# Agda 中的 coinductive data type

2018, May 30 by Tesla Ice Zhang

最近和 @16 聊了一些关于 Coq 的话题，16 给我讲了很多 Coq 对 coinductive 数据类型的处理方式，让我一脸懵逼。

由于 Agda 中（似乎？）没有像 Coq 那样区分 coinductive 和 inductive（而是作为几个 `postulate`，下面会说），所以我从来还没想过这些问题。 然后就研究了一下，感觉是之前不会的东西，就写篇博客聊聊。

本文命名仅仅是比较正常，请 Haskell 程序员不要模仿。

## 名称解释

比较面向小学生的说法是，coinductive 就是在归纳的时候上升的，而 inductive 是在归纳的时候下降的。

比较面向工业界编程的说法是，coinductive 的数据类型是无穷的，inductive 是有穷的。 比如 `java.util.Collection` 就是 inductive 的， `java.util.Iterable` 就是 coinductive 的。

## For experts

我知道 Agda 现在有 `coinductive record` + `copattern` 这种更推荐的方式来使用 coinductive， 但本文讲的是它的 “old-way” coinductive。

# 从字符串说起

Agda 的内置字符串类型（ `Agda.Builtin.String` ）是作为 `postulate` 定义的：

```
module Agda.Builtin.String

postulate String : Set
{-# BUILTIN STRING String #-}
```

然后又 `postulate` 了一堆函数：

```
postulate primStringToList   : String → List Char
postulate primStringFromList : List Char → String
postulate primStringAppend   : String → String → Strin
postulate primStringEquality : String → String → Bool
postulate primShowString     : String → String
```

---

很明显这些函数是直接映射到目标语言的原生函数的。 但是这就直接导致这些函数以及 `String` 类型自己的性质无法被用于形式验证了（因为实现对 checker 是不可见的）， 所以十分鸡肋（只能说可以运行时用用吧，但 Agda 基本都是不运行的）。 如果是需要 Haskell 的那种

```
type String = List Char
```

的 `String` 的话，又需要用 `primStringToList` 转来转去，十分~~键山雏~~麻烦。

之所以要研究这个，是因为我想试试在 Agda 里调用 putStrLn，whose 参数类型是 String。 然后我看向了标准库（而不是内置库）的 IO.Primitive.putStrLn 的实现：

```
postulate IO : Set → Set
{-# BUILTIN IO IO #-}

postulate putStrLn : Costring → IO Unit
{-# COMPILE GHC putStrLn = putStrLn #-}
```

看到 Costring 就应该知道这个应该是在外面被调用的，真正被使用的是 IO.putStrLn（就是所谓的『外面』）， 所以我们可以看看 IO 里对 putStrLn 的实现：

```
import IO.Primitive as Prim

infixl 1 _>>=_ _>>_

data IO {a} (A : Set a) : Set (suc a) where
  lift   : (m : Prim.IO A) → IO A
  return : (x : A) → IO A
  _>>=_  : {B : Set a} (m : ∞ (IO B)) (f : (x : B) → ∞
  _>>_   : {B : Set a} (m₁ : ∞ (IO B)) (m₂ : ∞ (IO A))

{-# NON_TERMINATING #-}

putStrLn∞ : Costring → IO ⊤
putStrLn∞ s =
```

```
  ♯ lift (Prim.putStrLn s) >>
  ♯ return _

putStrLn : String → IO ⊤
putStrLn s = putStrLn∞ (toCostring s)
```

顺带一提，`Unit` 就是 `⊤`，`Agda.Builtin.IO`（内置库）和 `IO`（标准库）是不一样的，所以标准库把这个重写了而不是单纯地扩展内置库。

这个时候如果你只看过我之前的博客（而不是早就知道这些东西），你肯定充满了疑问，那个 `∞` 是啥？`♯` 是啥？`Costring` 又是啥？

所以就引入了本文的正题，coinductive data type。

## 无穷的数据类型

假设你已经知道 Haskell 里的无穷序列的定义，如果不知道，你看了这个就知道了：

```
data Colist a = a :> Colist a
```

然后我们可以给它定义 `cozipWith`，`cohead` 和 `cotail`：

```
cozipWith :: (a -> b -> c) -> Colist a -> Colist b ->
cozipWith f (a :> as) (b :> bs)  = f a b :> cozipWith f

cohead :: Colist a -> a
cohead (x :> _) = x
```

```
cotail :: Colist a -> Colist a
cotail (_ :> xs) = xs
```

这个就是绝对 coinductive 的 `List`，它绝对不会是 inductive 的（反观 `List` 就可以是 inductive 的，因为它可以被 `[1, 2, 3, 4]` 这样有限地构造。 即使它经常被当成 coinductive 地使用，因为 Haskell 语言层面就支持惰性数据结构）。

举两个应用的例子：

```
-- n 的无限序列
repeat :: a -> Colist a
repeat n = n :> repeat n

-- 斐波那契数列的无限序列
fib :: Integral n => Colist n
fib = 0 :> 1 :> cozipWith (+) fib (cotail fib)
```

举个 `Colist` 无法实现的 `List` 的使用例子：

```
-- 取列表的最后一项
last :: List a -> a
last [] = error "empty list"
last [a] = a
last (_ : as) = last as
```

如果你觉得刚掌握这个概念，对它还不太熟悉，想练习一下，那么可以试试 这个 CodeWars Kata。

## Agda 中的表示

看起来你已经懂了什么是 coinductive data type 了，那么在 Agda 中这玩意是怎么表示的呢？ 我们来看看 Agda 是怎么定义 `Colist` 的:

```
infixr 5 _∷_

data Colist {a} (A : Set a) : Set a where
  []  : Colist A
  _∷_ : (x : A) (xs : ∞ (Colist A)) → Colist A

{-# FOREIGN GHC type AgdaColist a b = [b] #-}
{-# COMPILE GHC Colist = data MAlonzo.Code.Data.Colist.
```

emmmm… 居然提供了 `[]` 这个构造器。。。
考虑到这里可能是为了编译到 Haskell 的 `List` 时更好地一一对应，就不吐槽了。
但这个玩意和 `List` 很明显还是不一样的，我们看到 `_∷_` 这个构造器的第二个参数就知道事情并不简单。
`∞` 是什么鬼，`∞ (Colist A)` 和 `Colist A` 有什么区别，怎么构造一个 `∞ (Colist A)` 呢？

## 真相

事实上因为 Agda 和 Haskell 一样没有严格区分 GADT 的 coinductive 与否， 所以它引入了下面这三个音乐符号，一个作为 coinductive 的标记，两个作为互相转换的运算符:

```
postulate
  -- 标记
  ∞  : ∀ {a} (A : Set a) → Set a
  -- 互相转换的运算符
  ♯_ : ∀ {a} {A : Set a} → A → ∞ A
  ♭  : ∀ {a} {A : Set a} → ∞ A → A
{-# BUILTIN INFINITY ∞  #-}
{-# BUILTIN SHARP    ♯_ #-}
{-# BUILTIN FLAT     ♭  #-}
```

于是我们除了这种使用方式：

```
repeat : ∀ {a} {A : Set a} → A → Colist A
repeat a = a ∷ repeat a
```

还可以借助 ♯_ 来假装一个序列是无限序列，比如：

```
[_] : ∀ {a} {A : Set a} → A → Colist A
[ x ] = x ∷ ♯ []
```

以及我们可以更方便地使用 inductive 版本的函数来辅助我们对 coinductive 版本的数据类型的使用：

```
map : ∀ {a b} {A : Set a} {B : Set b} → (A → B) → Coli
map f []       = []
map f (x ∷ xs) = f x ∷ ♯ map f (♭ xs)
```

而上文提到的 Costring 就是 Colist Char 而已。

# 拓展阅读

- 更推荐的 coinductive 方式

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

## 协议/License