Monad 與副作用

單中杰 2022-08

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

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

暖身

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）：

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

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 $((0 + 3) + 5) + 2$

`TreeState1.hs` 用 $\mathit{sumTree}'$ 定義 $\mathit{sumTree}$

State threading

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

`TreeState2.hs` 用 *relabel′* 定義 *relabel*

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

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。

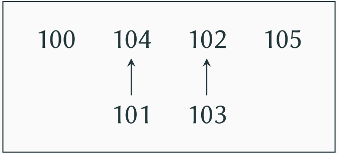
Local vs global state

UnionFind1.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

`UnionFind1.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") ]
```

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

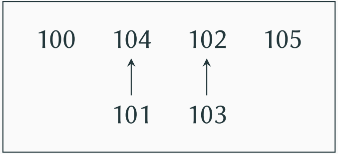
```haskell
fresh :: Info → State → (Key, State)
find  :: Key → State → (Key, Rank, Info, State)
union :: Key → Key → State → State
```

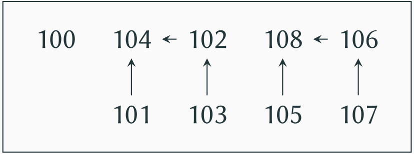
Local vs global state

`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

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

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

`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

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

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

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

Nondeterminism

Nondeterministic interpreter

`ArithNondet1.hs`

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

(McCarthy, 1963)

覆面算

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

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

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

覆面算

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

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

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

好像一個命令

好像一個命令

好像一個命令

覆面算

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

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

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

趁早檢查，免得做白工

覆面算

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

$$\mathit{concatMap} :: (a \to [b]) \to [a] \to [b]$$

$$\boxed{\mathit{concatMap}\ (\lambda d \to \boxed{\mathit{concatMap}\ (\lambda e \to \boxed{\mathit{concatMap}\ (\lambda y \to \boxed{\textbf{if}\ \mathit{mod}\ (d+e)\ 10 == y\ \textbf{then} \ldots \textbf{else}\ [\,])}\ ([0\,..\,9] \setminus\setminus [d,e]))}\ ([0\,..\,9] \setminus\setminus [d]))}\ [0\,..\,9]}$$

好像一個命令

好像一個命令

好像一個命令

趁早檢查，免得做白工

覆面算

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

$\textbf{type}\ \mathit{Digit} = \mathit{Int} \qquad \textbf{type}\ \mathit{Chosen} = [\mathit{Digit}] \qquad$ -- Crypta1.hs

$\mathit{digit} :: \mathit{Chosen} \to [(\mathit{Digit}, \mathit{Chosen})]$

$$
\begin{array}{l}
\mathit{concatMap}\ (\lambda(d, \mathit{chosen}) \to \\
\qquad\qquad \mathit{concatMap}\ (\lambda(e, \mathit{chosen}) \to \\
\qquad\qquad\qquad\qquad \mathit{concatMap}\ (\lambda(y, \mathit{chosen}) \to \textbf{if}\ \mathit{mod}\ (d + e)\ 10 == y \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then} \ldots \textbf{else}\ [\,]) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{digit}\ \mathit{chosen})) \\
\qquad\qquad\qquad\quad (\mathit{digit}\ \mathit{chosen})) \\
\qquad\qquad (\mathit{digit}\ \mathit{chosen})
\end{array}
$$

覆面算

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

$\textbf{type}\ \mathit{Digit} = \mathit{Int} \qquad \textbf{type}\ \mathit{Chosen} = [\,(\mathit{Char}, \mathit{Digit})\,]$ -- `Crypta2.hs`

$\mathit{digit} :: \mathit{Char} \rightarrow \mathit{Chosen} \rightarrow [\,(\mathit{Digit}, \mathit{Chosen})\,]$

$\mathit{concatMap}\ (\lambda(\mathit{carry}, \mathit{chosen}) \rightarrow \ldots)$
$\qquad (\mathit{add}\ \texttt{'D'}\ \texttt{'E'}\ \texttt{'Y'} \ldots)$

適合自資料檔讀取新題

Monad

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

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

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

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

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

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

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

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

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

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

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

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

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

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

抽象完畢

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

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

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

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

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。 `Laws1.hs` 用 *Int* 以外的型別呢？[ ] 以外的 monad 呢？

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

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

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

動機：很重要所以只說一遍

$$
\begin{aligned}
&\mathit{elem} :: a \to [a] \to \mathit{Bool}\\
&\mathit{elem}\ \ x\ \ [\,] = \mathit{False}\\
&\mathit{elem}\ \ x\ \ (y : ys) = x == y \mathbin{\|} \mathit{elem}\ x\ ys
\end{aligned}
$$

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

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

動機：很重要所以只說一遍

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

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

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

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

模組的使用者

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

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

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

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

模組的提供者

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

$$\begin{aligned}
&eqPair :: Eq\ a \rightarrow Eq\ b \rightarrow Eq\ (a, b) \\
&eqPair\ eq_a\ eq_b\ (a_1, b_1)\ (a_2, b_2) = eq_a\ a_1\ a_2\ \&\&\ eq_b\ b_1\ b_2
\end{aligned}$$

$$\begin{array}{lllll}
\multicolumn{5}{l}{eqList :: Eq\ a \rightarrow Eq\ [a]} \\
eqList\ eq_a & [\,] & [\,] & = & True \\
eqList\ eq_a & (x : xs) & (y : ys) & = & eq_a\ x\ y\ \&\&\ eqList\ eq_a\ xs\ ys \\
eqList\ eq_a & _ & _ & = & False
\end{array}$$

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

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

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

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

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

$$\textbf{class}\ Eq\ a\ \textbf{where}$$
$$\quad (==) :: a \to a \to Bool$$

$$\textbf{instance}\ Eq\ Int\ \textbf{where}$$
$$\quad (==) = eqInt$$

$$\boxed{(==) :: Int \to Int \to Bool}$$

$$\textbf{instance}\ Eq\ Char\ \textbf{where}$$
$$\quad (==) = eqChar$$

$$\boxed{(==) :: Char \to Char \to Bool}$$

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

$$
\begin{array}{l}
\textbf{class}\ \mathit{Eq}\ a\ \textbf{where} \\
\quad (==) :: a \to a \to \mathit{Bool}
\end{array}
$$

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

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

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

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

```
class Eq a where
  (==) :: a → a → Bool

instance Eq Int where
  (==) = eqInt

instance Eq Char where
  (==) = eqChar

instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (a1, b1) == (a2, b2) = a1 == a2 && b1 == b2

instance (Eq a) ⇒ Eq [a] where
  []       == []       = True
  (x : xs) == (y : ys) = x == y && xs == ys
  _        == _        = False
```

為了指定內定而建立新 wrapper 型別 (Wadler and Blott, 1989)

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (==) :: a \rightarrow a \rightarrow Bool
\end{array}
$$

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

$$
\textbf{newtype}\ Set\ a = MkSet\ [a]
$$

$$
\begin{array}{l}
\textbf{instance}\ (Eq\ a) \Rightarrow Eq\ (Set\ a)\ \textbf{where} \\
\quad MkSet\ xs == MkSet\ ys = all\ (\lambda x \rightarrow elem\ x\ ys)\ xs\ \&\& \\
\qquad\qquad\qquad\qquad\qquad\quad all\ (\lambda y \rightarrow elem\ y\ xs)\ ys
\end{array}
$$

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

Default method implementation

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

Class contexts (superclasses)

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

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

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

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

Monad 是一個 type class (constructor class) (Jones, 1995a,b)

$$
\begin{array}{l}
\textbf{class}\ \mathit{Monad}\ m\ \textbf{where} \\
\quad \mathit{return} :: a \rightarrow m\ a \\
\quad (\gg\!=) \ \ :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b
\end{array}
$$

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

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

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

Monad 是一個 type class (constructor class) (Jones, 1995a,b)

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

$$\boxed{\mathit{MkState} :: (s \rightarrow (a, s)) \rightarrow \mathit{State}\ s\ a}$$

$$\boxed{\mathit{runState} :: \mathit{State}\ s\ a \rightarrow (s \rightarrow (a, s))}$$

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

Monad 是一個 type class (constructor class) (Jones, 1995a,b)

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

$\boxed{\mathit{MkState} :: (s \to (a, s)) \to \mathit{State}\ s\ a}$

$\boxed{\mathit{runState} :: \mathit{State}\ s\ a \to (s \to (a, s))}$

$\boxed{\mathit{return} :: a \to \mathit{State}\ s\ a}$

$\boxed{(\gg\!=) :: \mathit{State}\ s\ a \to (a \to \mathit{State}\ s\ b) \to \mathit{State}\ s\ b}$

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！

`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？

I/O

`ArithIO1.hs`「輸入」、「輸出」是什麼意思呢？

適合用什麼 monad 來表達呢？

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

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.

$$\textbf{type}\ \mathit{IO}\ a = \mathit{World} \longrightarrow (a, \mathit{World})$$

(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

$$\mathit{putChar} :: \mathit{Char} \to \mathit{IO}\ ()$$

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

$$\mathit{getChar} :: \mathit{IO}\ \mathit{Char} \qquad \mathit{toUpper} :: \mathit{Char} \to \mathit{Char}$$

I/O 也是有意義的 (Peyton Jones, 2001)

$$\dfrac{}{\{\mathbb{E}[putChar\ c]\} \xrightarrow{!c} \{\mathbb{E}[return\ ()]\}}\ \textsc{PutC} \qquad \dfrac{}{\{\mathbb{E}[getChar]\} \xrightarrow{?c} \{\mathbb{E}[return\ c]\}}\ \textsc{GetC}$$

$$\dfrac{}{\{\mathbb{E}[return\ N \gg\!= M]\} \rightarrow \{\mathbb{E}[M\ N]\}}\ \textsc{LUnit} \qquad \dfrac{[\![M]\!] = [\![V]\!] \quad M \not\equiv V}{\{\mathbb{E}[M]\} \rightarrow \{\mathbb{E}[V]\}}\ \textsc{Fun}$$

Semantics 以 labeled transition 在外、denotation 在內

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

I/O 也是有意義的 (Peyton Jones, 2001)

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

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

Semantics 以 labeled transition 在外、denotation 在內

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

為何滿足 preservation? progress?

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

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

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

有哪些 labeled transition sequences?

Do notation

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

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

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

Do notation

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

$(\gg) :: (Monad\ m) \Rightarrow m\ a \to m\ b \to m\ b$

$m \gg n = m \gg\!= \backslash\_ \to n$

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

Do notation 用用看

把這個 interpreter...	用這個 monad...	在這裡寫成 do notation:
`ArithMonad1.hs`	*State Int*	$\longrightarrow$ `ArithDo1.hs`
`ArithMonad2.hs`	*Maybe*	$\longrightarrow$ `ArithDo2.hs`
`ArithMonad3.hs`	[ ]	$\longrightarrow$ `ArithDo3.hs`
`ArithIO1.hs`	*IO*	$\longrightarrow$ `ArithDo4.hs`

Do notation 用用看

把這個 interpreter...	用這個 monad...	在這裡寫成 do notation:
`ArithMonad1.hs`	*State Int*	→ `ArithDo1.hs`
`ArithMonad2.hs`	*Maybe*	→ `ArithDo2.hs`
`ArithMonad3.hs`	[ ]	→ `ArithDo3.hs`
`ArithIO1.hs`	*IO*	→ `ArithDo4.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

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

$$\boxed{\begin{aligned}&\mathbf{do}\ x \leftarrow return\ a\\ &\qquad k\ x\end{aligned}} \quad = \quad k\ a$$

Right identity

$$m \gg\!= \lambda x \rightarrow return\ x \quad = \quad m$$

$$\boxed{\begin{aligned}&\mathbf{do}\ x \leftarrow m\\ &\qquad return\ x\end{aligned}} \quad = \quad m$$

Associativity

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

$$\boxed{\begin{aligned}&\mathbf{do}\ a \leftarrow m\\ &\qquad b \leftarrow k\ a\\ &\qquad l\ b\end{aligned}} \quad = \quad \boxed{\begin{aligned}&\mathbf{do}\ b \leftarrow \mathbf{do}\ a \leftarrow m\\ &\qquad\qquad\qquad\quad k\ a\\ &\qquad l\ b\end{aligned}}$$

單一程式可以應用於各種 monad

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

有什麼用呢？

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

再多找一些用途！`Traverse1.hs`

單一程式可以應用於各種 monad

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

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

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。

自己的迴圈自己寫

`Loops1.hs`

1. $forever\ action = action \gg forever\ action$ 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. $iterateM\_\ f\ x = f\ x \ggg iterateM\_\ f$ 型別為何？
4. 用 *iterateM_* 寫一個一直讀數字然後顯示累計總和的程式。
5. $forever\ (getChar \ggg putChar)$ 有哪些 labeled transition sequences?
6. $iterateM\_\ (\lambda c \to putChar\ c \gg getChar)$ `'X'` 有哪些 labeled transition sequences?

自己的迴圈自己寫

`Loops2.hs`

1. 定義 $replicateM\_ :: (Monad\ m) \Rightarrow Int \rightarrow m\ a \rightarrow m\ ()$ 使得 $replicateM\_\ n\ action$ 的意思是把 $action$ 重複 n 遍。有什麼用？
2. 定義 $for :: (Monad\ m) \Rightarrow Int \rightarrow Int \rightarrow (Int \rightarrow m\ a) \rightarrow m\ ()$ 使得 $for\ from\ to\ f$ 的意思是做從 $f\ from$ 到 $f\ to$ 的一系列動作。有什麼用？
3. 定義 $while :: (Monad\ m) \Rightarrow m\ Bool \rightarrow m\ a \rightarrow m\ ()$ 使得 $while\ cond\ action$ 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

`Join1.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

邊 state 邊 IO

$\textbf{newtype}\ StateIO\ s\ a = MkStateIO\ \{ runStateIO :: s \to IO\ (a, s) \}$

`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 $\lambda s \to s + n$ 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)

$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \to [\,(a, \mathit{String})\,]$

$\mathit{parse}\ (\mathit{many1}\ \mathit{number})$ `"12345"`

> "A parser for things
> is a function from strings
> to lists of pairs
> of things and strings."

`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

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

Conditional probability (Shan and Ramsey, 2017)

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

`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

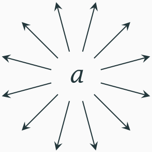
$$\begin{tikzcd}
{[a]} & & Maybe\ a \\
& a \arrow[ul] \arrow[ur] \arrow[dl] \arrow[dr] & \\
IO\ a & & s \to (a, s)
\end{tikzcd}$$

有無窮多種 monad

$s \to [(a, s)]$ $\quad$ $[Maybe\ a]$ $\quad$ $s \to (Maybe\ a, s)$

$[a]$ $\quad$ $Maybe\ a$

$[IO\ a]$



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

$IO\ a$ $\quad$ $s \to (a, s)$

$s \to (IO\ a, s)$ $\quad$ $s \to IO\ (a, s)$ $\quad$ $s \to IO\ (Maybe\ a, s)$

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 $t = StateT\ Int, MaybeT, \ldots$

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 $t = \mathit{StateT\ Int}, \mathit{MaybeT}, \ldots :: (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})$
- 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 $m = \mathit{State\ Int}, \mathit{Maybe}, [\,], \mathit{IO}, \ldots :: \mathit{Type} \to \mathit{Type}$
- 其他 type constructors 例如 $(,), (\to) :: \mathit{Type} \to \mathit{Type} \to \mathit{Type}$

Types

- 有 value 進駐 (inhabit) 的
- 例如 $\mathit{Int}, \mathit{Bool}, \mathit{Char}, \mathit{Int} \to \mathit{Int} \to \mathit{Bool}, \ldots :: \mathit{Type}$

Composing monad transformers

$$
\begin{aligned}
&\mathit{StateT} :: \mathit{Type} \rightarrow (\mathit{Type} \rightarrow \mathit{Type}) \rightarrow (\mathit{Type} \rightarrow \mathit{Type})\\
&\mathit{StateT}\ s\ m\ a && = s \rightarrow m\ (a, s)\\
&\mathit{StateT}\ \mathit{Chosen}\ [\,]\ a && = \mathit{Chosen} \rightarrow [(a, \mathit{Chosen})]\\
&\mathit{State}\ s && = \mathit{StateT}\ s\ \mathit{Identity}\\
&\mathit{Identity}\ a && = a
\end{aligned}
$$

Composing monad transformers

$$
\begin{aligned}
&\mathit{StateT} :: \mathit{Type} \to (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})\\
&\mathit{StateT}\ s\ m\ a &&= s \to m\ (a, s)\\
&\mathit{StateT}\ \mathit{Chosen}\ [\,]\ a &&= \mathit{Chosen} \to [\,(a, \mathit{Chosen})\,]\\
&\mathit{State}\ s &&= \mathit{StateT}\ s\ \mathit{Identity}\\
&\mathit{Identity}\ a &&= a
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{MaybeT} :: (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})\\
&\mathit{MaybeT}\ m\ a &&= m\ (\mathit{Maybe}\ a)\\
&\mathit{StateT}\ \mathit{Chosen}\ (\mathit{MaybeT}\ \mathit{Identity})\ a = ???\\
&\mathit{MaybeT}\ (\mathit{StateT}\ \mathit{Chosen}\ \mathit{Identity})\ a = ???
\end{aligned}
$$

Composing monad transformers

$$
\begin{aligned}
&\mathit{StateT} :: \mathit{Type} \rightarrow (\mathit{Type} \rightarrow \mathit{Type}) \rightarrow (\mathit{Type} \rightarrow \mathit{Type})\\
&\mathit{StateT}\ s\ m\ a &&= s \rightarrow m\ (a, s)\\
&\mathit{StateT}\ \mathit{Chosen}\ [\,]\ a &&= \mathit{Chosen} \rightarrow [\,(a, \mathit{Chosen})\,]\\
&\mathit{State}\ s &&= \mathit{StateT}\ s\ \mathit{Identity}\\
&\mathit{Identity}\ a &&= a
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{newtype}\ \mathit{StateT}\ s\ m\ a = \mathit{MkStateT}\ \{\mathit{runStateT} :: s \rightarrow m\ (a, s)\}\\
&\mathbf{newtype}\ \mathit{MaybeT}\ m\ a = \mathit{MkMaybeT}\ \{\mathit{runMaybeT} :: m\ (\mathit{Maybe}\ a)\}\\
&\mathbf{class}\ \mathit{MonadTrans}\ t\ \mathbf{where}\\
&\quad \mathit{lift} :: (\mathit{Monad}\ m) \Rightarrow m\ a \rightarrow t\ m\ a
\end{aligned}
$$

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*

`StateNondet3.hs`

- 使用共用的 *empty*（或 *lift*）以及 ⟨|⟩ 來定義 *amb*

`StateMaybe4.hs`

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

`StateNondet4.hs`

- 使用共用的 *lift*
- 使用共用的 *empty* 以及 ⟨|⟩ 來定義 *amb*

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`

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`

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`

References v

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`