An Introduction to Algebraic Data Type

Preface

本文是面向不熟悉函数式编程和类型论的读者的入门内容，主要参考Chris Taylor在`London Haskell`上主题演讲的slide.

代数数据类型(*Algebraic Data Type*,简称*ADT*)是许多函数式编程语言(Haskell, Scala, Idris等等)中极其重要的概念。很多人会无法理解，为什么代数数据类型是"代数的"？这是一个很数学的描述，但真正学习ADT时，大多数人可能并不会有一种在学数学的错觉。

从某种程度上来看，ADT在这一点上至少是成功的，它并没有给初学者带来过多的负担。不过在我看来，不求甚解地学习代数数据类型，对一名希望掌握函数式编程的初学者来说，并不是一件好事。

因此，我打算用一种更数学的手段来讲解什么是代数数据类型以及它的应用，帮助初学者更好地理解ADT.

Chapter 1 : What's a algebra?

有关ADT，最为重要的概念问题可能就是：**什么是代数?**

在数学上，代数(*Algebra*)由三部分组成：

- **对象**(*Object*)是代数的`thing`(事物)，对象的集合定义了我们讨论的内容。
- **操作**(*Operations*)为我们提供了将旧事物组合为新事物的方法。
- **定律**(*Laws*)是对象与操作之间的关系。

我们日常使用的数学运算也可以视为一个代数，如`1+2=3`或是稍微复杂一些的涉及`x, y, z...`这样的变量表达式，对象是数字或变量，操作是加法、乘法和友元(`friends`)，遵守着几条近乎trival的定律，如：

```
0 + x = x
1 * x = x
```

接下来，我将使用一门纯函数式语言(*Pure Functional Programming Language*)—Haskell来构造程序中常见的代数。

对Haskell来说，最广泛的代数毫无疑问是关于类型(*Type*)的代数，我们称其为`Hask代数`.这一结构与Haskell类型系统的特性密切相关，Haskell强大的类型推导(*Type Inference*)能力也源自它优秀的ADT设计。

`Hask代数`的`objects`是类型，例如`Bool`和`Int`.尽管Haskell的类型系统十分复杂，但最基础的类型也只需要囊括`Bool`和`Int`两类，有关这方面的内容可以参考类型论(*Type Theory*),比较好的教材可能是TAPL(*Types and Programming Language*).

`Operations`根据已经存在的类型生成新的类型.典型的有类型构造函数`Maybe`和`Either`,它们本身并不是类型，但你可以用它们来创建类型，例如`Maybe Bool`、`Maybe Int`和`Either Bool Int`,我们称这样的操作为**类型构造子**.

`Hask代数`中的`Laws`需要结合具体的`operations`才更有价值，我们留到之后再说...

Chapter 2 : Begin with Counting

我们接下来要引入`Hask代数`中几种`operators`，它们分别类比于数字代数中的加法和乘法.

在正式介绍这些`operators`之前，我们不妨通过计算类型的可能值来寻找`Hask代数`与更为常见的数字代数之间的联系。以`Bool`为例：

```
data Bool = False|True
```

`Bool`类型的值集只有两个元素:`False`和`True`.此时我们可以将`Bool`类型与数字代数中的`2`联系起来.

如果说`Bool`类型对应数字代数中的`2`,那么`1`和`0`对应的类型是什么?

很显然，`1`对应的应该是只有一个值的类型，而`0`对应的则是不具备值的类型.

在计算机科学中，`1`对应的类型通常称为`Unit`，并定义为：

```
data Unit = Unit
```

在Haskell中天然地附带了一个只有一个值的类型-`()`(被称为'*单位*'),我们不能自己定义它(因为这是Haskell自己附带的原始类型),但可以给出它的标准定义:

```
data () = ()
```

采用这种计数法将`Hask代数`与数字代数进行类比,`Int`对应数字2^{32}或2^{64}.这取决于你的操作系统是32位还是64位。

至于`0`对应的类型,根据定义,我们知道,这将是一个没有值的类型。尽管这听起来和奇怪,但就像更常见的集合论(*Set Theory*)中的空集$\emptyset$一样,`0`元素在类型论中几乎是不可缺少的。很容易定义出这样的类型:

```
data Void
```

注意,该数据定义中并没有构造函数,因此我们永远无法构造出`Void`类型的值——`Void`有零个值,这正是我们想要的!

Chapter 3 : Add & Mul

在了解基本的计数法之后,从原则上,可以键入(或者换个说法"系统自带")与3,4,5等等相对应的类型。但如果我们能够从尽可能少的原始类型的基础上构造新的类型,那就再好不过了。而这也正是`operators`在`Hask代数`中的作用。

对应于类型的加法是:

```
data Add a b = AddL a|AddR b
```

也就是说,类型`a + b`是持有`a`或`b`的`tagged union`.

利用类型的加法,可以很自然地构造出与`3`对应的类型:

```
Add Bool ()
--列出Add Bool ()的值
addValues = [AddL False, AddL True, AddR ()]
```

我们称这种类型为`sum`类型.在Haskell中,`sum`类型通常称为`Either`.

定义如下:

```
data Either a b = Left a|Right b
```

PS:在本文中仍用`Add`表示`sum`类型.

对应类型的乘法为:

```
data Mul a b =  Mul a b
```

换句话说,类型`a * b`是既包含`a`又包含`b`的容器(*Container*).

我们可以很自然地定义出`4`对应的类型:

```
Mul Bool Bool
--列出Mul Bool Bool的可能值
mulValues = [Mul False False, Mul False True, Mul True False, Mul True
True]
```

我们称这种类型为`product`类型.在Haskell中,`product`归属于`pair`类型:

```
data (,) a b = (a, b)
```

PS:在本文中仍用`Mul`表示`product`类型.

至此，我们已经可以定义出对应于`0`到所有整数的类型.

Chapter 4 : Laws in Hask Algebra

4.1 Basic Laws

跳开`Hask代数`的`objects`(也即类型)本身的特性，回到代数这个抽象概念本身。

最重要的概念，可能就是如何定义两个对象间的**相等性**.需要注意的是：这种相等并非是Haskell中的(`==`)函数意义上的平等，而是基于同构(*Isomorphism*)的`equality`.

换言之，如果两种类型是一一对应的，则可以说两种类型`a`和`b`相等。

那么我们可以编写两个函数:

```
from :: a → b
to :: b → a
```

将`a`与`b`的值进行配对，且下面的等式总是成立(`haskell`意义下的`==`):

```
to(from a) == a
from(to b) == b
```

来看一个例子：从直觉上来看，我们认为`Bool`类型与`Add () ()`是等价的。我可以通过下面的程序来演示相等性:

```
to :: Bool → Add () ()
to False = AddL ()
to True = AddR ()

from :: Add () () → Bool
from(AddL _) = False
from(AddR _) = True
```

为了方便，规定`===`表示类型之间的这种相等性。

4.2 Sum & Product Type Laws

4.2.1 Sum Laws

`sum`类型有两条基本的定律:

```
--第一条定律
Add Void a ≡≡ a

--第二条定律
Add a b ≡≡ Add b a
```

我们可能更熟悉对应数字代数的这些定律:

```
0 + x = x
x + y = y + x
```

当然，这些定律的正确性是能够被证明的，可以采用计数法，也可以编写`from`和`to`函数。

4.2.2 Product Laws

`product`类型的定律有三条：

```
--第一条定律
Mul Void a ≡ Void

--第二条定律
Mul () a ≡ a

--第三条定律
Mul a b ≡ Mul b a
```

对应数字代数的定律：

```
0 * x = 0
1 * x = x
x * y = y * x
```

4.2.3 Add & Product Laws

最后，还有一条定律将`sum`和`product`类型联系起来：

```
Mul a (Add b c) ≡ Add (Mul a b) (Mul a c)
```

对应数字代数中的乘法分配律：

```
a * (b + c) = a * b + a * c
```

我们将以上定律的证明留作练习~

Chapter 5 : Functional Type

5.1 Values

在Haskell中，除了`Int`和`Bool`这样的具体类型之外，还有`Int -> Bool`或是`Double -> String`这样的函数类型(*Functional Type*).

我们也需要将这些常用的函数类型纳入`Hask代数`中，它们同样有对应的代数定律。

为了将函数类型融入代数，让我们回归到计数法。

对`a -> b`这个函数类型而言，它的值实际上就是由`a`类型映射到`b`类型的函数个数。

考察具体一点的实例，`Bool -> Bool`类型，应该有2^2个可能的`Bool -> Bool`函数.

如下所示：

```haskell
f1 :: Bool → Bool
f1 True = True
f1 False = False

f2 :: Bool → Bool
f2 _ = False

f3 :: Bool → Bool
f3 _ = True

f4 :: Bool → Bool
f4 True = False
f4 Flase = True
```

如果`b`仍是`Bool`类型，而`a`变成一个`Trio`类型，定义如下：

```haskell
data Trio = First|Second|Third
```

很显然此时`a -> b`类型的函数有2^3个.

将这个结论推广到一般情况下也成立：如果有`A`类型的值，`B`类型的值，则`A -> B`类型的值的数目为B^A.

这也说明，函数类型是一种指数类型.

5.2 Laws

函数类型的定律可以与函数式编程中`currying`的概念联系起来，它对应的算术意义也几乎是一目了然。

```haskell
--第一条定律
() → a ≡ a

--第二条定律
a → () ≡ ()

--第三条定律
(a → b, a → c) ≡ a → (b, c)

--第四条定律
a → (b → c) ≡ (b, a) → c

--第五条定律
a → b → c ≡ (a, b) → c
```

熟悉函数式编程的读者不难发现，第五条定律就暗含Haskell中的`curry`和`uncurry`函数.

除`curry`以外，其他对应的数字代数定律(这一次采用数学公式表述)如下：

$$a^1 = a$$

$$1^a = 1$$

$$b^a \cdot c^a = (bc)^a$$

$$(c^b)^a = c^{b \cdot a}$$

这些定律的证明留作练习~

Chapter 6 : Recursive Type

至此，关于`Hask代数`的基础内容已经全部结束了！

接下来，我们会进一步探究代数数据类型的应用。

我们先给出一张关系图，方便大家联系`Hask代数`和数字代数。

$\text{Void} \leftrightarrow 0$

$()$,

$\text{Unit} \leftrightarrow 1$

$\text{Add}\ a\ b \leftrightarrow a + b$

$(a, b), \text{Mul}\ a\ b \leftrightarrow a \cdot b$

$a \to b\ \leftrightarrow b^a$

6.1 Maybe

在正式进入递归类型(*Recursive Type*)之前，先来探究一下`Maybe a`类型。

我们前面就讨论过`Maybe`是一个类型构造子，`Maybe a`表示它可能包含类型`a`的值也可能为空。

```
data Maybe a = Nothing|Just a
```

`|`说明这是一个`sum`类型，我们可以尝试用`Add`类型来编写它：

```
data Nothing = Nothing
data Just a = Just a

type Maybe a = Add Nothing (Just a)
```

这里我们用`type`代替了`data`，这意味着`Just a`不再是一种新的类型，而是我们已知类型的同义词。

更进一步，注意到，`Nothing`仅有一个值，因此等价于`()`.

同理，`Just`是一个具有类型`a`的单个值的容器，因此`Just a`与`a`等价，因此我们有：

```
type Maybe a = Add () a
```

由计数法，不难证明`Maybe a`与`1 + a`同构，`Maybe`只是为类型增加了一个可能值。

`Maybe`的用法尽管看起来十分简单，却为我们正式引入递归类型打开了大门。

6.2 Lists

Haskell中的基本列表是一个链表(可以用`as`表示)。

`as`的列表要么为空，记为`[]`，要么是单个`a`通过`cons`加入到另一个`as`的列表中，表示为`a:as`.

我们可以很自然地定义出`List a`：

```
data List a = Nil|Cons a (List a)
```

和`Maybe`类似，`List`类型是两个简单类型的`sum`.

第一个`summand`是`Nil`，这是一个等效于`()`的`nullary`构造函数.

第二个`summand`是`Cons a (List a)`，它是由`a`和`as`列表组成的`product`.

如果我们将列表写为`L(a)`，则对应的代数形式为：

```haskell
L(a) = 1 + a * L(a)
```

与`Maybe`不同的是，我们并不能将列表类型写成以下形式：

```haskell
type List a = Add () (a, (List a))
```

这样的代码不会进行编译，这严格来说与Haskell的类型系统有关。

类型同义词在Haskell中将在类型检查之后，编译之前进行`expanded`.而这个定义永远不会完成`expanded`，它只会不断做递归增长：

```haskell
Add () (a, Add() (a, Add () (a, ...)))
```

这个问题可以使用`newtype`(而不是`type`)声明来解决：

```haskell
newtype List a = L (Add () (a, List a))
```

在这段代码中，我们将`List a`类型包装在新的构造函数`L`中，使之满足编译器的要求，额外的构造函数会在这一过程中被优化掉，其他部分则和上面的`type`声明相同。

将`List a`的代数表达式进行`expanded`(*展开*)：

$$L(a) = 1 + a \cdot L(a) = 1 + a \cdot (1 + a \cdot L(a)) = 1 + a + a^2 \cdot (1 + a \cdot L(a)) = \ldots = 1 + a + a^2 + a^3 + a$$

它的各项系数均为1，这实际上告诉我们，`List a`要么是空列表，要么是包含单个`a`的列表，要么是包含两个`a`的列表...

到了这个地步，不妨将`List a`的代数表达式解出来，这只涉及非常基础的代数知识(没必要扯到母函数)：

$$L(a) - a \cdot L(a) = 1 \ \rightarrow (1 - a) \cdot L(a) = 1 \rightarrow L(a) = \frac{1}{1-a}.$$

这看起来非常有趣，我们并不知道另一种类型减去一种类型的含义；同样地，我们对一种类型`divide`(除以)另一种类型的含义也一无所知。

但这个式子是有意义的，根据微积分中学过的Taylor Series，我们也能知道`L(a)`的解析式对应着展开式：$L(a) = \sum_{i=0}^{\infty} a^i$，与我们展开得到的式子相同。

6.3 Trees

先考虑较为简单的二叉树，在Haskell中这样定义二叉树：

```haskell
data Tree a = Nil|Tree (Tree a) a (Tree a)
```

它也是两种类型的`sum`，其中`Nil`(或者写为`Empty`)等价于`()`类型，而`Tree (Tree a) (Tree a)`则是一个有三个`terms`的`product`类型。

同样,用`newtype`声明来(递归地)定义我们的二叉树：

```haskell
newtype Tree a = T (Add () (a, (Tree a, Tree a)))
```

我们将树的类型记为$\mathrm{T}(a)$，可以写出树的相关表达式：

$$T(a) = 1 + a \cdot T^2(a)$$

同样，可以尝试解出$\mathrm{T}(a)$的表达式,只需要经过简单的重新排列即得：

$$a \cdot T(a)^2 - T(a) + 1 = 0$$

将上面的式子视作是关于$\mathrm{T}(a)$的二次方程，可以解出：

$$T(a) = \frac{1 - \sqrt{1 - 4a}}{2a}$$

这又带来了一个全新的运算—对类型求平方根.目前我们仍然不清楚这一操作的具体意义。

利用级数知识对$T(a)$进行展开：$T(a) = 1 + a + 2a^2 + 5a^3 + 14a^4 + \ldots$

这些系数提示我们：一棵树有两种方式表示包含两个`A`类型的值，有5种方式表示包含三个`A`类型的值。

利用这个级数展开式的各项系数，就能够计算出可能存在的不同二叉树的数目。

熟悉*Analytic Combinatorics*的读者很容易看出这实际上就是一个*Generating Function*(母函数).

6.4 Even More Trees

6.4.1 Finger Trees

前面已经学习了二叉树在Haskell中的基本表达形式，接下来让我们学更多与树有关的内容。

对于纯函数式语言来说，Finger Trees提供了许多高效的序列操作，也充分体现了函数式编程特有的某些属性，非常值得学习。

遗憾的是，Finger Trees的实现及使用略显复杂，且需要一定的数据结构基础，这里只能介绍Finger Trees的基本动机和定义；对于它的应用，我们会在进一步学习有关*catamorphism*的内容后再进行补充。

首先，考虑Leafy形式的二叉树：

```haskell
data Tree a = Leaf a|Tree (Tree a) (Tree a)
```

这种形式下的二叉树只有在叶结点才能取到值，且无法表示一个空树。

Leafy二叉树看起来与原先的二叉树差别不大，但沿着这一思路，我们可以很快得到一个在纯函数式中颇为强大的数据结构—Finger Trees.

要得到Finger Trees，让我们先思考一个小问题：**如何表示一个满Leafy的二叉树?**

我们可以借鉴自然数的定义：

```haskell
data Nat = Z|S Nat
```

，这是一个很常见的递归结构，我们可以给`Nat`添加一定的操作`s`和对象`a`,进而可以表达类型的嵌套层数：

```haskell
data Nat s a = Z a|S (Nat s (s a))
```

于是我们只需要定义Leafy二叉树的结点类型：

```haskell
data Node a = Node a a
```

接下来只需要不断嵌套`Node`结构，得到的就是满Leafy的二叉树：

```haskell
type Tree a = Nat Node a
```

不妨拿一个例子来进行演示：

```haskell
Z 1 :: Tree Int
 S (Z (Node 1 2)) :: Tree Int
 S (S (Z (Node (Node 1 2) (Node 3 4))))
```

只需要把我们的`Node`结点替换为`2-3`树的结点，就得到Finger Trees:

```haskell
data Node23 a = Node2 a a|Node3 a a a
data Tree23 a = Nat Node23 a
```

而Finger Trees的正式定义如下：

```haskell
data FingerTree a =
     |Empty
     |Single a
     |Deep (Digit a) (FingerTree (Node a)) (Digit a)

type Digit a = Digit [a]
```

那么，本节有关Finger Trees的内容就告一段落。如果觉得本节的难度和跨度略大，理解上有些困难，也不要灰心，我们会在后续的内容中不断完善和补充Finger Trees的理论框架。如果您是数据结构的初学者，可以跳过本节的学习。

6.4.2 Seven Trees In One

当我们只考虑包含`unit`类型的树，即`Tree ()`，那么`T(a)`中的`a`等于1.

会有$T = 1 + T^2$, 进而$T - 1 = T^2$.

注意到
$T^6 = (T-1)^3 = T^3 - 3T^2 + 3T - 1 = T(T-1) - 3T^2 + 3T - 1 = -2T^2 + 2T - 1 = 1$
.

这个结果从类型上来看，显然是错误的，一颗六元组的树不可能和单位类型等价。而上面式子的错误正在于它对类型方程进行了**不合理的减法运算**。

这个事实说明：对类型不合理的操作可能会产生错误，`Hask代数`不能简单地同数字代数一般做推广。

但与之类似的另一个结果却是成立的，$T^7 = T|$.

事实上，如果你感兴趣的话，可以尝试证明：当`T`的幂不为`6`的倍数时，上面使用减法的推导能够转化为只需要加法和乘法的`忠实`证明，进而该运算对类型有效。（详细地，可以自行下载论文七棵树合一）

Chapter 7 : Calculus of Types

7.1 Zippers

7.1.1 Some Problems

在本章中，我们将讨论对类型进行演算的意义。

在前面的内容中，我们提到过常规的Haskell列表是一个链表。尽管它易于定义且易于使用，但访问这些链表或是修改其中的元素的效率不高。如果要将元素添加到长度为n的链表中，需要的时间是$O(n)$.另外，一个链表是不能支持有效随机访问的，这也为Haskell列表的使用带来的困难。

当然，使用数组能够解决此类问题，但Haskell这样的函数式语言具有数据结构不可变的特性.如果你希望数组中的元素在内存中是连续的，那么数据共享就会变得困难。

这里解释一下什么是数据结构不可变(*Immutable Data*)。在JavaScript中，这样一种写法十分常见：

```javascript
var a = {qty:1}
a.qty = 10;

a.qty;//10
```

但在数学上，我们知道这个式子是不合理的，变量`a`在运算过程中是不能**中途替换**的，新产生的`a.qty`应该作为一个新对象在新地址中储存。但在许多主流语言中，这种写法都是被允许的。深刻地讨论这个问题，需要一定的数据结构理论的基础，我们暂且跳过。

我们把这种在过程中内容发生变化的数据结构叫做*可变数据结构*.

而不可变数据结构则不允许发生数据的修改，转而进行数据的变换(从数学的角度来看，就是**数据间的映射**)。因此，Haskell中的函数与数学中的函数是等同的。

数据不可变在很大程度上消除了引用时可能出现的错误，比如在JavaScript中很常见的:

```
const a = {qty:1}
const b = a;

a.qty = 10;

b.qty;//10
```

将`a`赋值给其他变量时，其他变量的值也随着`a`的改变而发生改变，进而很容易出现难以排查的错误。

除此之外，可变数据结构对于并行和并发的支持几乎没有，在这个*Computer Architecture*发展的黄金时代，这样的缺陷甚至是致命的。

当然，这种引用赋值的形式实现的数据共享，在很大程度上节约了内存，也正因如此，可变数据结构在硬件性能相对落后的年代更为流行。

但正如前面所说，随着硬件性能的提升以及并行与并发编程的流行，不可变数据结构也融入到了许多主流语言中。函数式编程语言已经不再是Toy Language,在实际生产环境中的应用也越来越广泛。

例如，在C++中我们能够通过*Template Metaprogramming*实现对函数式的支持；前端开发最为著名的框架React也十分推崇函数式组件。有理由相信，在未来，函数式编程的应用前景会更加宽广。

7.1.2 List Zippers

幸运的是，并非所有应用程序都需要随机访问。如果我们只需要追踪元素在列表中的位置，能够在列表中四处移动，并能够随时修改元素，那么Haskell中的`zipper`提供了这一功能。

`zipper`是一种数据结构，具有一个可分辨位置的指针，称为`focus`.

我们可以为任何一种数据结构编写一个`zipper`拉链，但这里只讨论列表的`zipper`.

要想储存一个指针，我们可以用一个`Int`将列表包装起来，该`Int`将当前位置存储在`focus`中。但这样做充满各种各样的困难，而且难以满足对`focus`高效访问的需求。

更好的做法是定义一个新的类型，它由`focus`中的元素和一对列表组成，其中一个列表中包含`focus`之前的元素，另一个包含`focus`之后的元素.在Haskell中，我们将其写为

```
data Zipper a = Z [a] a [a]
```

为了使`zipper`有用，我们需要定义一些函数来左右移动`focus`.

考虑到效率问题，通常从第一个列表的末尾有效地添加和删除元素，以相反的顺序存储第一个列表:

```
left (Z (l:ls) c rs) = Z ls l (c:rs)
right (Z ls c (r:rs)) = Z (c:ls) r rs
```

7.1.3 One-Hole Contexts

结合前面的内容，可以发现`zipper`是一种`product`类型—它是两个`as`列表和一个`a`列表的`product`.

由于*Lists*的代数表达式为$L(a) = \frac{1}{1-a}$,所以*List Zippers*的代数表达式$LZ(a) = a \cdot L(a) \cdot L(a) = \frac{a}{(1-a)^2}$.

注意到，我们能够将*List Zippers*合并为$Z(a) = a \cdot L^2(a)$.

这实际上提示我们，两个列表的作用是告知我们`focus`的当前位置，而`a`的作用则是通过填写`focus`所在的数据来完成`zipper`.

那么，`product`的第一部分就变成了一段数据，我们可以把第二部分看作是一个带`hole`的数据结构，当我们使用数据来配对填充这个`hole`时，就得到了一个`zipper`.

我们把这种带一个`hole`的数据结构叫做`one-hole context`.在大多数情况下，`hole`代表无数据-它是数据的占位符，因此可以说`hole`是`()`类型.

此外，任何`zipper`都能够分解为`one-hole context`的形式，并由一条数据填充`hole`.

7.2 Tuples

前面提到过，任何一种数据结构都能够编写出对应的`zipper`，所以这一次让我们为元组编写一些`zipper`.

7.2.1 One-Tuples

先从最简单的情况—一元组(One-Tuples)开始.

任意一元组`a`只是一个数据片段，只有一个数据位置，因此它只有一个`focus`,该一元组的数据就是`a`.

那么我们只需要考虑`one-hole context`是什么，只有一个地方可以放置`hole`,一旦在一段数据中放入`hole`,我们就只剩下`hole`本身了!

数据与`one-hole context`的配对如下:

```
(a,o)
```

`one-hole context`的类型必须是`()`或`1`，唯有如此，当它与`a`配对(可看作特殊的`product`)时，才能再次获得数据`a`.

7.2.2 Two-Tuples

在两元组中，`hole`有两个可能的位置，也即左和右。我们可以使用`sum`类型来表示`hole`可能出现的情况，数据与`one-hole context`的配对如下:

```
(a,(o,a)+(a,o))
```

`one-hole context`类型为`a + a`或`2a`.

7.2.3 Finding the pattern

根据一元组和两元组的构造过程，我们可以很容易写出三元及以上元组的`one-hole context`.

以三元组为例，数据与`one-hole context`配对如下:

```
(a,(o,a,a)+(a,o,a)+(a,a,o))
```

`one-hole context`的类型为`a²+a²+a²`或`3a²`.

对于一个n元组来说，其`one-hole context`的类型为$n \cdot a^{n-1}$.

我们可以发现一个有趣的规律，在下图中，左侧是`n`元组的类型，右侧是对应的`one-hole context`的类型:

$$a \to 1$$

$$a^2 \to 2a$$

$$a^3 \to 3a^2$$

$$a^n \to n \cdot a^{n-1}$$

敏锐的读者可能会发现，一个类型为a^n的`one-hole context`的类型就是a^n的`derivative`(导数)!

实际上，这种模式对于任何一种数据结构都适用，我们将`zipper`与类型求导联系了起来。这也证明了任意一种数据结构都有对应的`zipper`.

7.3 Laws of Calculus

我们已经将一个类型的导数与它的`one-hole context`等同起来，类型能够进行导数运算可以说十分令人惊讶。

接下来，不妨考虑一下在数字代数中常见的微分算符的运算定律是否也能够推广到`Hask代数`中?

为了表示方便，我将使用∂a代表"对`a`求导"，这与我们熟悉的标记形式不同。

7.3.1 Constants

在微积分中，任何常数的导数为0是一条十分基础的规则:

$$\partial_a(\text{const}) = 0$$

放在类型中，这一规则也成立，只需要记住派生运算符$\frac{\mathrm{d}}{\mathrm{d}a}$在包含类型`A`的数据的数据结构中构造了`hole`.如果类型中没有任何类型`A`的数据，则对应的`one-hole context`的类型为`Void`，你不能创造它的任何实例。

7.3.2 Sums

对于任意可导的$f(a)$和$g(a)$，在数字代数中满足:

$$\partial_a(f(a) + g(a)) = \partial_a f(a) + \partial_a g(a)$$

我们将这个规则的类型代数证明留作练习，这一规则告诉我们，在`sum`类型中创建一个`hole`相当于在每一个被加数(`summands`)中创建一个`hole`，并对它们取`sum`。

更精确地说，如果`F a`和`G a`是具有`one-hole context`(分别为`DF a`和`DG a`)的类型，则

```haskell
type Sum a = Add (F a) (G a)
```

的`one-hole context`是

```haskell
type DSum a = Add (DF a) (DG a)
```

7.3.3 Products

在微积分中，`product`的导数遵从Leibniz's product rule:

$$\partial_a(f(a)\cdot g(a)) = \partial_a f(a)\cdot g(a) + f(a)\cdot \partial_a g(a)$$

在类型代数中，这意味着在两种类型的`product`中构造`hole`相当于在第一种类型中取`hole`(保留第二种形式)或是在第二种类型中取`hole`(保留第一种形式).也就是说

```
type Prod a = (F a,G a)
```

的`one-hole context`为

```
type DProd a = Add (DF a,G a) (F a,DG a)
```

7.3.4 Composition

如果数据结构`F`中包含`G a`类型的元素，那么

```
data Compose f g a = Compose (f(g a))
```

其中`f`和`g`的类型为`* -> *`，而`a`的类型为`*`，为了语法简洁可以直接写作`f(g a)`.

在微积分中，链式法则告诉我们:

$$\partial_a f(g(a)) = \partial_a g(a)\cdot \partial_g f(g(a))$$

在类型代数中，这意味着要在分层数据结构中创建一个`hole`，我们需要一个`product`，一半的`product`告诉我们`hole`在外部结构中的位置，另一半则追踪内部结构中的`hole`.也就是说:

```
type Comp a = F (G a)
```

的`one-hole context`为

```
type DComp a = (DG a,DF(G a))
```

7.4 Deriving Zippers

前面提到过，`ADT`的派生(*derivative*)与其`one-hole context`相对应，就可以为任意数据结构派生`zipper`.

现在，我们将来验证这个说法的合理性。

7.4.1 List Zippers

列表的类型为$L(a) = \frac{1}{1-a}$,我们可以利用微积分中的*quotient*法则和*chain*法则对$L(a)$求导，会有:

$$\partial_a L(a) = \frac{1}{(1-a)^2} = L^2(a)$$

因此,$L(a)$的`one-hole context`也是一对列表。正如我们之前所看到的，为了获得*List Zippers*,将`a`与一对列表`L(*a)`进行`product`.这也证实了我们的推导是正确的。

7.4.2 Tree Zippers

二叉树的类型是$T(a) = 1 + a\cdot T(a)^2$.利用微积分中的隐函数求导，会有

$$\partial T(a) = T^2(a) + 2\cdot a\cdot T(a)\cdot \partial T(a)$$

合并化简后能够得到

$$\partial T(a) = \frac{T^2(a)}{1 - 2 \cdot a \cdot T(a)}$$

注意类型$\frac{1}{1-x}$与`x`列表的类型`L(x)`等价，所以上式又可以写作

$$\partial T(a) = T^2(a) \cdot L(2 \cdot a \cdot T(a))$$

也就是说，树的`one-hole context`是两棵树与一个包含类型元素`(bool, a, tree a)`的`product`.

每一个树的节点都与一个`Bool`相关联，该`Bool`告诉我们经过该节点后是左分支或是右分支，还有一个`Tree a`，其中包含所有我们未沿着另一条路径错过的元素。

7.4.3 Building a Rose Tree Zipper

在利用`ADT`的派生推导出*List Zippers*和*Tree Zippers*之后，让我们来处理一个稍微复杂一些的数据结构.

玫瑰树(*Rose Trees*)是每个节点上具有任意多分支的树。在Haskell中，我们可以这样定义玫瑰树:

```haskell
data Rose a = Empty|Rose a [Rose a]
```

最常见的玫瑰树可能就是计算机中的目录结构:每一个目录下面可能什么也没有或者带有更多的子目录，而这些子目录又可能带有更多的"子子"目录.

我们很容易写出玫瑰树的代数表达式：$R = 1 + a \cdot L(R)$，其中$L(R)$代表一个包含玫瑰树的列表.

由隐函数求导，可以得到

$$\partial_a R = L(R) + a \cdot \partial_R L(R) \cdot \partial_a R$$

合并化简后会有

$$\partial_a R = \frac{L(R)}{1 - a \cdot \partial_R L(R)}$$

同样对$\frac{1}{1-a \cdot \partial_R L(R)}$进行变换，化为$L(a \cdot \partial_R L(R))$.根据前面得到的$\partial_x L(x) = L^2(x)$会有

$$\partial_a R = L(R) \cdot L(a \cdot L^2(R))$$

那么玫瑰树的`zipper`就是

```haskell
type RoseTreeZipper a = (a, ([Rose a], [(a, ([Rose a], [Rose a]))]))
```

尽管上面这段代码看起来十分可怕，可它仍有意义。具体意义可以仿照前面两个`zipper`进行分析。

`RoseTreeZipper`的应用也很广泛，举个例子，利用玫瑰树`zipper`我们可以很轻松地在计算机文件系统中移动和查找文件，这是很酷的一件事!

Chapter 8 : Generalized Algebraic Datatypes

之前我们已经知道，如何使用`ADT`定义所需的数据类型，例如定义一个列表:

```haskell
data List a = Nil|Cons a (List a)
```

我们可以将`Nil`和`Cons`用函数形式表示出来

```haskell
Nil :: List a
Cons :: a → List a → List a
```

不难发现，`Nil`和`Cons`最后都返回到类型`List a`.这里的`a`称为虚幻类型(*phantom type*),需要经过推导才能知道。例如列表`[a]`这样的类型，它可以看作是某一类型`a`的列表，如`[1,2]`，这里的`a`就是整数类型。

虚幻类型的存在是为了记录在计算中的类型，当在定义类型时，若递归的基本形式与类型参数无关，这个无关的类型就为虚幻类型。

当然，除了记录这些类型之外，还需要通过将虚幻类型与`Int`或`Bool`等具体类型联系起来，从而加入一些限制，否则我们引入虚幻类型就失去了意义。

那么，能否提供一种一般化的数据类型，可以让构造器携带更多类型信息来对类型做出需要的限制呢?

`GADT`(*Generalized Algebraic Datatypes*)就是Haskell提出的解决方案，但注意，`GADT`在Haskell中是作为拓展出现，需要在GHCi中使用`:set XGADTs`才能开启。使用`GADT`定义类型时，一般遵循如下格式:

```haskell
data TypeName arg1 arg2 ... where
  Con1 :: Type1
  Con2 :: Type2
  ...
```

回到原来`List a`的例子，我们采用`GADT`的形式重写`List a`.

```haskell
data List a where
  Nil :: List a
  Cons :: a → List a → List a
```

这样就得到了列表的`GADT`写法，我们还可以为这个列表添加类型限制，返回一个`List Int`类型的列表:

```haskell
{-# LANGUAGE GADTs #-}
data List a where
  IntNil :: List Int
  Cons :: a → List a → List Int
```

再例如，`Maybe`类型的`GADT`写法:

```haskell
data Maybe a where
  Nothing :: Maybe a
  Just :: a → Maybe a
```

那么`GADT`有什么实际作用呢?

熟悉类型论的读者，可能会发现`GADT`是Haskell对于*Dependent Type*的模拟，`GADT`原名甚至就叫做`Agda Type`.这也就意味着通过`GADT`可以证明一些定理，例如，加法交换律,乘法结合律等等。

当然，利用Haskell进行定理证明并没有Agda,Idris或是Coq这些支持*Dependent Type*的语言那样轻松，即便是在引入广义代数数据类型的情况下，也很少使用Haskell进行定理证明。

`GADT`更多还是在特定情境下辅助类型构造器构造出所需要的数据类型。

除此之外，`GADT`还可以把*type*转化为*value*，把一系列不同的类型保存在一个值中。

原本在`ADT`中，我们可以构造一个类型`Witness`

```haskell
data Witness a = Witness a
```

问题在于，你要提供一个`a`类型的值才能将`Witness`类型记录进去。可很多时候无法构造出这样的值，或是有的类型没有值(如空类型`Void`)，因此`ADT`在构造类型方面也存在着局限性。

但在使用`GADT`的情况下，我们不一定要接受一个值，才能得到这个类型

```haskell
data Witness a where
  IntWitness :: Witness Int
  BoolWitness :: Witness Bool
```

这就是所谓的*type witness*.

使用`GADT`定义数据类型与用`ADT`定义数据类型的区别在于：在`GADT`中需要明确指出每个构造器的类型;而在`ADT`中，每个构造器的类型由编译器自动生成且严格受限。

例如，`ADT`中带Leafy的二叉树类型为

```haskell
data Tree a = Leaf a|Branch (Tree a) (Tree a)
```

通过Haskell编译器的类型推导，能够自动得到对应构造器的类型签名。

在`GADT`中则不然，我们需要手动写出`Branch`的类型签名

```haskell
data Tree a where
  Leaf :: a → Tree a
  Branch :: Tree a → Tree a → Tree a
```

当然我们可以为结果指定任意类型，比如返回一个`Tree Int`类型的二叉树

```haskell
{-# LANGUAGE GADTs #-}
data Tree a where
  Leaf :: a → Tree Int
  Branch :: Tree a → Tree a → Tree Int
```

Appendix : Finger Trees