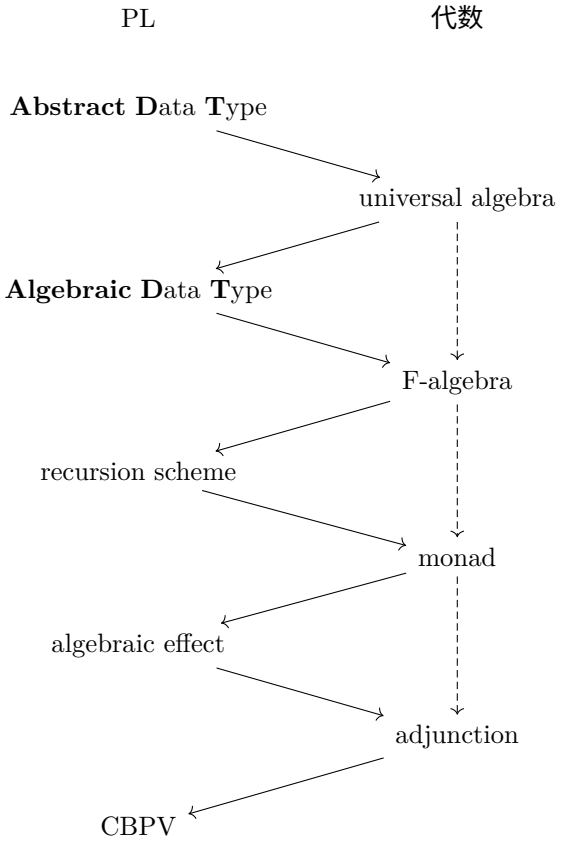
代数与 PL 与 PL 之恋

本文尝试从不同的角度介绍编程语言与 universal algebra 之间的联系。它的内容如下：



对于有些读者来说，有些内容可能会显得过于简单/常识。但我写这篇文章的主要目的之一是，将有关 universal algebra 的诸多基础内容从 PL 的角度提供动机、再将本文涉及到的诸多不同方面的 PL 内容通过 universal algebra 联系起来。此外，我希望阅读本文需要的基础知识尽可能地少。所以为了完整性起见我将 universal algebra 相关的数学定义也加入到了文章中。对它们已经有所了解的读者可以放心跳过有关内容。不过，如果你感兴趣，也可以看看我对这些内容给出的“PL 动机”。

如果是对于范畴论不太熟悉的读者，遇到不了解的范畴论内容可以放心跳过。不过，范畴和 functor 的基本定义的知识还是需要的。在接下来的内容强依赖于某个范畴构造的时候，我会尽量对该构造作出定义与说明。

1 复习：abstract data type 是什么

“隐藏实现细节，暴露接口”是软件工程最重要的原则之一。面向对象的程序员把这叫作“封装”，并用类的私有字段来实现它。而在函数式这边，则往往通过 Abstract Data Type 来实现这一点。一个非常典型的例子是 ML 的 module。例如，在 `OCaml` 中，一个表示逻辑命题的 Abstract DT 可以如此定义：

```
module type Proposition = sig
    type t (* 抽象的类型，表示一个命题 *)

    val tt : t
    val ff : t
    val pand : t -> t -> t
    val por  : t -> t -> t
end
```

同一个 Abstract DT 可以有多个不同的实现，`t` 可以被填入不同的具体类型，`tt`、`ff` 等可以有多个不同的实现。例如，`t` 可以是 `bool`，其余的各个操作进行相应的布尔计算。`t` 也可以填入更复杂的类型，表示那些真假不能直接计算出来的复杂命题。而通过 `Proposition` 这一接口，使用者可以在不知道具体的 `t` 和各个操作实现方式的情况下，构造出各种各样的命题。

2 Abstract? Algebraic!

许多 abstract data type 的形式都与上面的 `Proposition` 类似，它们由一个抽象的类型 `t`，以及关于 `t` 的一系列操作构成。然而，这一结构和数学上的抽象代数是几乎一模一样的。许多耳熟能详的抽象代数结构都能用 Abstract DT 进行表达：

```
module type Monoid = sig
    type t
    val unit : t
    val comp : t -> t -> t
end

module type Group = sig
    include Monoid
    val inv : t -> t
end
```

```
module type VectorSpace = sig
    type t
    val zero : t
    val add  : t -> t -> t
    val mul  : real -> t -> t
end
```

所以，抽象代数结构都能写成 Abstract DT。反过来，我们也可以说，Abstract DT 其实表达了一种代数结构。在数学中同样有表示“一个代数结构”的概念：universal algebra。

Universal algebra 的定义非常简单，基本就是把上面对 Abstract DT 的描述用集合论的语言重新说了一遍。一个 universal algebra 包含：

- 一个集合 A，称为 carrier set。代表对应代数结构中的操作对象的集合
- 一系列操作 $\mathtt{op} : A^n \to A$。例如在 monoid 中，单位元 e 是一个零元的操作，乘法 $- \cdot -$ 是一个二元操作。群中的逆元素 $-^{-1}$ 是一个一元操作
- 关于操作的一系列等式

在 Abstract DT 中，由于语言本身表达能力的局限性，我们无法在类型层面表达出等式。不过，加上等式这一元素之后，Abstract DT 和 universal algebra 就几乎一模一样了。接下来，通过 universal algebra 中的构造，我们可以“重新发现”编程语言中的许多有趣构造。

3 universal algebra: algebraic homomorphism

在 universal algebra 中，不同代数结构之间的态射，algebraic homomorphism，是十分重要的。例如，在两个 monoid 之间有 monoid homomorphism 的定义。假设 M、N 是 monoid，一个 monoid homomorphism $f : M \to N$ 是从 M 的 carrier 到 N 的 carrier 的一个函数，满足：

$$f(e_M) = e_N$$
$$f(a \cdot_M b) = f(a) \cdot_N f(b)$$

各种具体的 homomorphism 的定义可以很容易地拓展到任意 universal algebra。给定一个 universal algebra $\mathcal{A}$ 和两个具体的 $\mathcal{A}$-algebra A、B，一个 $\mathcal{A}$-homomorphism $f : A \to B$ 是一个从 A 的 carrier 到 B 的 carrier 的函数。而且对于 $\mathcal{A}$ 中的每个操作 $\mathtt{op} : A^n \to A$，f 必须满足：

$$f(\mathtt{op}_A(a_1, \ldots, a_n)) = \mathtt{op}_B(f(a_1), \ldots, f(a_n))$$

4 回到 Abstract DT：切换不同的实现

那么，既然 Abstract DT 和 universal algebra 如此相似，algebraic homomorphism 是否在 Abstract DT 中有对应呢？答案是肯定的：Abstract DT 上的

homomorphism，就是在同一个 Abstract DT 的不同实现间变换的函数。

假设我们有两个 Abstract DT 的实现 `AST : Proposition` 和 `Bool : Proposition`，分别表示不判断命题真假、只记录命题形式的 `AST`，和使用 `t = bool` 对命题进行求值的实现。现在，如果我们想要把一个 `AST.t` 的抽象命题求值成一个 `bool = Bool.t`，应该怎么做呢？很显然，我们需要一个 `AST.t -> bool` 的函数。但是，这个函数不能是任意的。`fun (x : AST.t) -> true` 显然不是我们想要的。那么，这个转换函数应该满足什么样的等式呢？如果把 algebraic homomorphism 的定义之间搬过来，可以发现，`Proposition` 的签名已经包含了我们需要的几乎所有等式：

```
f AST.tt         = Bool.tt               = true
f AST.ff         = Bool.ff               = false
f (AST.pand p q) = Bool.pand (f p) (f q) = f p && f q
f (AST.por  p q) = Bool.por  (f p) (f q) = f p || f q
```

上面的等式的左手侧看上去和对代数数据类型（**Algebraic** Data Type）的模式匹配十分相似。那么，这当中是否有某种关联呢？答案是肯定的。而且Abstract Data Type 和 **Algebraic** Data Type 的联系，同样有着非常直接的数学对应。

5 free algebra

在 universal algebra 中有一类叫作“free algebra”的特定 algebra。给定一个 universal algebra $\mathcal{A}$ 与一个代表变量的集合 V，我们可以“免费”构建出一个 $\mathcal{A}$-algebra $F(V)$，构造方式如下：

- 我们将集合 V 中的变量都加入到 $F(V)$ 中
- 对于每个 $\mathcal{A}$ 中的操作 $\mathtt{op} : A^n \to A$，我们将 $\mathtt{op}$ 作为一个符号与 n 的参数 $a_i \in F(V)$ 打包在一起，记作 $[\mathtt{op}; a_1, \ldots, a_n]$，并加入到 $F(V)$ 中

不断重复上述添加过程，直到得到的 $F(V)$ 不再变化，再根据 $\mathcal{A}$ 中的等式对最终的集合取等价类，就得到了 *free algebra*。Free algebra 上的操作被定义为：

$$\mathtt{op}_{F(V)}(a_1, \ldots, a_n) = [\mathtt{op}; a_1, \ldots, a_n]$$

可以看到，free algebra 中的操作不会做任何事情，只是把操作当作一个符号和参数一起打包。所以，我们可以把 free algebra 理解成一种“表达式”。

运用“表达式”这个直觉，可以发现 free algebra 在我们讨论 universal algebra 时简直无处不在。例如，在群中，单位元的性质被表述为：

$$\forall a, e \cdot a = a \cdot e = a$$

但是，在上面的等式中，我们并没有指明我们讨论的是**哪一个**群！既然如此，这条式子中的表达式 $e \cdot a$、$a \cdot e$ 等的意义是什么呢？在我们指定一个具体的群来解释这些表达式之前，它们并不能去到具体的语义。所以，我们只能

以“表达式”的形式原封不动地把操作记录、书写下来——而这正是 free algebra。

Universal algebra 和 Abstract Data Type 有紧密的联系。因此，理所当然地，Abstract DT 中也有“表达式”的概念。我们可以根据一个 Abstract DT 的签名写出一些与具体实现无关的表达式，例如 `pand tt (por ff tt)`。那么，Abstract DT 是否也有对应 free algebra 的一种构造呢？不仅有，而且它比上面基于集合的 free algebra 的定义更加优雅。当我们想在编程语言中用一个数据类型表示一些表达式时，我们往往会设计一颗抽象语法树（AST）。而定义 AST 时，我们有一样绝佳的工具：**Algebraic** Data Type！

6 Algebraic Data Type

考虑本文开头的 `Proposition` 的例子。如果我们要给 `Proposition` 中的命题设计一颗 AST，只需要把 `Proposition` 中的每个操作都变成 A(lgebraic)DT 中的构造器即可：

```
type prop =
    | TT   : prop
    | FF   : prop
    | PAnd : prop -> prop -> prop
    | POr  : prop -> prop -> prop
```

而反过来，对于每个 ADT（以下全部 ADT 指代 **Algebraic** DT），我们都能构建出一个对应的 Abstract DT，例如：

Algebraic Data Type	**Abstract** Data Type

`type nat =`
`  | Zero : nat`
`  | Succ : nat -> nat`
	`module type Nat = sig`
	`  type t`
	`  val zero : t`
	`  val succ : t -> t`
	`end`

`type tree =`
`  | Leaf : tree`
`  | Branch :`
`  tree -> tree -> tree`
	`module type Tree = sig`
	`  type t`
	`  val leaf : t`
	`  val branch : t -> t -> t`
	`end`

可以看到，和在 free algebra 的构造中一样，我们构造 ADT 的思路是把 Abstract DT 中的所有操作变成符号、变成构造器，并把操作和参数原封不动地打包。但是，和 free algebra 的构造不同的是，上面的 AST 定义方式中没有“变量”的概念。这有时会导致一些问题，例如，如果我们希望将 `Monoid` 翻译成 AST，我们会得到：

```
type monoid_bad =
```

```
  | Unit : monoid_bad
  | Comp : monoid_bad -> monoid_bad -> monoid_bad
```

但是，这只是一颗不携带任何数据的二叉树，直觉上，一个“monoid 表达式”应该包含更多灵活性。而且，如果把 monoid 中单位元和复合运算需要满足的等式考虑进去，取 quotient，那这颗 AST 中就只剩下单位元一个元素了！所以，表达变量有时是非常必要的。

要向 ADT 中加入变量也非常简单。只需要变量的类型作为整棵 AST 的一个参数、再加入对应变量的构造器即可：

```
type 'v free_monoid =
    | Var  : 'v -> 'v free_monoid
    | Unit : 'v free_monoid
    | Comp : 'v free_monoid -> 'v free_monoid -> 'v free_monoid
```

对于任意一个 universal algebra/Abstract DT，我们都能给出一个相似的构造。如果对一个抽象的 universal algebra $\mathcal{A}$ 用 ADT 构造出一颗清晰易懂 AST，再把得到的 AST 用集合进行编码，就能重新得到数学中 free algebra 的构造。

7 free algebra 的性质

前面我们提到，free algebra 是一种可以“免费”得到的 algebra。然而，free algebra 的意义远不止于此。Free algebra 作为“表达式”，被用来表达 universal algebra 上各种各样的性质。然而，我们实际想表达的，往往是某些性质对**所有** algebra 成立，而不是**只对 free algebra** 成立。所以，free algebra 必须能够用来代替“任意一个 algebra”。

直觉上，free algebra 之所以能做到这一点，是因为它是“表达式”，它对 algebra 中的操作没有做任何解释，只是原封不动地记录，所以没有损失任何信息。从编程语言的角度来看，free algebra 是 AST，而其它的 algebra 是 AST 描述的语言的各种语义。给定任意一个语义，我们都可以在其中“运行”AST=free algebra。

接下来，我们来严谨地给出 free algebra 的性质。给定一个 universal algebra $\mathcal{A}$，假设其上的 free algebra 为 $F(V)$，其中 V 是变量的集合。那么，free algebra 的性质可以表述为：

$\forall A : \mathcal{A}$-algebra,	在任意具体 algebra/具体语义中
$\forall \rho : V \to A$,	只需要给每个变量在 A 中赋一个值
$\exists! h : F(V) \to A$	就能在 A 中解释/运行任意 $F(V)$ 中的“表达式”
$h = \rho$ on V	而且解释方式是由对变量的解释 ρ 唯一确定的

注意这里稍稍滥用了记号，$\rho : V \to A$ 是 V 到 A 的 carrier 的集合函数，而 $h : F(V) \to A$ 则是一个 $\mathcal{A}$-homomorphism。“h 是一个 homomorphism”这一点，保证了我们得到的解释是符合预期的：$F(V)$ 中一个 $[\mathsf{op}; a_1, \ldots, a_n]$ 的表达式，在具体的 A 中真的会被解释为 $\mathsf{op}_A(a_1, \ldots, a_n)$。此外，根据 free algebra

性质中 h 的唯一性，可以知道 h 是 $\mathcal{A}$-homomorphism 这一要求和 ρ 决定了将 $F(V)$ 中的表达式在 $\mathcal{A}$ 中进行解释时的**计算规则**。

Free algebra 的这一性质，可以利用范畴论的语言得到更简洁的表述。首先，注意到给定一个 universal algebra $\mathcal{A}$，所有的 $\mathcal{A}$-algebra 和它们之间的 $\mathcal{A}$-homomorphism 构成一个范畴。将这个范畴记作 $\mathcal{A}$-alg。首先，考虑比较简单的、没有变量的情况。这时，$V = \varnothing$，$\rho : V \to A$ 是 trivial 的。所以，free algebra 的性质简化为：

$$\forall A \in \mathcal{A}\text{-alg}, \text{存在唯一的 } F(\varnothing) \to A \text{ 的 } \mathcal{A}\text{-homomorphism}$$

用范畴论的语言来说，这正是：

$$F(\varnothing) \text{ 是 } \mathcal{A}\text{-alg 中的 initial object}$$

接下来，考虑更一般的、带变量的情况。现在，每个 $F(V) \to A$ 的函数还会附带一个集合中的 $V \to A$ 的变量替换。令 $\mathtt{Set}$ 为全体（小）集合与它们间的函数构成的范畴，为了将 $\mathcal{A}$-algebra 的范畴与集合联系起来，有一个 forgetful functor $U : \mathcal{A}\text{-alg} \to \mathtt{Set}$，它把一个 $\mathcal{A}$-algebra 中的所有结构忘掉，把 algebra 映射到 carrier set，把 homomorphism 映射到对应的函数。

利用 U，变量替换 $\rho : V \to A$ 可以表达成 $\mathtt{Set}$ 中一个 $V \to U(A)$ 的态射。Free algebra 中对应变量的构造器则可以表达成一个特殊的态射 $\eta : V \to U(F(V))$。对于每个 $h : F(V) \to A$，我们可以通过 $U(h) \circ \eta$ 来提取出它对变量的解释。利用这一点，可以将 free algebra 的性质用范畴论的语言表述为：

$$\begin{aligned}
&\forall A \in \mathcal{A}\text{-alg} \\
&\forall \rho : V \to U(A) \in \mathtt{Set} \\
&\exists! h : F(V) \to A \in \mathcal{A}\text{-alg} \\
&U(h) \circ \eta = \rho
\end{aligned}$$

事实上，有上述性质的 $F(V)$，正是范畴论中的一个（关于 U 的、V 上的）*free object*。假设 $\mathcal{C}$、$\mathcal{D}$ 是两个范畴 $U : \mathcal{C} \to \mathcal{D}$ 是一个 functor，V 是 $\mathcal{D}$ 中的一个对象，那么，$\mathcal{C}$ 中（关于 U 的）V 上的 free object 是：

- $\mathcal{C}$ 中的一个对象 X（对应 $F(V)$）加上一个箭头 $\eta : V \to U(X)$（对应变量的构造器）
- 对于 $\mathcal{D}$ 中每个箭头 $\rho : V \to U(Y)$，$\mathcal{C}$ 中有唯一的箭头 $f : X \to Y$，使得 $U(f) \circ \eta = \rho$

8 回到 ADT：fold 与 recursor

在介绍 free algebra 的性质时，我提到对于 $F(V) \to A$ 的 homomorphism，变量替换 ρ 和 $\mathcal{A}$-homomorphism 这一条件能决定它的 **计算规则**。既然提到了计算，那么这一性质是否在 A(lgebraic)DT 中也有对应呢？

首先，依然考虑 $V = \varnothing$ 的简单情况。以自然数为例，对应的 A(bstract)DT 和 A(lgebraic)DT 分别是：

$\mathcal{A}$ (**Abstract** Data Type)	$F(\varnothing)$ (**Algebraic** Data Type)

`module type Nat = sig`
`  type t`
 	
`  val zero : t`
`  val succ : t -> t -> t`
`end`
	`type nat =`
	`  | Zero : nat`
	`  | Succ : nat -> nat`

在 free algebra 的性质中，由于 $V = \varnothing$，唯一的输入是一个具体的 algebra，也就是一个 `A : Nat` 的 A(bstract)DT 实现。这当中包含的数据是：

- 一个类型 `t`
- 一个值 `zero : t`
- 一个函数 `succ : t -> t`

给定这些数据，我们希望得到一个 `nat -> t` 的函数 `f`。f 必须是 Nat-homomorphism，所以 f 的计算规则**必须**是：

```
f Zero     = zero
f (Succ n) = succ (f n)
```

这个函数可以通过模式匹配和递归定义出。但如果再仔细地看一看，就会发现 `f` 就是 `nat` 上的 recursor，或者说一个 fold！

所以说，free algebra 的性质在编程中的应用其实到处都是，只是可能没被注意到罢了：**free algebra 的性质就是 ADT 上的 structural recursion**。它在一般的函数式编程中被叫作 fold，在定理证明器中被叫作 inductive type 的 recursor。而一个普通的 algebra，正是 fold/recursor 的一组参数！

9 F-algebra：更好的定义方式

虽然我们已经定义了什么是 universal algebra，对于一个具体的 universal algebra $\mathcal{A}$，也已经能定义出所有 $\mathcal{A}$-algebra 构成的范畴等等。但 universal algebra 的定义本身还是比较零乱的：有数量不定的操作，每个都有数量不定的参数。此外，研究两个不同的 universal algebra 之间的联系也不太方便。比如说，想证明掉换操作顺序对 algebra 的性质没影响，就没有比较直接的办法。

下面，让我们尝试给出一个更简洁的 universal algebra 的定义。首先，有数量不定的操作这点很麻烦。为了合并不同的操作，我们可以借助 coproduct。Coproduct 有如下重要的性质：

$$[A + B, C] \simeq [A, C] \times [B, C]$$

其中 $[X, Y]$ 表示 X 到 Y 的函数/态射的集合。所以，假设有两个操作 $\mathtt{op}_1 : A^m \to A$ 和 $\mathtt{op}_2 : A^n \to A$，我们可以把它合并为 $\mathtt{op}_{1,2} : A^m + A^n \to A$。如此一来，一个 universal algebra $\mathcal{A}$ 中的操作就可以合并为：

$$\mathtt{op}_{\mathcal{A}} : \left(\sum_{\mathtt{op}: A^n \to A \in \mathcal{A}} A^n \right) \to A$$

注意到 $\mathtt{op}_{\mathcal{A}}$ 的左侧有非常确定的形式：它一定是由 A 通过 finite product 和 finite coproduct 生成的。因此，我们可以把不同操作的参数数量的信息通过一个 *polynomial functor* 来表示。一个 polynomial functor 就是一个通过对参数取 finite product 与 coproduct 得到的 functor：就像多项式一样，这也是它的名字的由来。如此一来，$\mathcal{A}$ 中的操作的签名就变成了：

$$\mathtt{op}_{\mathcal{A}} : P(A) \to A$$

其中 P 是一个 polynomial functor。那么，polynomial 这个条件是否是必要的呢？我们是否能直接讨论一个普通的 functor，甚至把这个 functor 的 domain 和 codomain 从 `Set` 换成一个任意的范畴呢？如果我们这么做，我们就得到了 *F-algebra* 的定义。假设 $\mathcal{C}$ 是一个范畴，F 是一个 $F : \mathcal{C} \to \mathcal{C}$ 的 functor，那么一个 F-algebra 包含：

- 一个 $\mathcal{C}$ 中的对象 A
- 一个 $\mathcal{C}$ 中的箭头 $\mathtt{op} : F(A) \to A$

对应到 universal algebra 中，范畴 $\mathcal{C}$ 是对 `Set` 的泛化，F 是签名，表达了 universal algebra 中的各个操作及它们各自的签名的信息。在一个具体的 F-algebra 中，对象 A 是对 carrier set 的泛化，$F(A)$ 可以看作是一个 (操作, 这一操作的参数) 的 tuple，而 `op` 正是操作的实现。

Universal algebra 中的各种构造都能在 F-algebra 中复刻。一个 F-algebra homomorphism $h : (A, \mathtt{op}_A) \to (B, \mathtt{op}_B)$ 是 $\mathcal{C}$ 中一个 $A \to B$ 的态射，额外使得下面的交换图交换：

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow{\scriptstyle \mathtt{op}_A} & & \downarrow{\scriptstyle \mathtt{op}_B} \\ A & \xrightarrow{f} & B \end{array}$$

这张交换图中，左下的路径对应 universal algebra 中的 $f(\mathtt{op}_A(a_1, \ldots, a_n))$，右上的路径则对应 $\mathtt{op}_B(f(a_1), \ldots, f(a_n))$。所以，这张交换图以更加简洁的方式表达了 universal algebra 中 homomorphism 的定义！

有了 F-algebra homomorphism 的定义之后，给定 functor F，全体 F 上的 F-algebra 和它们间的态射就能构成一个范畴，记作 F-alg。在 universal algebra 中，有从 $\mathcal{A}$-alg 到 `Set` 的 forgetfun functor U。而在 F-algebra 中，也有对应的 forgetful functor $U : F\text{-alg} \to \mathcal{C}$，它的定义就是忘掉操作只留 carrier：

$U(A, \texttt{op}) = A$，$U(f) = f$。

有了 forgetful functor U 之后，free algebra 也能通过范畴中的 free object 定义出了。不过，并不是所有 F-algebra 都存在 free algebra。

10 回到 PL：recursion scheme

既然 F-algebra 可以看作 universal algebra 的一种更简洁的定义方式，在 PL 中，是否也能复刻 F-algebra 的定义呢？答案是肯定的。假设我们已经有一个 functor `f` 了，也就是说，我们有：

```
type 'a f
val fmap : ('a -> 'b) -> 'a f -> 'b f
```

那么，一个以 `a` 为 carrier 的 F-algebra 就是一个 `a f -> a` 的函数。依然以自然数为例，自然数中有两个操作 `zero` 和 `succ`，将它们打包起来，可以得到如下的 functor：

```
type 'a f =
    | Zero : 'a f
    | Succ : 'a -> 'a f

let fmap (func : 'a -> 'b) (fa : 'a f) =
    match fa with
    | Zero   -> Zero
    | Succ a -> Succ (f a)
```

用更加简洁的记号写的话，就是 `f(a) = 1 + a`，其中 `1` 是 unit type。其他 ADT 对应的 functor 也可以类似地构造出。有了一个 functor `f` 之后，应该如何构造它的 initial/free algebra 呢？在 universal algebra 中，free algebra 的 AST 是由直接将操作和它们的参数打包起来构造出的。而在 F-algebra 中，“操作和它们的参数”对应的正是 $F(A)$！所以，initial algebra 的定义如下：

```
type init =
    | Op : init f -> init
```

由于这是 initial algebra，没有变量，所以只有当 `f` 中包含零元操作（常量）时，才能构造得出表达式。这也是为什么 `init` 的定义看上去没有 base case：base case 是通过“`f` 不使用它的参数”的情况，也就是常量提供的。如果把对应自然数的 `f` 代入 `init` 的定义，有：

```
type init =
    | Op : (unit + init) -> init
```

利用 coproduct 的性质拆分这个构造器，可以得到：

```
type init =
    | Case1 : unit -> init
    | Case2 : init -> init
```

不难看出，这个类型就是 `nat`。所以，`init` 的构造是与 universal algebra/ADT 的构造相吻合的。接下来，initial algebra 的性质/ADT 上的 fold 又该如何复现呢？给定一个具体的 F-algebra，也就是一个类型 `t` 和操作 `op : t f -> t`，我们只需要遍历 `init` 构成的 AST，将每个 `Op` 构造器用具体的 `op` 实现即可：

```
let rec fold (op_of_t : 't f -> 't) (expr : init) =
     match expr with
     | Op args -> op_of_t (fmap (fold op_of_t) args)
```

如果把 `nat` 对应的 `f` 代入，可以看到这里的 `fold` 和 `nat` 的 recursor 是相同的。所以，在编程中，F-algebra 同样可以复刻 universal algebra/ADT 的全部功能。而且，在有 Higher Kinded Polymorphism 的语言，例如 Haskell 中，利用 F-algebra 可以写出**对任意 ADT 都适用**的代码，这些代码还有一定停机等优秀性质。因此，有一种编程方式就鼓励使用 F-algebra 来代替递归的 ADT，使用 `fold` 来代替递归与模式匹配，这种编程方式叫作 *recursion scheme* [1]。Recursion scheme 中有许多便利的组合子用于递归遍历 initial algebra，而上面的 `fold` 就是其中最基础的一个，在 recursion scheme 的世界中被叫作 *catamorphism*。

11 从 free F-algebra 到 monad

介绍完了如何在编程语言中实现 initial F-algebra 以及其上的 fold 之后，free F-algebra 又如何呢？参照实现 free universal algebra 的思路，假设 `f` 是一个 functor，那么 `f` 上的 free F-algebra 可以如此定义：

```
type 'v free =
    | Var : 'v          -> 'v free
    | Op  : ('v free) f -> 'v free
```

Free algebra 上的 recursor 也可以轻松地定义出，只需要多要求一个`'v -> t` 的参数作为赋给变量的值即可。然而，上面的 `free` 类型很多时候并不被叫作 free F-algebra，它的一个更常见的名字是 *free monad*。

让我们回到 F-algebra 与范畴的世界中，并考虑这么一个问题：如果以另一个 free F-algebra 为目标，运用 free F-algebra 的性质，会发生什么？假设我们有两个对象 $V, W \in \mathcal{C}$，代表两个不同的变量集合。假设它们上的 free F-algebra 分别是 $\texttt{Free}(V)$、$\texttt{Free}(W)$。假如我们要对 $\texttt{Free}(V)$ 运用 free algebra 的性质、将其变换到 $\texttt{Free}(W)$ 的话，我们需要提供一个 $\rho : V \to U(\texttt{Free}(W))$ 的变量

替换。这个变量替换是一个**变量到表达式**的替换。有了这个变量替换之后，free algebra 的性质会给出一个 F-algebra homomorphism $h : \texttt{Free}(V) \to \texttt{Free}(W)$。

如果把 ρ 看作一个 substitution，把 V 和 W 看作记录了表达式中允许出现哪些变量的 context，用 $V \vdash e$ expr 来表示 e 是一个使用 V 中变量的合法表达式，也就是 $e \in \texttt{Free}(V)$ 的话，上述“free 到 free”的变换可以写成：

$$\frac{V \vdash e \text{ expr}}{W \vdash e[\rho] \text{ expr}}$$

这和编程语言/λ-calculus 中的变量替换何其相像！假设对于所有 $V \in \mathcal{C}$ 对应的 free algebra 都存在，那么 `Free` 就会成为一个 functor。这时，如果我们忘掉 `Free` 的 F-algebra 结构（也就是让它与 U 复合），抽象化地表达它“支持 substitution”的性质的话，得到的数学结构正是一个 *monad*。对于一个范畴 $\mathcal{C}$，$\mathcal{C}$ 上的一个 monad 包含：

- 一个 $\mathcal{C} \to \mathcal{C}$ 的 functor T （对应 $U \circ \texttt{Free}$）
- 对于每个对象 $V \in \mathcal{C}$，一个箭头 $\eta : V \to T(V)$ （对应 `Free` 中变量的构造器）
- 对于每个对象 $V, W \in \mathcal{C}$ 以及箭头 $\rho : V \to T(W)$，有一个箭头 $-[\rho] : T(V) \to T(W)$ （对应 `Free` 中将变量替换拓展到整个表达式上）
- 上述两个操作需要满足一些等式：

  $$-[\eta] = \texttt{id}$$
  $$-[\rho] \circ -[\rho'] = -[(-[\rho]) \circ \rho']$$

  这些等式从 substitution 的角度都很好理解。因为下面的内容不会涉及到它们，此处不做更多说明

在数学的应用中，第三个、对应 substitution 的拓展的操作 $-[\rho]$ 往往用另一个等价的操作 $\texttt{join} : T(T(V)) \to T(V)$ （V 任意）代替。此外，上述的 η、$-[\rho]$ 和 `join` 都必须是对 V natural 的。也就是说，使用一个 $V \to W$ 的“变量重命名”不会改变各个操作的语义。

12 monadic 与 algebraic effect

上面我以“表达式中的变量替换”为例引入了 monad。但对于对函数式编程、尤其是 Haskell 有所了解的人来说，“monad”最著名的应用想必是它可以用来刻画副作用。这两种解释并不矛盾，因为 monad 是一种抽象的数学结构，许多不同的具体应用可能都具有同样的结构。

要用 monad 的结构来解释 side effect 同样非常直白。只需要把 monad 的各个组分的解释更改一下可以。首先，V、W 不再被解释为变量的集合，而是被解释成**代表无副作用的值的类型**。接下来，$T(V)$ 的含义不再是“使用 V 中

变量的表达式”，而是**一个可能有副作用、运算结果类型为 V 的计算**。接下来，$\eta : V \to T(V)$ 可以改名为 `return`，表示不做任何计算，直接返回一个现成的值。$-[\rho] : T(V) \to T(W)$ 可以改名为 `let x = _ in c`，其中 $\lambda x.c : V \to T(W)$ 是一个依赖 $x : V$ 的、结果类型为 W 的计算 `c`。通过在下划线处填入一个类型为 $T(V)$ 的计算 `c0`，我们就能将 `c0` 与 `c` 按顺序执行，得到一个类型为 $T(W)$ 的计算。所以，monad 的第二个操作现在表达了将计算按顺序执行的能力。

Monad 在 Haskell 中的应用取得了很大的成功。然而，用 monad 来表达副作用也是有一定不足的。两个 monad 比较难直接组合。要用 monad 来便利地组合、处理多个不同的 effect 往往需要一些“类型体操”。因此，有另外一种性质更好的表达副作用的方式正在得到越来越多的关注，它就是 algebraic effect。

在 algebraic effect 中，每个副作用都被拆分成了两部分：操作和 handler。使用副作用的程序像调用普通函数一样调用该副作用的操作，而副作用的实现者提供一个 handler 来实现这些操作。一些常见的副作用和它们对应的操作如下：

副作用	monad	操作
异常	$T(V) = V + E$	`raise : exception -> a`
可变状态	$T(V) = S \to S \times V$	`get : unit -> state` `set : state -> unit`
输入输出	$T(V) = I \to O \times V$	`input  : unit -> input` `output : output -> unit`
不确定性	$T(V) = \mathrm{List}(V)$	`choose : 'a list -> 'a`

可以看到，虽然组合两个 monad 不那么简单，组合两套操作却非常简单：取一个并集即可。而且操作这一抽象也非常符合直觉，副作用的使用者完全可以在不知道情况下、像使用普通函数一样使用副作用。

那么，又要如何在 algebraic effect 中实现副作用呢？要实现一个副作用中的操作需要提供一个 handler，里面包含了各个操作的实现。但是，副作用毕竟不是普通函数，它们可能会改变程序的控制流，例如扔出异常就会导致接下来的程序被丢弃。所以，handler 中各个操作的实现除了能拿到调用者提供给参数的实现之外，还能拿到一个 continuation 参数，表示操作返回后，程序“剩余”的部分（以整个被 handle 的程序为界）。例如，下面是一个同时处理输入和异常两种副作用的例子：

```
handle
    let i = input () in
    if p(i)
    then raise err
    else return i
```

```
with
| value       -> Ok value
| raise err _ -> Error err
| input ()  k -> k default_input_value
```

其中，handler 的第一个分支处理的是被 handle 的计算没有触发副作用、直接返回的情况。两个处理操作的分支中，第一个参数是被提供给操作的参数，第二个参数是 continuation。当被 handle 的计算执行到 `let i = input () in ...` 时，副作用触发，handler 被调用，此时 continuation 参数 `k` 被绑定到了：`fun i -> if p(i) then raise err else return i`：整个计算剩余的部分。接下来，handler 给 `k` 提供了参数，于是 `i` 绑定到了 `default_input_value`，被中断的计算继续运行下去。

13 竟能如此 algebraic

读过了前面关于 abstract data type 与 universal algebra 的内容之后，当看到 algebraic effect 中“操作”，“操作与实现分离”的内容时，你很可能会产生一点既视感：这两者似乎有深刻的联系。是的，algebraic effect 的“algebra”，正是 universal algebra。

我们不妨直接从 universal algebra 的角度，来把 algebraic effect 重新发明一遍。首先，一个副作用对应的操作和 universal algebra 中的操作其实很像，所以我们可以把它们塞进一个 functor，做成一个 F-algebra。但是，这里需要做出第一个改动：在上面的例子中的操作签名里，没有 functor 的参数，也就是 carrier 的位置。对于副作用来说，carrier 是“结果值的类型”。所以，我们可以通过把“整个计算剩下的部分”塞入操作中，来得到一个 functor，例如：

副作用	原操作	对应的 functor

异常	`raise : exception -> 'a`	`type 'result op =`
		`  | Raise : exception`
		`  -> ('a -> 'result)`
		`  -> 'result op`

输入输出	`input  : unit -> input`
	`output : output -> unit`
		`type 'result op =`
		`  | Input  : (input -> 'result)`
		`  -> 'result op`
		`  | Output : output`
		`  -> (unit -> 'result)`
		`  -> 'result op`

不确定性	`choose : 'a list -> 'a`	`type 'result op =`
		`  | Choose : 'a list`
		`  -> ('a -> 'result)`
		`  -> 'result op`

一般地，每个副作用的操作 `op : arg_type -> result_type`，在 functor 中都会对应一个构造器 `Op : arg_type -> (result_type -> 'a) -> 'a op`。如此一来，每个副作用都能对应一个 functor、一个 universal algebra 了。那么，一个 handler 对应什么呢？由于“程序剩余的部分”、continuation 已经被打包进了 functor 中，所以一个 handler 刚好就是一个具体的 F-algebra：各个操作的实现被打包在了 $F(V) \to V$ 的箭头中。

接下来，使用副作用的表达式又是什么呢？考虑到它们是“表达式”，而且可以在任意一个 handler（具体的 algebra）中得到解释，答案已经呼之欲出了：使用副作用的表达式就是 free F-algebra （定义见 11）！Free F-algebra 又名 free monad，所以 free F-algebra，也就是使用副作用的表达式，支持像 monad 一样的“顺序执行”！而在 handler 中解释一个表达式，正是对 free F-algebra 性质的应用！下面的表格展示了 algebraic effect 于 F-algebra 间漂亮的一一对应：

Algebraic Effect		F-algebra	
对象	意义	对象	意义
`type 'r op`	副作用中的操作	functor F	algebra 中的操作
`handler : r op -> r`	一个副作用的 handler	$(A, h : F(A) \to A)$	一个具体的 F-algebra
`'r op` 上的 free F-algebra	使用副作用的表达式	$\texttt{Free}(V)$	free F-algebra
`handle ... with ...`	使用副作用的表达式可以在任意 handler 中解释	$h : \texttt{Free}(V) \to A$	free F-algebra 的性质
`let x = c1 in c2`	使用副作用的表达式可以按顺序执行	$\texttt{Free}(V)$ 上的 monad 操作	free F-algebra 是 monad

14 algebra 与 monad

通过 monad 和通过 F-algebra，分别能得到两种不同的表达副作用的方式。而且，free F-algebra 又刚好是 *free* monad，而不是随便一个普通的 monad。那么，一个很自然的问题是：monad 与 F-algebra 之间有什么联系呢？

为了回答这个问题，首先不妨从对“free monad”这个词的拆解开始。我们已经知道了 free F-algebra 的含义是什么，但是，monad 并不是一种 algebra，要如何定义一个 free monad 呢？答案是，monad 其实可以是一个 F-algebra。在之前的例子中，F-algebra 使用的范畴 $\mathcal{C}$ 往往是 `Set` 或类型的范畴。但是，如果把 $\mathcal{C}$ 换成一个**以 functor 为对象的范畴**，就可以表达**带参数的 algebra**。而 monad 就可以写成这么一个带参数的 algebra：

```
module type Monad = sig
    type 'a t
    val return : 'a       -> 'a t
    val join   : ('a t) t -> 'a t
end
```

由于这里的操作 `return` 和 `join` 都是参数化的，我们可以把它们看作两个 functor 之间的自然变换：

$$\eta : \mathtt{id} \to T$$
$$\mu : T \circ T \to T$$

这里，η 和 μ 是范畴论中对 `return` 和 `join` 的命名。令 $\mathcal{C}$ 是一个范畴，用 $[\mathcal{C},\mathcal{C}]$ 来表示 $\mathcal{C} \to \mathcal{C}$ 的 functor 和它们之间的自然变换构成的“自函子范畴”，那么 monad 的两个操作就可以被打包在一个 $[\mathcal{C},\mathcal{C}] \to [\mathcal{C},\mathcal{C}]$ 的 functor 中，从而形成一个 $[\mathcal{C},\mathcal{C}]$ 上的 F-algebra：

$$M(T) = \mathtt{id} \times (T \circ T)$$

现在，一个 $\mathcal{C}$ 上的 monad 就是一个 M-algebra。接下来的事情，就和一个普通的 F-algebra 没有任何区别了：我们可以得到一个 forgetful functor $U : M\text{-alg} \to [\mathcal{C},\mathcal{C}]$，它提取出一个 monad 中的 functor。我们可以定义出 free M-algebra，也就是 free monad。而，巧合的是，functor F 上的 free M-algebra（这是一个具体的 free object），正是 functor `Free`，它把**每个** $\mathcal{C}$ 中的对象 V 送到它上面的 free F-algebra，是一个 free functor（而不是一个具体的 F-algebra 中的 free object）。所以，每个 free monad 都给出了对应的 functor F 的 free F-algebra 的一个一般的构造方法。

15 monad 与 algebra，在 PL 中

通过 free F-algebra 也是 free monad 这一结论，我们可以得到 free F-algebra 的另一层性质：

> 对于每个 monad T 与自然变换 $F \to T$，存在一个唯一的 monad 自然变换 `Free` $\to T$。其中 `Free` 是 free F-algebra functor

如果把这条性质代入副作用的例子中的话，就是：

> 对于每个 monad `'a t` 与函数 `handler : 'a op -> 'a t`，有一个唯一的函数 `handle : 'a free -> 'a t`，其中`'a free` 是`'a op` 的 free algebra

换言之，只要我们提供了副作用的操作在一个 monad 中的解释（`handler`），就能将任意使用该副作用的程序通过该 monad 实现（`handle`）！这说明，monad 其实是副作用的**实现**，而在 algebraic effect 的框架中，使用副作用的程序是对操作不做任何解释、原封不动记录的符号表达式，而我们可以用一个 monad 来对操作进行解释，从而恢复 monadic effect！

16 adjunction

Monad 与 algebra 的联系，在范畴论中同样是一个重要结论。只不过，它往往是借助另外一个概念被呈现的：adjunction。其实，本文的前面已经出现过许多个 adjunction 了，例如 free algebra functor `Free` 和 forgetful functor U 就是

一对 adjunction。

对于范畴 $\mathcal{C}$ 中的一个具体对象 V，我们可以定义它在 U 下的 free object。而如果对于 $\mathcal{C}$ 中的每个对象，都存在对应的 free object，这些 free object 就可以被打包为一个 functor $\texttt{Free} : \mathcal{C} \to F\text{-alg}$。我们知道，每个 $h : \texttt{Free}(\texttt{V}) \to A$ 的 homomorphism，都是唯一地由它对 V 中变量的解释 $U(h) \circ \eta$ 确定的。而每个变量替换 $\rho : V \to U(A)$，也都会对应唯一的 homomorphism h。所以，Free 和 U 的这种联系可以写作：

$$F\text{-alg}(\texttt{Free}(V), A) \simeq \mathcal{C}(V, U(A)) \qquad \forall V \in \mathcal{C}, A \in F\text{-alg}$$

换言之，对于任意的 $V \in \mathcal{C}$、$A \in F$-alg，范畴 F-alg 中 $\texttt{Free}(V) \to A$ 的箭头（homomorphism）和范畴 $\mathcal{C}$ 中 $V \to U(A)$ 的箭头（变量替换）是一一对应的。如果这些一一对应对于 V、A 是 natural 的，也就是说它们的语义不会对特定的 V 或 A 做特殊处理，那么此时 Free 和 U 就构成一对 *adjunction*。给定范畴 $\mathcal{C}$、$\mathcal{D}$，它们间的一对 adjunction 是：

- 两个 functor $F : \mathcal{D} \to \mathcal{C}$、$U : \mathcal{C} \to \mathcal{D}$
- 对于每个 $A \in \mathcal{C}$、$B \in \mathcal{D}$，一个箭头间的一一对应

  $$\mathcal{C}(F(A), B) \simeq \mathcal{D}(A, U(B))$$

  这些一一对应对 A、B natural

F 和 U 构成一对 adjunction 记作 $F \dashv U$，其中 F 称为（U 的）left adjoint，其中 U 称为（F 的）right adjoint。Adjunction 有多种等价的定义方式。在 free algebra 的例子中出现的“变量构造器”η 可以从一一对应中导出：只需要令 $B = F(A)$，对 $\texttt{id}_{F(A)}$ 运用一一对应即可。在 adjunction 中，η 称为 adjunction 的unit，类似地，可以定义出一个箭头 $\epsilon : F(U(B)) \to B$，称为 adjunction 的 counit。

有了 adjunction 之后，F-algebra 的许多属性可以得到更简洁的表述。例如，“对任意对象都存在 free algebra”可以表述为 forgetful functor U 有一个 left adjoint F，也就是 free functor。而当 free functor F 存在时，A 上关于 U 的 free object 就是 $F(A)$。

而 monad 和 algebra 的联系，用 adjunction 同样可以得到更一般的刻画。Free F-algebra/free monad 是通过 $U \circ \texttt{Free}$ 得到的。而事实上，对于**任意一对** adjunction $F \dashv U$，$U \circ F$ 都是一个 monad。反过来，每个 monad 也都有多种（至少两种）方式可以拆分成 adjunction。

17 将 adjunction 用于 PL：CBPV

Adjunction 看上去是个非常好的概念：它可以同时用来刻画 F-algebra 和 monad。那么，既然 F-algebra 和 monad 都能用于描述副作用，adjunction 是否也能用于导出一种更一般的描述副作用的方式呢？

为了在编程语言中复现出 adjunction，首先需要考虑范畴 $\mathcal{C}$、$\mathcal{D}$ 分别是什么。参照 algebraic effect，free functor（left adjoint）F 的 domain $\mathcal{D}$ 应该代表**值类型**的范畴 $\mathcal{V}$。而它的 codomain 中装着使用了副作用的表达式，所以 $\mathcal{C}$ 应该是**计算类型**的范畴 $\mathcal{C}$。可以看到，通过 adjunction，我们得到了一种新的语言模型：在这种新语言模型里，**值**和**计算**是独立的两种东西，并分别有不同的类型。接下来，我将会用字母 $ABCXY$ 指代值类型和计算类型，但在计算类型下加一条下划线已示区分。

有了范畴 $\mathcal{C}$、$\mathcal{D}$ 后，左右 adjoint F 和 U 的含义又是什么呢？F 的含义在 algebraic effect 的例子中已经很清晰了。对于一个值类型 A，$F(A)$ 是“结果类型为 A 的计算”。那么，$U : \mathcal{C} \to \mathcal{V}$ 的含义又应该是什么呢？从签名上看，它把计算送到值。因此，可以大胆猜测 U 的功能就是“延迟”一个计算，把它裹在一个 closure/thunk 中，留待以后再触发。在表达式层面，也有对应的操作 `thunk` 和 `force`，用于延迟/触发计算：

$$\frac{\Gamma \vdash M : \underline{B}}{\Gamma \vdash \texttt{thunk}\ M : U(\underline{B})} \qquad \frac{\Gamma \vdash V : U\underline{B}}{\Gamma \vdash \texttt{force}\ V : \underline{B}}$$

下一个问题是，哪些类型和表达式应该是值、哪些类型和表达式应该是计算呢？在这个问题上，有一门叫作 Call By Push Value（CBPV）的演算 [2, 3] 给出了非常有趣的答案：值类型是 tuple、sum 和被延迟的计算的类型，值表达式是**变量**、tuple 和 sum 的构造器以及被延迟的计算。计算类型有结果为某个值的计算以及**函数类型**，计算表达式包括直接 `return` 一个值、λ 函数、函数调用，以及具体的各种副作用对应的操作。

为什么 CBPV 要做出这种选择呢？尤其是，为什么函数被归到了计算类型中呢？首先，CBPV 要求变量的类型必须是值，这保证了副作用一定是通过显式的操作触发的，变量可以被当成值到处使用。接下来，通过将 sum type 归入值中，CBPV 规避了 Call By Name + 副作用 + sum type 时的一些不好行为：在 CBN + 副作用 + sum type 的组合下，对 sum type 进行模式匹配又可能触发副作用，从而导致 sum type 的 η 等价 `f e = match e with Inl x -> f (Inl x) | Inl y -> f (Inl y)` 失效：考虑 `e` 不停机、且 `f` 不使用它的参数的情况。因为 CBN 只在必要时才会求值，等式左侧的参数 `e` 不会被求值，所以 `f e` 会停机。然而，等式右侧的求值需要进行模式匹配，因此 `e` 必须被求值，等式右侧不停机，等式两侧不等价。

接下来，CBPV 中把函数归到了计算中去。这意味，一个类型为 $A \to \underline{B}$ 的**计算**中的副作用 **只能通过提供参数来触发**。于是，这么一来，就能在有副作用的情况下恢复函数的 η 等价 `f = fun x -> f x`。这一等价在 Call By Value + 副作用的情况是不成立的：考虑 η 等价左侧的 `f` 不会停机的情况，此时右侧的 `fun x -> f x` 会停机，从而导致等式两侧不等价。

所以，可以看到，CBPV 中的各种类型都有非常良好的性质，可以说是集合了 CBV 和 CBN 两种求值顺序的长处。这也导致 CBPV 在定义语义时比 CBV 和 CBN 更加简单。此外，CBPV 中的求值顺序是固定的：要使用一个结果为 A 的计算 $M : F(A)$ 的结果，就必须通过顺序求值的构造 `let x =`

M `in` ... 对 M 进行求值。副作用可能被触发的位置也可以通过语法直接判断出。然而，通过插入适当的 `thunk` 和 `force` 来延迟/触发计算，利用 adjunction $F \dashv U$ 在值和计算之间来回，CBPV 可以完全模拟 CBV **和** CBN 的求值顺序。所以，正如 CBPV 的作者所说，CBPV 是一种“subsuming paradigm”。

到目前为止，一切看上去都很美好。然而，为了让 CBPV 和 adjunction 完全匹配上接下来就需要做出一些改动了。首先，为了利用 adjunction 来描述 CBPV 的语义，需要考虑 open term 的情况。对于一个 open value $\Gamma \vdash V : A$，它可以在范畴中解释为一个 $\mathcal{V}(\Gamma, A)$ 的箭头。但一个 open computation $\Gamma \vdash M : \underline{B}$ 却不能解释为一个 $\mathcal{C}$ 中的箭头，因为它依赖的 Γ 是 value 而不是 computation。

18 oblique morphism

既然计算表达式不能表达为计算的范畴 $\mathcal{C}$ 中的箭头，应该如何表达它呢？为此，我们可以借助一个叫作 *oblique morphism* 的概念。在 adjunction 中，有一个一一对应 $\mathcal{C}(F(A), \underline{B}) \simeq \mathcal{V}(A, U(B))$。如果我们想要表示这组一一对应中的一个箭头，既可以使用一个 $\mathcal{C}$ 中的箭头，也可以使用一个 $\mathcal{V}$ 中的箭头。不过，这两种方法都不太对称。实际上，还有另一种对称的表示方法。我们可以要求对于每个 $A \in \mathcal{V}$、$\underline{B} \in \mathcal{C}$，有一个集合 $\mathcal{O}(A, \underline{B})$，称为 *oblique morphism*。接下来，我们通过要求**两个**一一对应：

$$\mathcal{C}(F(A), \underline{B}) \simeq \mathcal{O}(A, \underline{B}) \simeq \mathcal{V}(A, U(\underline{B}))$$

就可以恢复 $F \dashv U$ 的 adjunction。注意到 $\mathcal{O}(A, \underline{B})$ 正好就是一个从值类型 A 到计算类型 $\underline{B}$ 的“箭头”的集合！所以，我们可以利用 oblique morphism 来定义值范畴和计算范畴之间的 adjunction，如此一来，oblique morphism 就能作为“计算表达式”的语义。

那么，adjunction 的两个一一对应的含义又是什么呢？首先看 $\mathcal{O}(A, \underline{B}) \simeq \mathcal{V}(A, U(\underline{B}))$，它正是 `thunk` 与 `force`：

$$\mathcal{O}(\Gamma, \underline{B}) \underset{\texttt{force}}{\overset{\texttt{thunk}}{\rightleftarrows}} \mathcal{V}(\Gamma, U(\underline{B}))$$

$$\Gamma \vdash M : \underline{B} \xrightarrow{\texttt{thunk}} \Gamma \vdash \texttt{thunk}\ M : U(\underline{B})$$

$$\Gamma \vdash \texttt{force}\ V : \underline{B} \xleftarrow[\texttt{force}]{} \Gamma \vdash VM : U(\underline{B})$$

Adjunction 的另一侧，$\mathcal{C}(F(A), \underline{B}) \simeq \mathcal{O}(A, \underline{B})$，又是什么呢？为此，首先我们需要考虑一个先前被忽略的问题：计算的范畴 $\mathcal{C}$ 中的箭头是什么？

如果箭头的 domain 是 $F(A)$，一个产生值的计算，那么事情很好办。一个 $\mathcal{C}(F(A), \underline{B})$ 的箭头就是一个依赖于 $x : A \vdash M : \underline{B}$ 的计算。假如我们有

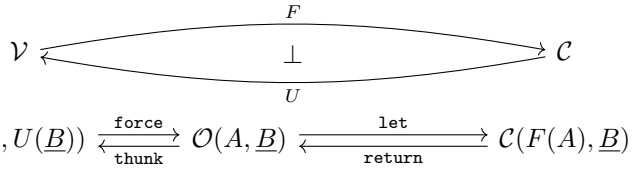
$F(A) \xrightarrow{M} F(B) \xrightarrow{N} \underline{C} \in \mathcal{C}$，那么箭头的复合 $N \circ M$ 也可以定义成：`let x = M in N` 的顺序执行。但是，当箭头的 domain 不是 $F(A)$，而是其他计算类型，例如函数时，情况就变得难办起来。这也是 CBPV 的理论中最为尴尬的一个美中不足。

对于这个问题，有两种不完美的解决办法。其一，是忠实于语言的语法，只允许 $\mathcal{C}$ 中箭头的 domain 形如 $F(A)$。但是，这么一来，$\mathcal{C}$ 就不再是一个范畴了，需要用一套不同的理论去定义、研究。第二种解决办法，则是人为引入一种新的语法构造来充当 $\mathcal{C}$ 中的箭头。但是，这种新构造从语言和 operational semantic 的角度来看是没有必要的。

这里，简单起见，我将采取第二种解决办法：人为引入两个计算类型之间的箭头。从一个计算类型 $\underline{B}$ 到另一个计算类型 $\underline{C}$ 的箭头，可以定义为一个带洞的计算表达式。例如，一个从 $F(A)$ 到 $\underline{B}$ 的箭头可能形如 `let x = _ in M`，其中 `_` 表示表达式中的洞，$x : A \vdash M : \underline{B}$。一个从函数类型 $\vec{A} \to F(B)$ 到 $\underline{C}$ 的箭头，则可能形如 `let x = _` $\vec{V}$ `in M`，其中 $\vec{V} : \vec{A}$，$x : B \vdash M : \underline{C}$。箭头的复合，则可以通过“填洞”来实现。

定义出两个计算类型之间的箭头后，我们终于可以开始考察 adjunction 的另一侧了：$\mathcal{C}(F(A), \underline{B}) \simeq \mathcal{O}(A, \underline{B})$。从左到右看，对于一个带洞的计算 $M[] : F(A) \to \underline{B}$，我们可以通过 `return` 来得到 $x : A \vdash M[\texttt{return}\ x] : \underline{B}$。从右到左看，给定一个计算 $x : A \vdash M : \underline{B}$，我们可以通过顺序执行来得到 `let x = _ in M` $: F(A) \to \underline{B}$。所以，adjunction 的另一侧，就是 `return` 和顺序执行！于是，最终我们得到的 CBPV 的 adjunction 语义的架构就是：

值类型与值　　计算表达式　　计算类型与带洞的计算表达式



19 其实，刚刚那个语义是错的

刚刚呈现的、基于 adjunction 的 CBPV 语义，虽然非常接近 CBPV 原文中的 adjunction model，但是它简化了一个地方：计算的范畴中的箭头，也就是带洞的计算表达式，不能依赖于 context。这意味它们中不能有 free variable，也意味着在 `let` 和 `return` 的操作中，context 中只能有一个变量。为了解决这个问题，需要使得 $\mathcal{C}$ 成为一个 *locally $\mathcal{V}$-indexed* 的范畴。相关的定义也会变得更复杂。感兴趣的读者可以去阅读 CBPV 原文。

虽然上面给出的语义是错、或者说过度简化的，我认为它能凸显出 CBPV 以及它的 adjunction model 中最优雅的部分，而更细节的处理则超出了本文的目标范围。所以，我选择了呈现这个过度简化的错误语义。

20 总结

最初，我们复习了 **Abstract** Data Type

我们从它当中观察到了与抽象代数相似的结构，并引出了 universal algebra 的定义

通过应用 universal algebra 中 homomorphism 的定义，我们得到了在同一个 ADT 的两个不同实现间切换的方法

从 universal algebra 和 ADT 中都存在的“表达式”的概念，我们引出了 free algebra

但其实，编程语言中的 **Algebraic** Data Type 比数学中的集合语言更适合描述 free algebra，因为 free algebra 其实就是 AST。于是，我们用 A(legbraic)DT 对各种 free algebra 进行了描述

我们回到数学中，研究了 free algebra 的重要性质。我们引入了范畴论的语言来简化表述，并引入了 free algebra 的特例 initial algebra，它在编程中非常常见

将得到的 free algebra 的性质应用于编程，我们发现，两种 ADT 有非常深刻的联系。A(lgebraic)DT 就是 initial A(bstract)DT。它们之间可以通过 fold/recursor 联系起来，这对应 initial/free algebra 的性质

通过范畴论的语言，我们给出了 F-algebra 的定义，它是一种更简单的定义 universal algebra 的方式。我们在 F-algebra 中复刻了 universal algebra 中的一些定义

我们回到 PL，尝试在编程语言中实现 F-algebra。由此，我们得到了一种非常强大的编程方法：recursion scheme。它可以用来写性质优良、对各种递归类型都适用的程序

介绍 recursion scheme 时探讨的主要是 initial F-algebra，而 free F-algebra 有另一个名字：free monad。通过 free F-algebra 上可以进行“表达式的替换”的特点，我们引出了 monad 这一重要的数学结构

Monad 在编程中最有名的应用，当然就是表示副作用了。我们复习了基于 monad 的副作用，以及比 monad 更灵活、更易组合的另一种副作用表示法：algebraic effect

我们看到，algebraic effect 是名副其实的，因为它与 universal algebra 有着非常漂亮的一一对应

Monad 与 algebra 都能用于表示副作用。于是，我们考察它们在数学上的联系。我们发现，monad 也可以表示成一个 F-algebra，而 free monad 的正是 free F-algebra 的构造方法

我们把 monad 与 free F-algebra 的联系应用到 PL 中，并由此发现了 algebraic effect 与 monadic effect 之间深刻的联系

通过引入 adjunction 的概念，我们进一步研究了 monad 与 algebra 的关系

我们尝试将 adjunction 也用于表示副作用，并由此引出了 CBPV 演算，介绍了它的设计思想与优良性质

我们引入了 oblique morphism 的概念，解决了如何在范畴中表示计算表达式的问题，并由此导出了 CBPV 的一个（过度简化、错误，但很漂亮的）范畴语义

References

[1] E. Meijer, M. Fokkinga, and R. Paterson, “Functional programming with bananas, lenses, envelopes and barbed wire,” in *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, (Berlin, Heidelberg), p. 124–144, Springer-Verlag, 1991.

[2] P. B. Levy, "Call-by-push-value: A subsuming paradigm," in *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, (Berlin, Heidelberg), p. 228–242, Springer-Verlag, 1999.

[3] P. B. Levy, "Call-by-push-value." `https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf`.