Haskell 與代數視角

第一性原理：從算術到函數式設計模式

Number

- 閉合： $+ : Number \times Number \to Number$
- 單位元：$0 + a = a + 0 = a$
- 結合律：$(a + b) + c = a + (b + c)$

Number

- 閉合： $\times : Number \times Number \to Number$
- 單位元：$1 \times a = a \times 1 = a$
- 結合律：$(a \times b) \times c = a \times (b \times c)$

Number

- 閉合： $max : Number \times Number \to Number$
- 單位元：$max(-\infty, a) = max(a, -\infty) = a$
- 結合律：$max(max(a, b), c) = max(a, max(b, c))$

Boolean

- 閉合： $\wedge : Boolean \times Boolean \rightarrow Boolean$
- 單位元：$T \wedge a = a \wedge T = a$
- 結合律：$(a \wedge b) \wedge c = a \wedge (b \wedge c)$

String

- 閉合: $+ : String \times String \rightarrow String$
- 單位元：$\epsilon + a = a + \epsilon = a$
- 結合律：$(a + b) + c = a + (b + c)$

List

- 閉合： $+ : List \times List \to List$
- 單位元：$[] + a = a + [] = a$
- 結合律：$(a + b) + c = a + (b + c)$

List

- 閉合： $merge(a, b) := sorted(a + b)$
- 單位元：$merge([], a) = merge(a, []) = a$
- 結合律：$merge(merge(a, b), c) = merge(a, merge(b, c))$

Function

$\cdot \to \cdot$ (泛型)

```
type Func<A, B> = (_: A) => B;
```

- 閉合：$\circ : (Y \to Z) \times (X \to Y) \to (X \to Z)$
- 單位元：$(x \to x) \circ f = f \circ (x \to x)$
- 結合律：$(f \circ g) \circ h = f \circ (g \circ h)$

Function

函數調用不是一個閉合的二元運算

- $\cdot(\cdot) : (X \to Y) \times X \to Y$
- 單位元：$(x \to x)(y) = y$
- 結合律：$(f \circ g)(x) = f(g(x))$

應用式函子 Applicative Functor

函數調用 2.0（代數化）：把函數裝進盒子（泛型），把調用變成加法

```
// 數學（類型）意義上的“盒子”，不（只）是物理意義上的“盒子”
class BlackBox<T> { magic(): Map<string, () => BlackBox<T> | T> {} }
type Applicative<T> = () => BlackBox<BlackBox<T>>;
```

- 閉合： $ap : A[X \to Y] \times A[X] \to A[Y]$
- 單位元：$ap(A[x \to x], Ay) = Ay$
- 結合律：$ap(Af \circ Ag, Ax) = ap(Af, ap(Ag, Ax))$

解析器組合子 Parser Combinator

概念由 Haskell 的 parsec 庫首創

```
const jsonArray = T.leftBracket
  .apr(sepBy(T.comma, jsonValue))
  .apl(T.rightBracket);

const jsonProperty = pure(makeKeyValuePair)
  .ap(jsonString)
  .apl(T.colon)
  .ap(jsonValue);

const jsonObject = T.leftBrace
  .apr(sepBy(T.comma, jsonProperty))
  .apl(T.rightBrace)
  .map(makeObject);

const jsonValue = jsonNull
  .or(jsonBoolean)
  .or(jsonNumber)
  .or(jsonString)
  .or(jsonArray)
  .or(jsonObject);
```

黑洞

吞噬一切

- $0 \times a = 0 \times a = 0$（Number）
- $NaN \cdot a = a \cdot NaN = NaN$（對於任何數字運算）
- $null(x) = f(null) = null$ （純屬妄想）

定義 $ap\left(\begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}, \begin{pmatrix} x_0 \\ x_1 \\ \vdots \end{pmatrix}\right) := \begin{pmatrix} f_0(x_0) \\ f_0(x_1) \\ \vdots \\ f_1(x_0) \\ f_1(x_1) \\ \vdots \end{pmatrix}$，顯然 $ap(Af \circ Ag, Ax) = ap(Af, ap(Ag, Ax))$

定義 $NULL := ()$，則自然得到

$$ap\left(NULL, \begin{pmatrix} x_0 \\ x_1 \\ \vdots \end{pmatrix}\right) = ap\left(\begin{pmatrix} f_0 \\ f_1 \\ \vdots \end{pmatrix}, NULL\right) = NULL$$

函子 Functor

函數調用 1.5：$map : (X \to Y) \times A[X] \to A[Y]$

對於應用式函子：$map(f, Ax) = ap(A[f], Ax)$

$$map\left(f, \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \right) := ap\left(\left(f \right), \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} \right) = \begin{pmatrix} f(x_0) \\ f(x_1) \\ f(x_2) \end{pmatrix}$$

Map + 坍縮

結構坍縮

$$
\begin{aligned}
&bind(A[x], f) && f : X \to A[Y] \\
:=&flatMap(f, A[x]) && \\
:=&flat(map(f, A[x])) && flat : A[A[X]] \to A[X]
\end{aligned}
$$

內容坍縮

> 這裏 M 表示定義了某種加法的集合/類型，數學上叫 Monoid 幺半群

$$
\begin{aligned}
&foldMap(f, A[x]) && f : X \to M \\
:=&fold(map(f, A[x])) && fold : A[M] \to M \\
:=&reduce((m, x) \to m \cdot_M f(x), e, A[x]) &&
\end{aligned}
$$

比如求最大年齡：$A[X] \mapsto List[Person], \quad f \mapsto getAge, \quad \cdot_M \mapsto max$

新代數：單子 Monad

函數調用3.0：拍平也要講基本法

- 定義 $f \circ_K g := x \to flatMap(f, g(x))$，要求 $flat$ 必須滿足：
- 單位元：$flatMap(x \to A[x], f(y)) = flatMap(f, (x \to A[x])(y)) = f(y)$
- 結合律：$flatMap(f \circ_K g, x) = flatMap(f, flatMap(g, x))$

$$
\begin{aligned}
ap\left(\begin{pmatrix} f_0 \\ f_1 \end{pmatrix}, \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \right) :=& bind\left(\begin{pmatrix} f_0 \\ f_1 \end{pmatrix}, f_i \to bind\left(\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, x_j \to \big(f_i(x_j)\big) \right)\right) \\
=& flat\begin{pmatrix} bind\left(\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, x_j \to \big(f_0(x_j)\big) \right) \\ bind\left(\begin{pmatrix} x_0 \\ x_1 \end{pmatrix}, x_j \to \big(f_1(x_j)\big) \right) \end{pmatrix} \\
=& flat\begin{pmatrix} flat\begin{pmatrix} \big(f_0(x_0)\big) \\ \big(f_0(x_1)\big) \end{pmatrix} \\ flat\begin{pmatrix} \big(f_1(x_0)\big) \\ \big(f_1(x_1)\big) \end{pmatrix} \end{pmatrix} = \begin{pmatrix} f_0(x_0) \\ f_0(x_1) \\ f_1(x_0) \\ f_1(x_1) \end{pmatrix}
\end{aligned}
$$

新代數： Kleisli 組合

函數組合3.0：單子的第二種表述

- 定義 $f \circ_K g := x \to flatMap(f, g(x))$，要求 $flat$ 必須滿足：
- 閉合： $\circ_K : (Y \to A[Z]) \times (X \to A[Y]) \to (X \to A[Z])$
- 單位元：$(x \to A[x]) \circ_K f = f \circ_K (x \to A[x]) = f$
- 結合律： $(f \circ_K g) \circ_K h = f \circ_K (g \circ_K h)$

Haskell 中的單子（IO 單子）

(OCT. 1992) 論文 IMPERATIVE FUNCTIONAL PROGRAMMING

```
program = tell "Greetings!" >>= \_ →
            ask "What is your name?" >>= \name →
              tell "Hi " ++ name
```

(MAY. 1996) HASKELL 1.3 "DO-NOTATION" 語法糖:

```
program = do
             _ ← tell "Greetings!"
             name ← ask "What is your name?"
             _ ← tell "Hi " ++ name
             return name
```

Scala 中的單子

```scala
val program = tell("Greetings!") flatMap { s ⇒
  ask("What is your name?") flatMap { name ⇒
    tell("Hi " ++ name) flatMap {_ ⇒
      name
    }
  }
}
```

```scala
val program = for {
  _    ← tell("Greetings!")
  name ← ask("What is your name?")
  _    ← tell("Hi " ++ name)
} yield name
```

```scala
for {
  i ← 0 until n
  j ← 0 until m if i + j > v
} {
  println(s"($i, $j)")
  yield i + j
}
```

JavaScript 中的單子

Monad	`>>=`	"Do-Notation"
`Array`	`Array.prototype.flatMap`	不支持
`Promise`	`Promise.prototype.then`	`async`/`await`
`rxjs/Observable`	`rxjs/operators/concatMap`	不支持
`rxjs/Observable`	`rxjs/operators/mergeMap`	不支持
`rxjs/Observable`	`rxjs/operators/switchMap`	不支持

`Promise` 單子結合律

$$flatMap(f \circ_K g, x) = flatMap(f, flatMap(g, x))$$

```
const x = Promise.resolve(1);
const f = async x => x + 1;
const g = async x => x * 2;

// flatMap(compose(f, g), x)
x.then(y => g(y).then(f))

// flatMap(f, flatMap(g, x))
x.then(g).then(f);
```

Free Monad（自由單子）

React 與 Redux 陣營的理論根源

- **【必讀】Scala Cats**：https://typelevel.org/cats/datatypes/freemonad.html

> 為什麼會出現這些想法：React Fiber 兩階段渲染，React Hooks，Redux，Redux Saga

- 函子可以被直接改造成一個單子，通常用於內嵌 DSL
- 與 algebraic effect 的概念關係密切（effect 系統相當於 free monad 進行 foldMap）

Haskell 運算符

舉世皆代數，萬物皆可加

> **FP == PPAP**

> I have a pen
>
> I have a apple
>
> Uh!
>
> Apple-Pen!

> I have a pen
>
> I have pineapple
>
> Uh!
>
> Pineapple-Pen!

> Apple-Pen
>
> Pineapple-Pen
>
> Uh!
>
> Pen-Pineapple-Apple-Pen!

$a \cdot b$	`a <> b`
$f \circ g$	`f . g`
$f \circ_K g$	`f <=< g`
$f(x)$	`f $ x`
$map(f, x)$	`f <$> x`
$ap(f, x)$	`f <*> x`
$apl(f, x)$	`f <* x`
$apr(f, x)$	`f *> x`
$bind(x, f)$	`x >>= f`

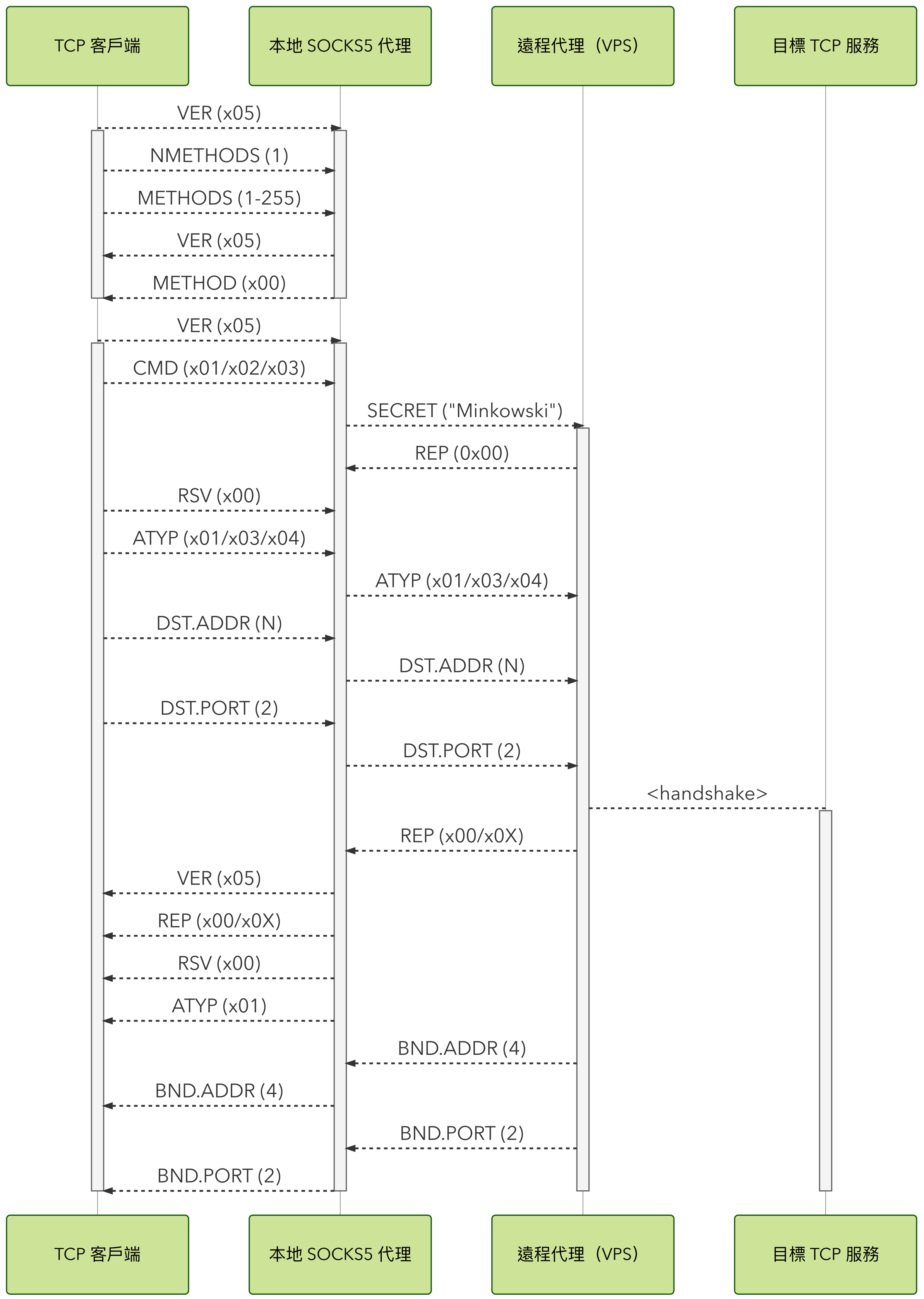
擴展閱讀

- 文章：Functor, Applicative, and Monads in Picture （英 / 中）
- 文章：Free Monad （英）
- 課本：Learn You a Haskell For Great Good（英 / 中）
- 課本：Haskell Programming from First Principles （英）

```python
class Proxy(StreamRequestHandler):
    def forward(self, sock_a: socket, sock_b: socket, size: int = 4096) -> bool:
        chunk = sock_a.recv(size)
        if not chunk:
            return True
        sock_b.send(chunk)
        return False

    def event_loop(self, sock_a: socket, sock_b: socket):
        selector = selectors.DefaultSelector()
        selector.register(sock_a, selectors.EVENT_READ, sock_b)
        selector.register(sock_b, selectors.EVENT_READ, sock_a)
        EOF = False
        while not EOF:
            for key, _ in selector.select():
                EOF = self.forward(key.fileobj, key.data)
        selector.unregister(sock_a)
        selector.unregister(sock_b)

    def handle(self):
        self.accept()
        self.event_loop(self.connection, self.connect())
```



https://datatracker.ietf.org/doc/html/rfc1928