ADT是个啥

初始代数语义下代数数据类型的结构与解释 -- 图解

我以网名担保不出现超过中学算术的公式。

Algebraic Data Type

起源于HOPE语言 (1980 ACM Lisp Conference)

发扬于ML (Meta Language)

目测很像算术

Numbers	Types
0	`Void`
1	`()`
a + b	`Either a b = Left a | Right b`
a * b	`(a, b)` or `Pair a b = Pair a b`
2 = 1 + 1	`data Bool = True | False`
1 + a	`data Maybe = Nothing | Just a`

类型算术

实际上，它的确存在着某种像算术一样的性质

让我们来动手试试吧

0/1之间

Void类型没有可用的值(从值的数量上看它应该是0)

```
Prelude Data.Void> :i Void
data Void
        -- Defined in ‘Data.Void’
```

它的定义不可谓不简单

空元组在haskell中通常读作unit，写作()

它仅有一个值

```
Prelude> :i ()
data () = ()
        -- Defined in ‘GHC.Tuple’
```

非此即彼

haskell使用严格的布尔类型

```
Prelude> :i Bool
data Bool = False | True
        -- Defined in ‘GHC.Types’
```

它所表示的数是2，因为它有True和False俩个值

|是什么

首先排除汉字gun(丨)

然后排除命令行管道

它起到的作用是分隔集合元素

想象一下，类型Bool是个集合，而False和True是它里面蕴含的元素

乘法

简单的2元组担任乘法的职责

```
Prelude> :t ((), True)
((), True) :: ((), Bool)
Prelude> :t ((), False)
((), False) :: ((), Bool)
```

显然，1 * 2 = 2，所以((), Bool)这个类型有俩个值

加法

虽然|已经把加法的活干了，但是haskell还特地定义了Either

```
Prelude> :i Either
data Either a b = Left a | Right b
        -- Defined in ‘Data.Either’
```

非常直觉

```
Prelude> :t (Left "foo")
(Left "foo") :: Either [Char] b
Prelude> :t (Right False)
(Right False) :: Either a Bool
```

函数类型

以() -> Bool这个函数类型为例，它表达的是Bool ^ () => 2 ^ 1 = 2

```
Prelude> :t \() -> False
\() -> False :: () -> Bool
Prelude> :t \() -> True
\() -> True :: () -> Bool
```

这个结论看起来稍微有点怪异

但是仔细想想

我们可以把函数转换为打表，就像指数运算可以转化为大量的乘法

代码即数据

没有了

多乎哉？不多矣！

ADT是崇尚引用透明的

我们已经见到了代数数据类型的基本单元

其他复杂的类型大可通过它们合成

等等。递归呢?

它可以很直观地表达

```
Prelude> data Nat = Z | S Nat
Prelude> :t S (S Z)
S (S Z) :: Nat
Prelude> data List a = Nil | Cons a (List a)
-- f(a) = 1 + a * f(a)
```

但是在更贴近理论时，我们会谈到一个叫做不动点的玩意。

微分

我们有能力做加和乘

也可以定义递归

那么能求导吗?

Zipper

对代数数据类型的求导会得到著名的函数式抽象 -- The Zipper

它常被用于遍历复杂的数据结构

和微积分中的导数相同，对类型求导的结果描述了原本结构的局部特征

你也许会需要：haskell趣学指南 第14章

但是求导过程中出现了-和除

可能已经有人手快算了几个例子

别急，其实-和/在类型上有对应的概念和相关研究

它们有意义，不过暂时还没看出来有什么用例

简单的类比缺乏说服力

为什么类型居然能求微分

(如果有人手快，他可能已经发现Leibniz rule和chain rule也能验证)

这背后藏着什么?

Ghost in the shell

我们需要引入所谓的范畴论

它是初始代数语义的基石

语义是什么?

一般来说，我们认为编程语言的语义就是这门语言是如何被执行的。 但是，这只是语义中的一种：Operation Semantic（操作语义？）。 操作语义就是通过一系列规则描述语言中的表达式如何一步步求值， 也就是用逻辑规则“实现”了一个解释器来对表达式求值。 然而，在编程语言的研究中，还有另外一种很重要的语义描述方式： Denotation Semantic（指称语义？）。 指称语义中，我们并不描述表达式是如何求值的， 而是将表达式、类型等等在某种数学模型下赋予意义。 换言之，指称语义把语言中的表达式“编译”成了数学模型中的对象。 那么，这有什么意义呢？ 我们把C编译成汇编，是因为汇编更容易在机器上执行。 而我们通过指称语义把语言编译成数学对象，是因为通过已经被详细研究过的已知数学对象，可以更好地研究语言的性质。

-- 游客账户0x0 : 推导Simply Typed Lambda Calculus的范畴论语义

都市传说

haskell并非一种崇尚范畴论的语言，实际上，它在ML语言家族里的兄弟，OCaml，其名称中的“Caml”是

Categorical Abstract Machine Language

的缩写。

谁更热爱理论?

我仅站在个人角度呼吁，拒绝一些完全罔顾事实的偏见，*haskell是完全的工业语言*。

鱼梯

美籍作家罗格纳.本森在他的一本技术性手册中提到，他认识一个在俄勒冈州渔业部门工作的年轻人，热衷于谈论硝基甲烷。

在他的工作中常常需要用它和粉状硝酸铵混合去构建鱼梯(?)。

好吧，其实就是起爆

大体上说来，鱼梯是一种帮助鱼类洄游的设施，是对兴建水库，水坝等设施的补偿。

转向理论

现在假定有台阶A B C，它们在同一条鱼梯上从上到下排布

很自然地，我们可以把水从A流到B画成纸上从符号A到B的箭头，写出来大概是A -> B。

那么很自然地，有A -> A, B -> B, C -> C

因为水从某个台阶上流过可以看作水从它流向自身。

A -> B 和 B -> C, 也是天经地义的事, 水往低处流嘛。

等等，我们遗落了什么?

水从A流向B，又从B流向C，从规则上看我们可以增加一条描述: A -> C

结合性

当我们拿到一对头尾相连的箭头，总是可以把它们组装成一个箭头

别忘了, 我们其实是在谈论鱼梯上的水流。

但是此时我们几乎离开了水

也不在乎鱼梯对鱼到底有没有用

我们开始只关注一些脱离现实世界复杂度的**抽象性质**

暂时回到现实

不管怎么样，一条河有俩条岸，建2条平行的鱼梯当然可行

那么，平行鱼梯上的台阶，它们之间能画箭头吗？

在现实中和我讨论这个问题，我大概会使劲拿水循环来抬杠

但是在这里，我们要为模型的简单做考虑

所以答案是：没有

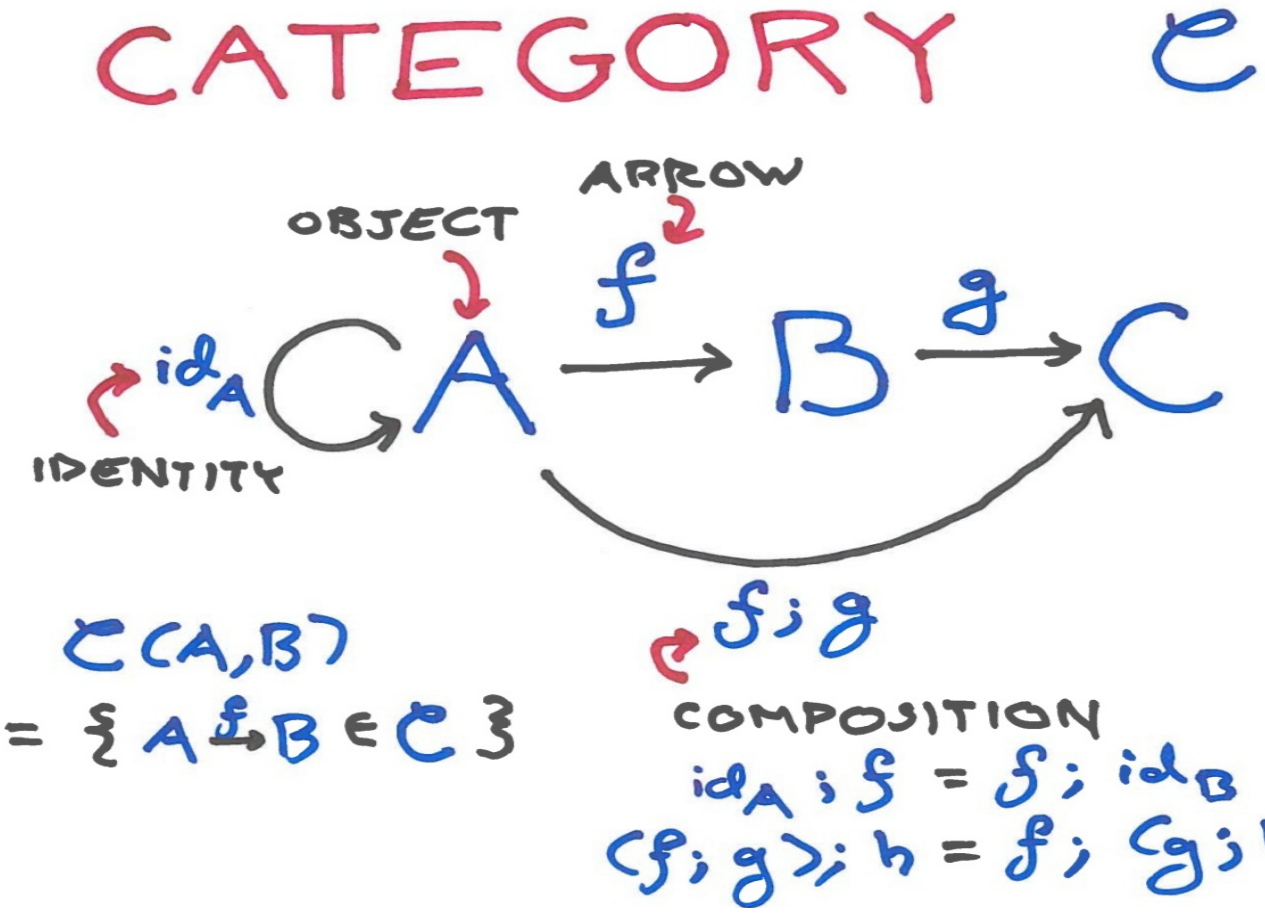
重新观察结合性

单鱼梯模型中所有台阶都相连的事实稍微有点误导性

实际上，结合性只意味着可以从已有的箭头组合出新箭头

而不强制要求某个地点所有鱼梯台阶之间都有箭头连接

范畴


CATEGORY C
OBJECT
ARROW
f
g
A
B
C
id_A
IDENTITY
C(A,B)
= { A →f B ∈ C }
f;g
COMPOSITION
$id_A ; f = f ; id_B$
$(f;g);h = f;(g;h)$

那就是范畴的定义吗

是的。

鱼梯台阶 => 对象 (Object)

箭头 => 态射 (morphism || arrow)

恒等态射和结合性让范畴具有特定结构。

但是鱼梯听起来一点也不酷

那么把鱼梯当作一个个偏序集吧。

偏序集与幺半群

范畴这一结构便是由它们俩合成的

恒等态射(id)和任意其他态射f连接，其结果总是等价于f(幺半群)

不是所有对象之间都有箭头,但是已经存在的箭头一定可结合/传递(偏序集)

Category, Revisited

严肃讲起来范畴有俩个等价的基础定义

各路教材为了方便引入可能还会有其他奇怪的定义(比如从图论引入)

既然这只是一个“引起兴趣”的分享，就跳过吧

对此感兴趣可以参考ncatlab.org提供的大量资源

Set category

集合与全函数构成的范畴

亦是Haskell壬主要居住区

(现在可别说什么bottom和unsafe煞风景)

Hom集

在Set范畴中任意对象A，B之间的全体态射也构成一个集合

一般写作Hom(A, B)

对应到haskell，就是函数类型

```
A -> B
```

有些范畴中任意一对对象之间的态射不构成集合

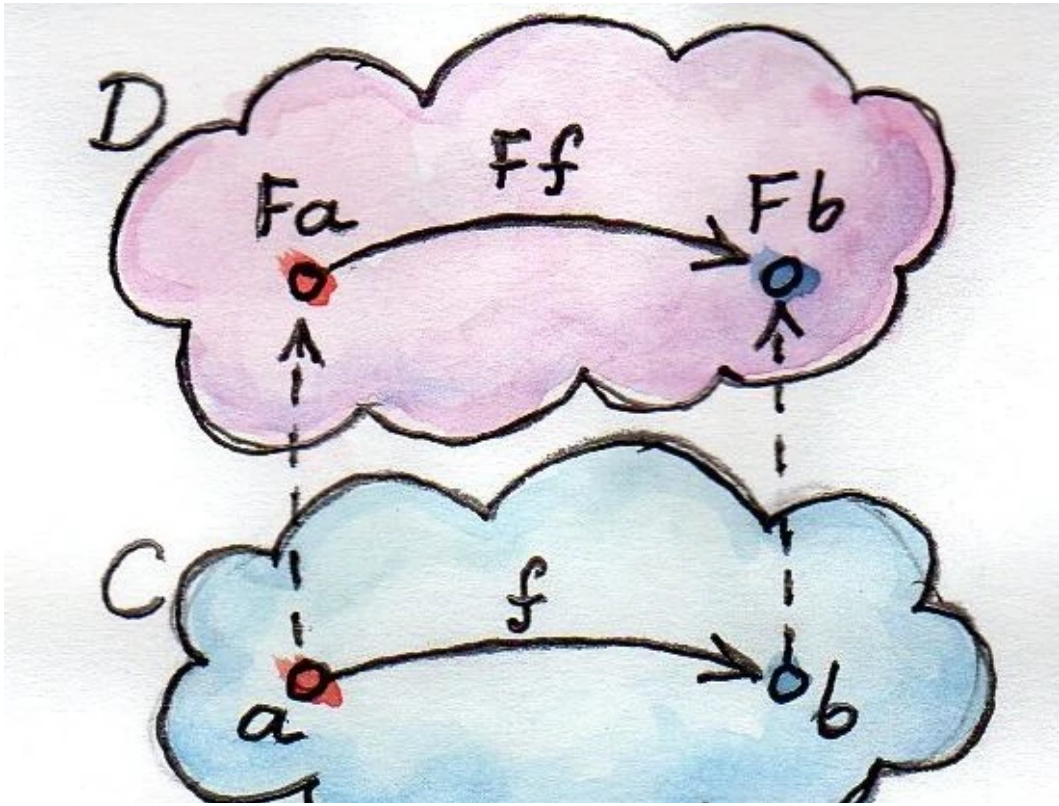
好在那不会出现在这里

你会看见的另一种说法：这个范畴是locally small的

函子

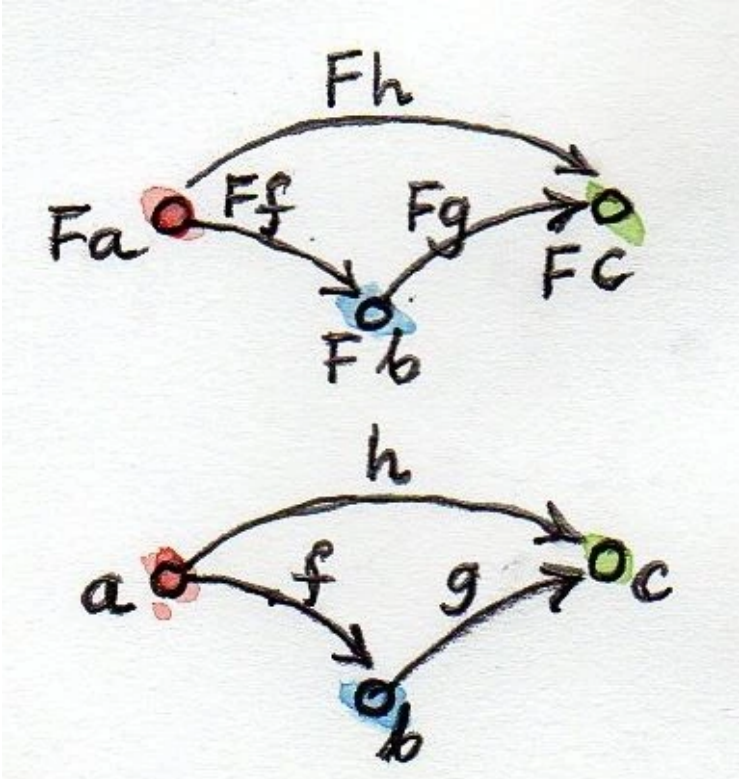
把范畴想象成一张网

函子的工作是不切断其中的线，通过卷起来或者拉伸的方法把它投影到另一张网上



分治

函子需要分别处理范畴中的对象和态射,但大多数时候重点在态射上。对于原范畴中的恒等态射，它应该被投影到目标范畴中的恒等态射上,而原范畴中头尾相连的态射在目标范畴中仍然应该头尾相连。可以看出，范畴论重视对象间关系胜过具体对象



class Functor

具体到haskell中稍微有点不同

haskell中typeclass Functor所定义的函子

其实是从Set到Set自身的自函子(EndoFunctor)

haskell只是参考了范畴论的很多概念

而非严格对应

fmap

fmap函数是函子处理态射的那一部分

从签名就能看出来了

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

当我们拿到原本范畴中的一个态射,fmap将它转换为目标范畴中的一个态射

当然了

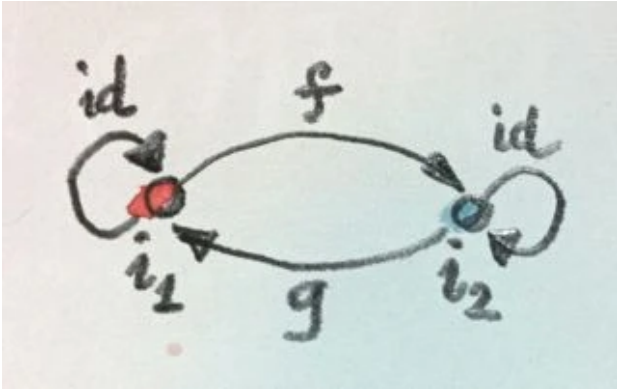
```
fmap id == id
-- id这个函数是haskell标准库的一部分
```

这里本来应该有一些例子

但是我想这是haskell书都会写的内容

请看haskell趣学指南的第11章吧!

同构



当我们可以在俩个对象之间来回穿梭而不伤身体--

其实就是上图中的一对态射可以“无损”地做转换

它们的组合可分别得到俩个对象上的恒等态射。

在haskell中，这意味着俩个类型的值数量一致

当涉及可数无穷时，事情会变得很怪异

泛泛而谈

实际上，同构是一种所谓的**泛构造**

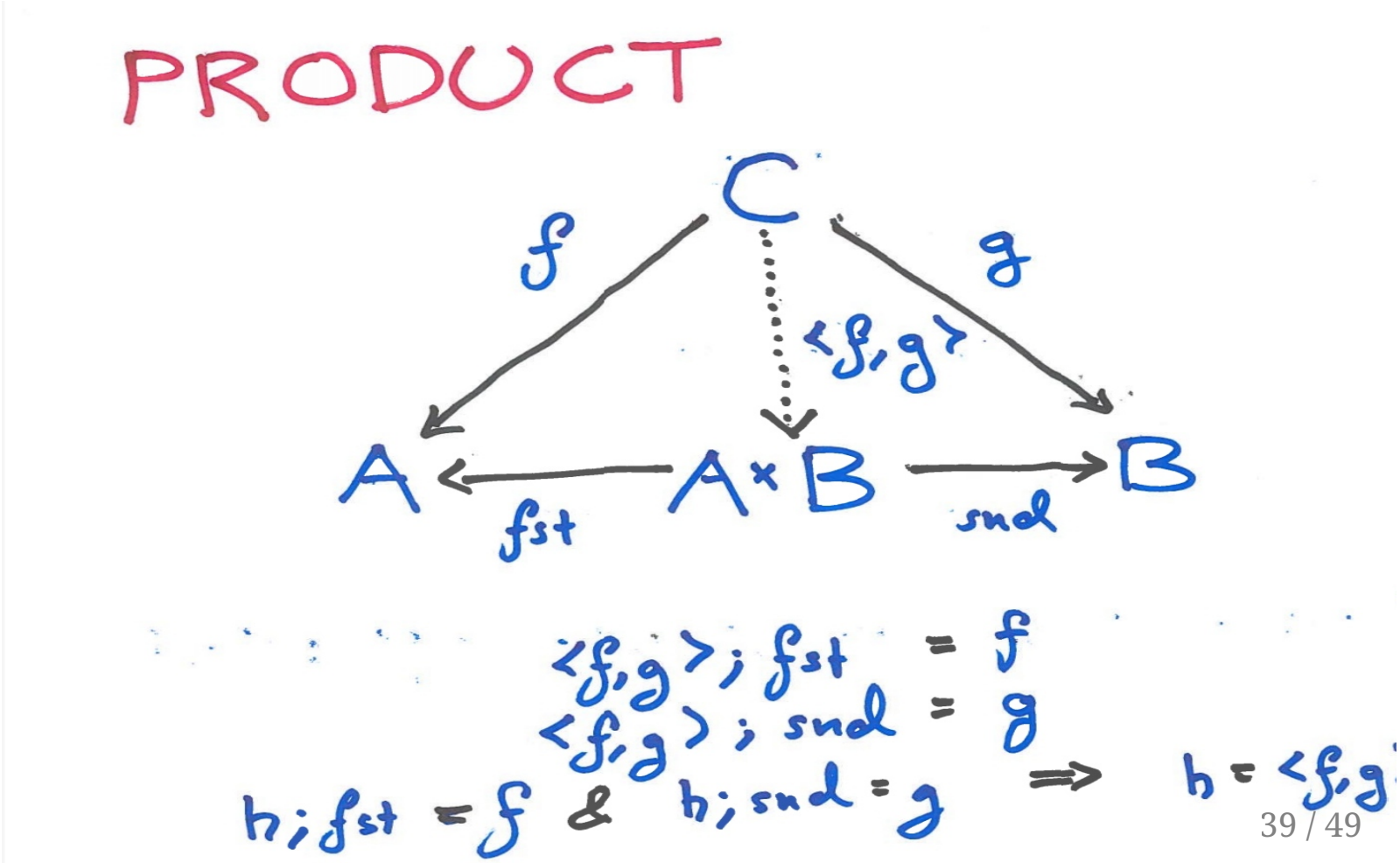
> 范畴论中有一个常见的构造，叫做泛构造（Universal Construction），它就是通过对象之间的关系来定义对象，其方式之一就是拮取一个模式——由对象与态射构成的一种特殊的形状，然后在范畴中观察它的各个方面。如果这个模式很常见而且范畴也很大，你就会有大量的机会命中它。技巧是如何对这些命中机会进行排序，并选出最好的那个。
>
> -- category theory for programmers

product coproduct limit colimit end coend left/right [Adjunction, Kan Extension]

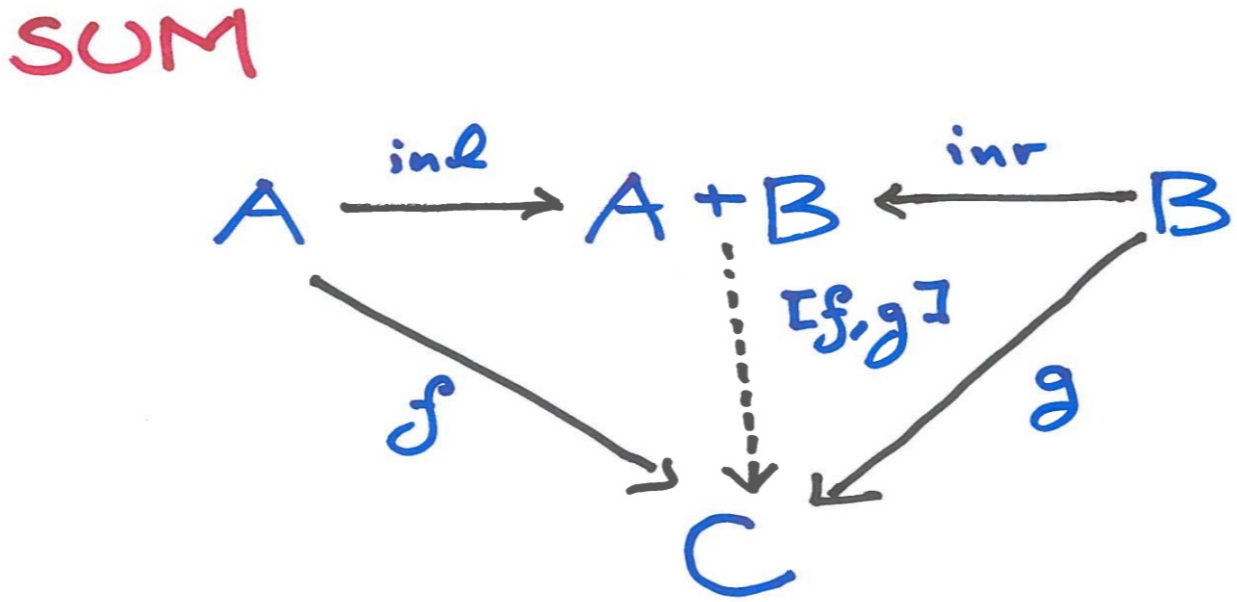
一个不小的菜单

作为流动摊点，很遗憾只能略举product/coproduct为例了

product



coproduct



$$inl ; [f,g] = f$$
$$inr ; [f,g] = g$$
$$inl ; h = f \;\&\; inr ; h = g \Rightarrow h = [f,g]$$

最终降临:多项式函子

和，积， 指数，它们可以构成所谓的多项式函子

它是多项式这一概念的范畴化

就像裴蜀定理可以从多项式推广到主理想整环上

微积分中的一些定理同样适用于多项式函子

(精确的定义，还请参阅ncatlab和Notes on Polynomial functor)

F-Algebra

我们首先对一个代数结构进行范畴化，画出它的交换图

(详细步骤请参阅刘新宇先生的同构一书，章节4.5)

然后用一个多项式函子F编码其结构

通过抽取出基底对象，所谓的“Algebra”和同态映射

我们得到了此代数结构的范畴表示

来点直觉

这是所谓的Algebra

```
type Algebra f a = f a -> a
```

这里的Algebra来自阿拉伯语词根，意为“重组”

描述下代数结构的形状

```
data MonoidF a = MEmpty | MAppend a a
```

用一个具体的Algebra去解释它

```
alg4String :: Algebra MonoidF String
alg4String MEmpty = ""
alg4String (MAppend x y) = x ++ y
```

加点类比

F => 抽象语法树

a => 抽象语法数的具体解释方式

具体的某个Algebra f a => 解释器

Initial Algebra

对一个多项式函子来说

Initial Algebra是一个这样的Algebra

```
initAlg :: f i -> i
```

类型f i和i同构

无穷的开端

请把i想象成f的一个不动点吧。因为 f i ~= i

是的，这里潜藏着递归

这个不动点i，可以粗略地看作ADT中的类型 -- 但不全

因为还有对偶的概念F-CoAlg和Terminal Algebra呢

标题回收

Initial Algebra正是初始代数语义的核心概念

一些事实

- 显然我没做到展示完整的初始代数语义这个框架，由于要支持一个叫做fold/build的规则转换，我们得用到一个所谓的扩展初始代数语义，为此将不得不引入极限和泛锥的概念，那真的太耗时了，它没办法直接用haskell代码对应
- haskell里面的不是真ADT，有些类型上定义不出来所需的同构代数，关键词请搜strict positivity

鸣谢

Trebor告诉了我Zipper对应ADT的微分运算

感谢我所读过的所有相关材料的作者