
   Z²        MONEY       OUT
```

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

$$\begin{aligned}&concatMap\ (\lambda x \to concatMap\ (\lambda y \to concatMap\ (\lambda z \to \mathbf{if}\ x^2 + y^2 == z^2\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ [(x, y, z)]\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ [\,])\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad [0\,..\,9])\\ &\qquad\qquad\qquad\qquad [0\,..\,9])\\ &\qquad\quad [0\,..\,9]\end{aligned}$$

覆面算

$$\begin{array}{r} X^2 \\ +\ Y^2 \\ \hline Z^2 \end{array} \qquad \begin{array}{r} \mathtt{SEND} \\ +\ \mathtt{MORE} \\ \hline \mathtt{MONEY} \end{array} \qquad \begin{array}{r} \mathtt{TO} \\ +\ \mathtt{GO} \\ \hline \mathtt{OUT} \end{array}$$

$$\begin{array}{l} \textbf{type}\ \mathit{Digit} = \mathit{Int} \\ \mathit{digit} :: (\mathit{Digit} \to [\mathit{Answer}]) \to \mathit{Answer} \\ \mathit{digit}\ (\lambda x \to \mathit{digit}\ (\lambda y \to \mathit{digit}\ (\lambda z \to \textbf{if}\ x^2 + y^2 == z^2 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then}\ [(x, y, z)] \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ [\,]))) \end{array}$$

可以把每一個 loop body 想成一個 *Digit* 的 continuation
所以「$\mathit{digit}\ (\lambda x \to$」好像一個命令

覆面算

$$\begin{array}{r} \mathtt{X}^2 \\ +\ \mathtt{Y}^2 \\ \hline \mathtt{Z}^2 \end{array} \qquad \begin{array}{r} \mathtt{SEND} \\ +\ \mathtt{MORE} \\ \hline \mathtt{MONEY} \end{array} \qquad \begin{array}{r} \mathtt{TO} \\ +\ \mathtt{GO} \\ \hline \mathtt{OUT} \end{array}$$

$$\begin{array}{l} \mathit{concatMap}\ (\lambda d \rightarrow \mathit{concatMap}\ (\lambda e \rightarrow \mathit{concatMap}\ (\lambda y \rightarrow \mathbf{if}\ \mathit{mod}\ (d+e)\ 10 == y \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ [\,]) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ([0\,..\,9] \mathbin{\backslash\backslash} [d, e])) \\ \qquad\qquad\qquad\qquad\quad ([0\,..\,9] \mathbin{\backslash\backslash} [d])) \\ \qquad\quad [0\,..\,9] \end{array}$$

趁早檢查，免得做白工

覆面算

$$\begin{array}{r} X^2 \\ +\ Y^2 \\ \hline Z^2 \end{array} \qquad \begin{array}{r} \mathtt{SEND} \\ +\ \mathtt{MORE} \\ \hline \mathtt{MONEY} \end{array} \qquad \begin{array}{r} \mathtt{TO} \\ +\ \mathtt{GO} \\ \hline \mathtt{OUT} \end{array}$$

$$\begin{array}{l} \mathbf{type}\ \mathit{Chosen} = [\mathit{Digit}] \\ \mathit{digit} :: (\mathit{Digit} \to \mathit{Chosen} \to [\mathit{Answer}]) \to \mathit{Chosen} \to [\mathit{Answer}] \\ \mathit{digit}\ (\lambda d \to \mathit{digit}\ (\lambda e \to \mathit{digit}\ (\lambda y \to \mathbf{if}\ \mathit{mod}\ (d + e)\ 10 == y \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then} \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \lambda \mathit{chosen} \to [\,]))) \end{array}$$

Crypta-1.hs

覆面算

```
  X²          SEND        TO
+ Y²        + MORE      + GO
-----       ------      ----
  Z²         MONEY       OUT
```

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

$digit :: Char \rightarrow (Digit \rightarrow Chosen \rightarrow [Answer]) \rightarrow Chosen \rightarrow [Answer]$

add `'D' 'E' 'Y' ...`

`Crypta-2.hs` 適合自資料檔讀取新題

Nondeterministic interpreter

`ArithNondet-1.hs`

$$\textbf{data}\ \mathit{Expr} = \cdots \mid \mathit{Amb}\ \mathit{Expr}\ \mathit{Expr}$$

(McCarthy, 1963)

Monad

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

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

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

把副作用抽象成 monad (Moggi, 1990; Wadler, 1995)

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

抽象完畢

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \to M\ \mathit{Int} \\
&\mathit{eval}\ (\mathit{Lit}\ v) = \mathit{return}\ v \\
&\mathit{eval}\ (\mathit{Add}\ e_1\ e_2) = \mathit{concatMap}\ (\lambda v_1 \to \mathit{concatMap}\ (\lambda v_2 \to \mathit{return}\ (v_1 + v_2)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{eval}\ e_2)) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{eval}\ e_1)
\end{aligned}
$$

先做 $\mathit{eval}\ e_1$ 這個動作，再拿結果 $u1$ 去做另一個動作⋯⋯

$$
\textbf{type}\ M\ a = \begin{cases} \mathit{State} \to (a, \mathit{State}) \\ \mathit{Maybe}\ a \\ [\,a\,] \end{cases}
$$

$$
\begin{aligned}
&\mathit{return} :: a \to M\ a && \text{-- unit, pure, eta } \eta \\
&\mathit{concatMap} :: (a \to M\ b) \to M\ a \to M\ b && \text{-- bind, } {=\!<\!<}, \cdot^{\star}
\end{aligned}
$$

Monad laws

$$
\begin{aligned}
&\mathit{return} :: a \to M\ a \\
&(\gg\!=)\ :: M\ a \to (a \to M\ b) \to M\ b
\end{aligned}
$$

$$
\begin{aligned}
\mathit{return}\ a \gg\!= k &= k\ a \\
m \gg\!= \mathit{return} &= m \\
m \gg\!= \lambda a \to (k\ a \gg\!= l) &= (m \gg\!= k) \gg\!= l
\end{aligned}
$$

檢查具體特例。 `Laws-1.hs` 用 Int 以外的型別呢？[ ] 以外的 monad 呢？

$$
\begin{aligned}
&\mathbf{type}\ M\ a = [a] \\
&a\ = 9 && :: \mathit{Int} \\
&k\ = (\lambda n \to [1\,..\,n]) && :: \mathit{Int} \to M\ \mathit{Int} \\
&m = [5, 3] && :: M\ \mathit{Int} \\
&l\ = (\lambda n \to [n, n \times 10]) && :: \mathit{Int} \to M\ \mathit{Int}
\end{aligned}
$$

Monad laws

$$
\begin{aligned}
&\mathit{return} :: a \to M\ a \\
&(\gg\!=) \ \ :: M\ a \to (a \to M\ b) \to M\ b
\end{aligned}
$$

$$
\begin{aligned}
\mathit{return}\ a \gg\!= k &= k\ a \\
m \gg\!= \mathit{return} &= m \\
m \gg\!= \lambda a \to (k\ a \gg\!= l) &= (m \gg\!= k) \gg\!= l
\end{aligned}
$$

原本的定義：

$$
\begin{aligned}
&\mathit{return} :: a \to M\ a \\
&\mathit{fmap} \ \ :: (a \to b) \to M\ a \to M\ b \\
&\mathit{join} \ \ \ \ :: M\ (M\ a) \to M\ a
\end{aligned}
$$

Type classes

動機：很重要所以只說一遍

$$
\begin{array}{llll}
elem :: a \rightarrow [a] & \rightarrow & Bool \\
elem\ \ x\ \ [\,] & = & False \\
elem\ \ x\ \ (y : ys) & = & x == y \mathbin{\|} elem\ x\ ys
\end{array}
$$

動機：很重要所以只說一遍

$$
\begin{array}{lllll}
\mathit{elem} :: a & \to [a] & \to \mathit{Bool} \\
\mathit{elem}\ \ x & [\,] & = \mathit{False} \\
\mathit{elem}\ \ x & (y : ys) & = x == y \parallel \mathit{elem}\ x\ ys
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{elemInt} :: & \mathit{Int} & \to [\mathit{Int}] & \to \mathit{Bool} \\
\mathit{elemInt} & x & [\,] & = \mathit{False} \\
\mathit{elemInt} & x & (y : ys) & = \mathit{eqInt}\ x\ y \parallel \mathit{elemInt}\ x\ ys
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{elemChar} :: & \mathit{Char} & \to [\mathit{Char}] & \to \mathit{Bool} \\
\mathit{elemChar} & x & [\,] & = \mathit{False} \\
\mathit{elemChar} & x & (y : ys) & = \mathit{eqChar}\ x\ y \parallel \mathit{elemChar}\ x\ ys
\end{array}
$$

動機：很重要所以只說一遍

$$
\begin{array}{llllll}
elem :: & a & \rightarrow [a] & \rightarrow & Bool \\
elem & x & [\,] & = & False \\
elem & x & (y : ys) & = & x == y \mathbin{\|} elem\ x\ ys
\end{array}
$$

$$
\begin{array}{llllll}
elemInt :: & Int & \rightarrow [Int] & \rightarrow & Bool \\
elemInt & x & [\,] & = & False \\
elemInt & x & (y : ys) & = & eqInt\ x\ y \mathbin{\|} elemInt\ x\ ys
\end{array}
$$

$$
\begin{array}{llllll}
elemChar :: & Char & \rightarrow [Char] & \rightarrow & Bool \\
elemChar & x & [\,] & = & False \\
elemChar & x & (y : ys) & = & eqChar\ x\ y \mathbin{\|} elemChar\ x\ ys
\end{array}
$$

$$
\begin{array}{lllllll}
elemBy :: & (a \rightarrow a \rightarrow Bool) & \rightarrow a & \rightarrow [a] & \rightarrow & Bool \\
elemBy & eq & x & [\,] & = & False \\
elemBy & eq & x & (y : ys) & = & eq\ x\ y \mathbin{\|} elemBy\ eq\ x\ ys
\end{array}
$$

模組的使用者

$$\mathbf{type}\ Eq\ a = a \to a \to Bool$$

$$\begin{array}{lllll}
lookupBy :: Eq\ a \to a \to [(a,b)] & & & \to & Maybe\ b \\
lookupBy & eq & x\ [\,] & = & Nothing \\
lookupBy & eq & x\ ((y,b):ybs) & = & \mathbf{if}\ eq\ x\ y\ \mathbf{then}\ Just\ b \\
 & & & & \mathbf{else}\ lookupBy\ eq\ x\ ybs
\end{array}$$

$$\begin{array}{llll}
nubBy :: Eq\ a \to [a] \to [a] & & & \\
nubBy & eq\ xs & = & nubBy'\ eq\ xs\ [\,]
\end{array}$$

$$\begin{array}{lllll}
nubBy' :: Eq\ a \to [a] \to [a] & & \to & [a] & \\
nubBy' & eq\ [\,] & seen & = & [\,] \\
nubBy' & eq\ (x:xs) & seen & = & \mathbf{if}\ elemBy\ eq\ x\ seen\ \mathbf{then}\ nubBy'\ eq\ xs\ seen \\
 & & & & \mathbf{else}\ x : nubBy'\ eq\ xs\ (x:seen)
\end{array}$$

模組的提供者

$$\textbf{type}\ Eq\ a = a \to a \to Bool$$

$$
\begin{aligned}
&eqPair :: Eq\ a \to Eq\ b \to Eq\ (a, b)\\
&eqPair\ eq\_a\ eq\_b\ (a_1, b_1)\ (a_2, b_2) = eq\_a\ a_1\ a_2\ \&\&\ eq\_b\ b_1\ b_2
\end{aligned}
$$

$$
\begin{aligned}
&eqList :: Eq\ a \to Eq\ [a]\\
&eqList\ eq\_a\ [\,]\qquad\quad [\,]\qquad\quad = True\\
&eqList\ eq\_a\ (x : xs)\ (y : ys) = eq\_a\ x\ y\ \&\&\ eqList\ eq\_a\ xs\ ys\\
&eqList\ eq\_a\ \_\qquad\quad \_\qquad\quad = False
\end{aligned}
$$

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

內定的 dictionary 叫做 instance (Wadler and Blott, 1989)

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Eq}\ \mathit{Int}\ \textbf{where} \\
\quad (==) = \mathit{eqInt} \\
\textbf{instance}\ \mathit{Eq}\ \mathit{Char}\ \textbf{where} \\
\quad (==) = \mathit{eqChar}
\end{array}
$$

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

$$
\begin{aligned}
&\textbf{class } \mathit{Eq}\ a \textbf{ where}\\
&\quad (==) :: a \rightarrow a \rightarrow \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Eq}\ \mathit{Int} \textbf{ where}\\
&\quad (==) = \mathit{eqInt}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{Eq}\ \mathit{Char} \textbf{ where}\\
&\quad (==) = \mathit{eqChar}
\end{aligned}
$$

$$
\begin{array}{llll}
\mathit{elem} :: (\mathit{Eq}\ a) \Rightarrow & a \rightarrow [\,a\,] & \rightarrow & \mathit{Bool}\\
\mathit{elem} & x \quad [\,] & = & \mathit{False}\\
\mathit{elem} & x \quad (y : ys) & = & x == y \,\|\, \mathit{elem}\ x\ ys\\
\mathit{lookup} :: (\mathit{Eq}\ a) \Rightarrow & a \rightarrow [\,(a, b)\,] & \rightarrow & \mathit{Maybe}\ b\\
\mathit{lookup} & x \quad [\,] & = & \mathit{Nothing}\\
\mathit{lookup} & x \quad ((y, b) : ybs) & = & \textbf{if } x == y \textbf{ then } \mathit{Just}\ b\\
 & & & \textbf{else } \mathit{lookup}\ x\ ybs
\end{array}
$$

自動組成新 instance (Wadler and Blott, 1989)

$$
\begin{aligned}
&\textbf{class } Eq\ a \textbf{ where}\\
&\quad (==) :: a \to a \to Bool\\
&\textbf{instance } Eq\ Int \textbf{ where}\\
&\quad (==) = eqInt\\
&\textbf{instance } Eq\ Char \textbf{ where}\\
&\quad (==) = eqChar\\
&\textbf{instance } (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b) \textbf{ where}\\
&\quad (a_1, b_1) == (a_2, b_2) = a_1 == a_2\ \&\&\ b_1 == b_2\\
&\textbf{instance } (Eq\ a) \Rightarrow Eq\ [a] \textbf{ where}\\
&\quad [\,] == [\,] = True\\
&\quad (x : xs) == (y : ys) = x == y\ \&\&\ xs == ys\\
&\quad \_ == \_ = False
\end{aligned}
$$

為了指定內定而建立新型別 (Wadler and Blott, 1989)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (==) :: a \rightarrow a \rightarrow \mathit{Bool} \\[1ex]
\textbf{instance}\ \mathit{Eq}\ \mathit{Int}\ \textbf{where} \\
\quad (==) = \mathit{eqInt} \\[1ex]
\textbf{instance}\ \mathit{Eq}\ \mathit{Char}\ \textbf{where} \\
\quad (==) = \mathit{eqChar} \\[1ex]
\textbf{newtype}\ \mathit{Set}\ a = \mathit{MkSet}\ [a] \\[1ex]
\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Set}\ a)\ \textbf{where} \\
\quad \mathit{MkSet}\ \mathit{xs} == \mathit{MkSet}\ \mathit{ys} = \mathit{all}\ (\lambda x \rightarrow \mathit{elem}\ x\ \mathit{ys})\ \mathit{xs}\ \&\& \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{all}\ (\lambda y \rightarrow \mathit{elem}\ y\ \mathit{xs})\ \mathit{ys}
\end{array}
$$

Default method implementation

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (==) :: a \to a \to Bool \\
\quad (/=) :: a \to a \to Bool \\
\quad x \mathrel{/=} y = not\ (x == y) \\
\quad x == y = not\ (x \mathrel{/=} y)
\end{array}
$$

Class contexts (superclasses)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (==) :: a \rightarrow a \rightarrow \mathit{Bool} \\
\quad (/=)\ :: a \rightarrow a \rightarrow \mathit{Bool} \\
\quad x \mathrel{/=} y = \mathit{not}\ (x == y) \\
\quad x == y = \mathit{not}\ (x \mathrel{/=} y) \\
\textbf{class}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Ord}\ a\ \textbf{where} \\
\quad (<), (\leqslant), (>), (\geqslant) :: a \rightarrow a \rightarrow \mathit{Bool} \\
\quad x < y = \mathit{not}\ (x == y)\ \&\&\ \qquad (x \leqslant y) \\
\quad x > y = \mathit{not}\ (x == y)\ \&\&\ \mathit{not}\ (x \leqslant y) \\
\quad x \geqslant y = \qquad (x == y)\ \ \|\ \ \mathit{not}\ (x \leqslant y) \\
\quad \ldots
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (==) :: a \rightarrow a \rightarrow Bool \\
\quad (/=)\ :: a \rightarrow a \rightarrow Bool \\
\quad x \mathbin{/=} y = not\ (x == y) \\
\quad x == y = not\ (x \mathbin{/=} y) \\
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} \\
\quad (<), (\leqslant), (>), (\geqslant) :: a \rightarrow a \rightarrow Bool \\
\quad x < y = not\ (x == y)\ \&\&\ \qquad (x \leqslant y) \\
\quad x > y = not\ (x == y)\ \&\&\ not\ (x \leqslant y) \\
\quad x \geqslant y = \qquad (x == y)\ \ \|\ \ not\ (x \leqslant y) \\
\quad \ldots \\
\textbf{class}\ Show\ a\ \textbf{where}\ show :: a \rightarrow String \ldots
\end{array}
$$

Monad 是一個 type class

$$
\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m\ \textbf{where}\\
&\quad \mathit{return} :: a \to m\ a\\
&\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

$$
\textbf{newtype}\ \mathit{State}\ s\ a = \mathit{State}\ \{\mathit{runState} :: s \to (a, s)\}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Monad}\ (\mathit{State}\ s)\ \textbf{where}\\
&\quad \mathit{return}\ a = \mathit{State}\ (\lambda s \to (a, s))\\
&\quad m \gg\!= k\ \ = \mathit{State}\ (\lambda s \to \textbf{let}\ (a, s') = \mathit{runState}\ m\ s\\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ \mathit{runState}\ (k\ a)\ s')
\end{aligned}
$$

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

輕鬆實作 superclasses

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ m\ \textbf{where}\\
&\quad \mathit{fmap} \ :: (a \to b) \to m\ a \to m\ b\\
&\textbf{class}\ (\mathit{Functor}\ m) \Rightarrow \mathit{Applicative}\ m\ \textbf{where}\\
&\quad \mathit{pure} \ :: a \to m\ a\\
&\quad (\langle * \rangle) \ :: m\ (a \to b) \to m\ a \to m\ b\\
&\textbf{class}\ (\mathit{Applicative}\ m) \Rightarrow \mathit{Monad}\ m\ \textbf{where}\\
&\quad \mathit{return} :: a \to m\ a\\
&\quad (\gg\!=) \ :: m\ a \to (a \to m\ b) \to m\ b\\
&\textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \textbf{where}\ \mathit{fmap} = \mathit{liftM}\\
&\textbf{instance}\ \mathit{Applicative}\ (\mathit{State}\ s)\ \textbf{where}\ \mathit{pure} = \mathit{return};\ (\langle * \rangle) = \mathit{ap}
\end{aligned}
$$

來寫範例吧！

`AithMonad-1.hs`

`ArithMonad-2.hs`

`ArithMonad-3.hs`

Imperative programming

`ArithIO-1.hs` 「輸入」、「輸出」是什麼意思呢？

適合用什麼 monad 來表達呢？

`ArithIO-1.hs` 「輸入」、「輸出」是什麼意思呢？

適合用什麼 monad 來表達呢？

$$
\begin{aligned}
\textbf{data}\ \mathit{IO}\ a &= \mathit{Return}\ a \\
&\mid \mathit{Input}\ (\mathit{Int} \rightarrow \mathit{IO}\ a) \\
&\mid \mathit{Output}\ \mathit{Int}\ (\mathit{IO}\ a)
\end{aligned}
$$

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.*
>
> $\textbf{type}\ \mathit{IO}\ a = \mathit{World} \rightarrow (a, \mathit{World})$
>
> *(Peyton Jones, 2001)*

I/O 也是有意義的 (Peyton Jones, 2001)

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

可譯為

$$main \quad = \quad getChar \ggg\!= \lambda c \rightarrow putChar\ (toUpper\ c)$$

I/O 也是有意義的 (Peyton Jones, 2001)

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

可譯為

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

$$\frac{}{\{\mathbb{E}[\mathit{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\mathit{return}\ ()]\}}\ \text{PUTC} \qquad \frac{}{\{\mathbb{E}[\mathit{getChar}]\} \xrightarrow{?c} \{\mathbb{E}[\mathit{return}\ c]\}}\ \text{GETC}$$

$$\frac{}{\{\mathbb{E}[\mathit{return}\ N \gg\!= M]\} \rightarrow \{\mathbb{E}[M\ N]\}}\ \text{LUNIT} \qquad \frac{[\![M]\!] = [\![V]\!] \quad M \not\equiv V}{\{\mathbb{E}[M]\} \rightarrow \{\mathbb{E}[V]\}}\ \text{FUN}$$

Semantics 以 labeled transition 在外、denotation 在內

$$\mathit{main} \quad = \quad \mathit{getChar} \gg\!= \lambda c \rightarrow \mathit{putChar}\ (\mathit{toUpper}\ c)$$

I/O 也是有意義的 (Peyton Jones, 2001)

$$\frac{}{\{\mathbb{E}[\mathit{putChar}\ c]\} \xrightarrow{!c} \{\mathbb{E}[\mathit{return}\ ()]\}}\ \text{PUTC} \qquad \frac{}{\{\mathbb{E}[\mathit{getChar}]\} \xrightarrow{?c} \{\mathbb{E}[\mathit{return}\ c]\}}\ \text{GETC}$$

$$\frac{}{\{\mathbb{E}[\mathit{return}\ N \gg\!= M]\} \to \{\mathbb{E}[M\ N]\}}\ \text{LUNIT} \qquad \frac{[\![M]\!] = [\![V]\!] \quad M \not\equiv V}{\{\mathbb{E}[M]\} \to \{\mathbb{E}[V]\}}\ \text{FUN}$$

Semantics 以 labeled transition 在外、denotation 在內

$$\begin{array}{lll}
 & \{\mathit{getChar} \gg\!= \lambda c \to \mathit{putChar}\ (\mathit{toUpper}\ c)\} & \\
\xrightarrow{?\texttt{'w'}} & \{\mathit{return}\ \texttt{'w'} \gg\!= \lambda c \to \mathit{putChar}\ (\mathit{toUpper}\ c)\} & (\text{GETC}) \\
\to & \{(\lambda c \to \mathit{putChar}\ (\mathit{toUpper}\ c))\ \ \texttt{'w'}\} & (\text{LUNIT}) \\
\to & \{\mathit{putChar}\ \texttt{'W'}\} & (\text{FUN}) \\
\xrightarrow{!\texttt{'W'}} & \{\mathit{return}\ ()\} & (\text{PUTC})
\end{array}$$

Do notation

$$
\begin{aligned}
main = {} & getChar \gg\!= \lambda c \rightarrow \\
& putChar\ (toUpper\ c)
\end{aligned}
$$

$$
\begin{aligned}
main = \mathbf{do}\ & c \leftarrow getChar \\
& putChar\ (toUpper\ c)
\end{aligned}
$$

Do notation

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

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

Do notation

$$
\begin{aligned}
main = {} & getChar \ggg\!= \lambda c_1 \to \\
& getChar \ggg\!= \lambda c_2 \to \\
& putChar\ (toUpper\ c_1) \gg \\
& putChar\ (toLower\ c_2)
\end{aligned}
$$

$$
\begin{aligned}
&(\gg) :: (Monad\ m) \Rightarrow m\ a \to m\ b \to m\ b \\
&m \gg n = m \ggg\!= \backslash\_ \to n
\end{aligned}
$$

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

Do notation

$$\begin{aligned} main = {} & getChar \gg\!= \lambda c_1 \rightarrow \\ & getChar \gg \\ & putChar\ (toUpper\ c_1) \gg \\ & putChar\ (toLower\ c_1) \end{aligned}$$

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

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

Do notation 用用看

把這個 interpreter...	用這個 monad...	在這裡寫成 do notation:
`ArithMonad-1.hs`	*State Int*	→ `ArithDo-1.hs`
`ArithMonad-2.hs`	*Maybe*	→ `ArithDo-2.hs`
`ArithMonad-3.hs`	[ ]	→ `ArithDo-3.hs`
`ArithIO-1.hs`	*IO*	→ `ArithDo-4.hs`

Do notation 用用看

把這個 interpreter...	用這個 monad...	在這裡寫成 do notation:
`ArithMonad-1.hs`	*State Int*	⟶ `ArithDo-1.hs`
`ArithMonad-2.hs`	*Maybe*	⟶ `ArithDo-2.hs`
`ArithMonad-3.hs`	[ ]	⟶ `ArithDo-3.hs`
`ArithIO-1.hs`	*IO*	⟶ `ArithDo-4.hs`

import *Control.Monad.Trans.State*

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

$runState :: State\ s\ a \rightarrow (s \rightarrow (a, s))$

$state \quad :: (s \rightarrow (a, s)) \rightarrow State\ s\ a$

Do notation 表達了 monad laws 的 imperative 直覺

Left identity

$$\mathit{return}\ a \gg\!= \lambda x \rightarrow k\ x \quad = \quad k\ a$$

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

Right identity

$$m \gg\!= \lambda x \rightarrow \mathit{return}\ x \quad = \quad m$$

$$\boxed{\begin{array}{l}\textbf{do}\ x \leftarrow m \\ \quad\ \mathit{return}\ x\end{array}} \quad = \quad m$$

Associativity

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

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

單一程式可以應用於各種 monad

$$
\begin{aligned}
&\mathit{traverse} :: (\mathit{Monad}\ m) \Rightarrow (a \rightarrow m\ b) \rightarrow [\,a\,] \rightarrow m\ [\,b\,] \quad \text{-- 又名 } \mathit{mapM} \\
&\mathit{traverse}\ f\ [\,]\qquad\quad = \mathit{return}\ [\,] \\
&\mathit{traverse}\ f\ (a : as) = \mathbf{do}\ b \ \leftarrow f\ a \\
&\qquad\qquad\qquad\qquad\quad\ \ \ bs \leftarrow \mathit{traverse}\ f\ as \\
&\qquad\qquad\qquad\qquad\quad\ \ \ \mathit{return}\ (b : bs)
\end{aligned}
$$

有什麼用呢？

$$
\begin{aligned}
&\mathit{renumber}\ \texttt{"hello"} = [\,0, 1, 2, 3, 4\,] \\
&\mathit{choices}\quad [\,2, 3\,] \qquad = [\,[\,0, 0\,], [\,0, 1\,], [\,0, 2\,], [\,1, 0\,], [\,1, 1\,], [\,1, 2\,]\,] \\
&\mathit{dec}\qquad\ \ [\,2, 5, 3\,] = \mathit{Just}\ [\,1, 4, 2\,] \\
&\mathit{dec}\qquad\ \ [\,2, 0, 3\,] = \mathit{Nothing}
\end{aligned}
$$

再多找一些用處！`Traverse-1.hs`

單一程式可以應用於各種 monad

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Branch}\ \mathit{Tree}\ \mathit{Tree} \\
&\quad \mathbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{traverseTree} :: (\mathit{Monad}\ m) \Rightarrow (\mathit{Int} \to m\ \mathit{Int}) \to \mathit{Tree} \to m\ \mathit{Tree} \\
&\mathit{traverseTree}\ f\ (\mathit{Leaf}\ n) \qquad\quad = \mathbf{do}\ n' \leftarrow f\ n \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{Leaf}\ n') \\
&\mathit{traverseTree}\ f\ (\mathit{Branch}\ t_1\ t_2) = \mathbf{do}\ t_1' \leftarrow \mathit{traverseTree}\ f\ t_1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ t_2' \leftarrow \mathit{traverseTree}\ f\ t_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathit{return}\ (\mathit{Branch}\ t_1'\ t_2')
\end{aligned}
$$

有什麼用呢？`Traverse-1.hs`

很多資料結構只要提供 *traverse* 就是用處很廣的 API 了。

自己的迴圈自己寫

`Loops-1.hs`

1. $\mathit{forever}\ \mathit{action} = \mathit{action} \gg \mathit{forever}\ \mathit{action}$ 型別為何？
2. 用 *forever* 寫一個一直讀一行（用 *getLine*）然後馬上寫出（用 *putStrLn*）的程式。
3. 定義 $\mathit{replicateM\_} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Int} \to m\ a \to m\ ()$ 使得 $\mathit{replicateM\_}\ n\ \mathit{action}$ 的意思是把 *action* 重複 n 遍。有什麼用？
4. 定義 $\mathit{for} :: (\mathit{Monad}\ m) \Rightarrow \mathit{Int} \to \mathit{Int} \to (\mathit{Int} \to m\ a) \to m\ ()$ 使得 $\mathit{for}\ \mathit{from}\ \mathit{to}\ f$ 的意思是做從 $f\ \mathit{from}$ 到 $f\ \mathit{to}$ 的一系列動作。有什麼用？
5. 定義 $\mathit{while} :: (\mathit{Monad}\ m) \Rightarrow m\ \mathit{Bool} \to m\ a \to m\ ()$ 使得 $\mathit{while}\ \mathit{cond}\ \mathit{action}$ 的意思是重複做 *action* 直到 *cond* 的結果成為 *False* 為止。有什麼用？

兩種 monad 的定義可以互相轉換

`Join-1.hs`

$$
\begin{aligned}
\mathit{return} &:: a \to m\ a \\
\mathit{fmap} &:: (a \to b) \to m\ a \to m\ b \\
\mathit{join} &:: m\ (m\ a) \to m\ a
\end{aligned}
$$

用 fmap 和 join 定義 $\gg\!=$

用 return 和 $\gg\!=$ 定義 fmap 和 join

$$
\begin{aligned}
\mathit{return} &:: a \to m\ a \\
(\gg\!=) &:: m\ a \to (a \to m\ b) \to m\ b
\end{aligned}
$$

組合副作用

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/

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`

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..

References iii

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