无限大的大小与余归纳数据结构

这篇文章是写给对 Agda 有一定基础的人看的，所以我会使用一定程度的 Unicode（会尽量避免）。 当然，有基本的 Haskell 基础也能看。 本文会介绍如何使用 Agda 表达无限大的类型，并证明对它的有限使用是停机的。

这些特性 Idris 都没有。

```
{-# OPTIONS --no-unicode  #-}
{-# OPTIONS --without-K   #-}
{-# OPTIONS --copattern   #-}
{-# OPTIONS --sized-types #-}

module MuGenHackingToTheGate where

open import lib.Basics


variable i : ULevel
```

后两个 pragma 是默认打开的，我写出来是为了让它更明显。

停机性的重要性

```
module PleaseGoDieIfNotTerminate where
```

```
open import lib.types.Nat
```

如果我们可以写出不停机的函数，会怎么样？

在类型系统较弱的编程语言（Haskell，Rust）中，我们没有禁止死循环。 这很常规，谁不想写点无限运行的程序呢对吧。

在形式验证中，所有函数必须停机。否则，比如，我们可以用一个命题本身作为它自己的证明， 这样任意假命题都得证了：

```
{-# NON_TERMINATING #-}
test2 : 1 + 1 == 3
test2 rewrite test2 = idp
```

很明显这是不科学的。 这也是为什么明明 Coq 和 Agda 有着强大的多的类型系统，却『图灵不完备』的原因。

于是，我们需要保证两件事：

0. 保证我们的函数停机

1. 说服编译器，让它认为我们的函数停机

第二件事可不太容易，因为编译器判断停机性的方法很迷，请听下文分解。

传统定义

无穷大，在一般的编程语言中是没有办法表示的。 在拥有惰性求值的语言中，我们可以使用余归纳（Coinductive）数据类型表示。 比如，Haskell 语言。

```
data Nat = O | S Nat deriving (Eq, Show)
```

```
mugen :: Nat
mugen = S mugen
```

它的意义在于，考虑到自然数的减法和比较运算：

```
minus :: Nat -> Nat -> Nat
minus O _ = O
minus a O = a
minus (S a) (S b) = minus a b

lessThan :: Nat -> Nat -> Bool
lessThan _ O = False
lessThan O _ = True
lessThan (S a) (S b) = lessThan a b
```

我们发现，给定任意一个非无限大的自然数（假设它叫 `a` ），
我们都有：

- `lessThan a mugen` 返回 `True`
- `lessThan mugen a` 返回 `False`
- `minus mugen a` 返回的东西还是 `mugen`
- `minus a mugen` 返回 `O`

这正是我们想要的无限大的性质，不是吗？
有个小问题，我们不能对 `mugen` 求值，或者比较两个 `mugen` 。
这正好也符合无限大的数学意义，因为我们是不能比较这玩意的

（我记得应该是可以说无限大等于无限大的， 但这没法用 Haskell 表达出来）。

Agda 是一个编译期 call-by-name，运行时 call-by-need 的语言，所以我们很明显可以定义出无限大的自然数。

```
module TranditionalNatMuGen where
  {-# NON_TERMINATING #-}
  mugen : Nat
  mugen = S mugen
```

但是，这个函数的定义本身就是个无限循环， 我们必须要把这样的函数定义成 `NON_TERMINATING` 的，不然 Agda 会报错，说：

```
 Termination checking failed for the following functions:
   TranditionalNatMuGen.mugen
Problematic calls:
   mugen
```

然后，如果我们试图对这个 `mugen` 进行一些基本的使用：

```
  testNeq : mugen ≢ 0
  testNeq ()
```

会报错，因为 Agda 对 `NON_TERMINATING` 的定义根本不会产生编译期计算：

```
 Failed to solve the following constraints:
   Is empty: mugen == FromNat.read ℕ-reader 0
```

这很不好！我们知道这个函数不停机，但我们对它的使用却是有限的！ 如何让编译器知道我们对它的使用是有限的呢？

Agda 有个旧方法可以实现 Coindutive 数据类型，HoTT-Agda 还封装了它：

```
import lib.Coinduction as OldWayCoinduction using (
```

我上古时期还写过博客介绍它， 但是现在我已经不会再用这个库了——首先它是不安全的，借助它可以证明假命题。 其次，我们有更好的替代品。

次时代余归纳数据结构

```
module NextGenerationCoinductiveDataTypes where
```

首先，这种余归纳数据结构是需要『记录（Record）』的， 也就是只有一个数据构造器的数据类型。为了满足 `Nat` 有两个数据构造器的需求， 我们需要 `Maybe` 。

```
  Maybe : (A : Type i) -> Type i
  Maybe A = Coprod Unit A
```

在介绍余归纳的 `Conat` 前，我们先写一个我们熟悉的归纳版本的、使用记录实现的 `Conat'` ，作为一个过渡：

```
  module InductiveConat where
    record Conat' : Type0 where
      inductive
      constructor nat
      field
```

```
    succ : Maybe Conat'
open Conat'
```

我们可以做配套的转换器：

```
toNat' : Conat' -> Nat
toNat' (nat (inl unit)) = O
toNat' (nat (inr x)) = S (toNat' x)

fromNat' : Nat -> Conat'
fromNat' O = nat (inl unit)
fromNat' (S n) = nat (inr (fromNat' n))
```

然后随手写一个函数，比如除以二：

```
-- 除以二
_/2 : Conat' -> Conat'
nat (inl unit) /2 = nat (inl unit)
nat (inr (nat (inl unit))) /2 = nat (inl unit)
nat (inr (nat (inr x))) /2 = nat (inr (x /2))
```

然后证明一下这个除以二的正确性：

```
_ = idp :> (toNat' (fromNat' 10 /2) == 5)
_ = idp :> (toNat' (fromNat' 11 /2) == 5)
_ = idp :> (toNat' (fromNat' 9 /2) == 4)
_ = idp :> (toNat' (fromNat' 0 /2) == 0)
_ = idp :> (toNat' (fromNat' 1 /2) == 0)
_ = idp :> (toNat' (fromNat' 2 /2) == 1)
_ = idp :> (toNat' (fromNat' 3 /2) == 1)
```

哦，顺便证明一下这个 `fromNat'` 和 `toNat'` 的正确性吧：

```
proofNat' : forall n -> toNat' (fromNat' n) ==
proofNat' O = idp
proofNat' (S n) = ap S (proofNat' n)

proofNat'' : forall n -> fromNat' (toNat' n) ==
proofNat'' (nat (inl unit)) = idp
proofNat'' (nat (inr x)) = ap (nat ∘ inr) (proo
```

哦，再顺便玩玩 Univalence Axiom 吧，这可是难得的证明了一下同构关系啊：

```
conat-is-nat : Conat' == Nat
conat-is-nat = ua $
  equiv toNat' fromNat' proofNat' proofNat''
```

停止玩耍！回到正题。

余归纳的 `Conat` 就是把 `inductive` 修饰符换成 `coinductive` 。

定义这个数据结构：

```
record Conat : Type0 where
  coinductive
  constructor nat
  field
    succ : Maybe Conat
open Conat
```

随手定义一个转换函数：

```
fromNat : Nat -> Conat
fromNat O = nat (inl unit)

fromNat (S n) = nat (inr (fromNat n))

-- toNat 暂时定义不了
```

很常规，无非就是把 `Nat` 从 GADT 形式的 union 变成基于 `Maybe` 的 union 了， 这有什么厉害的？
之前那个无限大定义，不还是得带上 `NON_TERMINATING` 才能过？

```
{-# NON_TERMINATING #-}
mugen-bad : Conat
mugen-bad = nat (inr mugen-bad)
```

这个 `mugen-bad` 可以通过 `succ` 函数解构出里面的 `Maybe Conat` ， 然后无限解构，因为它是无穷大嘛。
有穷的数据的话，用 `nat (inl unit)` 表示 0 ， `nat ∘ succ` 表示后继操作就好了。 这也是 `fromNat` 的根据：

```
_ = fromNat 233 :> Conat
```

我们这个 `Conat` 用了 `coinductive` 修饰符，因此不能被模式匹配。 为什么我们需要禁止模式匹配余归纳数据结构呢？
我们仔细思考一下余归纳数据结构的本质。 普通的归纳数据结构，我们对它的使用方法是：

- 写代码，构造出这个数据结构（**如何构造，这是我们关心的重点**）
- 写代码，使用这个构造出来的数据结构

对余归纳数据结构，我们在 Haskell 中对它的使用方法是：

- 写代码，构造出这个数据结构，由于惰性求值，我们可以构造无限的数据结构
- 写代码，对它无限的东西进行有限地解构并使用（**如何解构，这是我们关心的重点**）

构造对应解构，归纳对应余归纳（Inductive 和 Coinductive），妙不可言。
而正是因为 Haskell 依赖了运行时的惰性求值，才可以放心地构造无限的数据结构。 而正是因为 Haskell 让我们**构造**本应该**只关心如何解构**的数据结构，余归纳数据结构才对初学者那么不直观。
我们，不如直接定义数据结构的解构方式？
我们调用 `mugen` 的时候，可以这样描述调用过程：

- 通过 `succ mugen` 解构 `mugen`
- 返回的还是一个 `mugen`

那么我们直接用这个解构的过程来描述这个 `mugen` 函数好了！

```
mugen : Conat
succ mugen = inr mugen
```

我们就直接使用这种方式定义数据的解构规则，就不要模式匹配了！

这种语法，叫**余模式匹配**（Copattern Matching），Idris 没有这个功能。要在 Idris 里使用无限的数据结构，请关掉停机检查（滑稽）。

当然，我们还是可以使用解构器（比如 `succ` ）的。 这也是为什么我们使用『记录』来定义余归纳数据结构，因为余模式匹配只能针对一组解构器，不能多组。

这样，我们定义了『解构这个 `mugen` 的规则』。我们可以对 `mugen` 进行有限的解构， 然后编译器就可以很轻送地判断我们的函数是否停机了！

不过，自然数只是一个很平凡的例子。

余归纳列表

还记得 Haskell 里的余归纳列表——流（Stream）吗？

```haskell
data Stream a = a :>: Stream a
```

我们可以构建无限的列表：

```haskell
ones :: Stream Int
ones = 1 :>: ones
```

我们可以实现 `zipWith` 、 `head` 、 `tail` 、 `take` ：

```
zipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream (
zipWith f (a :>: as) (b :>: bs) = f a b :>: zipWith f as bs

head :: Stream a -> a
head (a :>: _) = a

tail :: Stream a -> Stream a
tail (_ :>: as) = as

take :: Int -> Stream a -> [a]
take 0 _ = []
take n (a :>: as) = a : take (n-1) as
```

我们可以对无限的列表进行有限地解构：

```
λ> take 5 ones
[1, 1, 1, 1, 1]
```

我们可以实现 `fib` ：

```
fib = 0 : 1 : zipWith (+) fib (tail fib)
```

是不是呀？

但是，很明显， `ones` 、 `zipWith` 、 `fib` 都是不停机的！我们可以试着使用 Agda 的余模式匹配来看看能不能构造出无限的数据然后有限地解构。

普通的余归纳列表

```
module SimpleCoinductiveList where
  open import lib.types.List
```

先定义一个记录：

```
  record Colist {i} (A : Type i) : Type i where
    coinductive
    constructor _:>:_
    field
      cohead : A
      cotail : Colist A
  open Colist
```

我们先试试那个 `ones`，使用余模式匹配：

```
  ones : Colist Nat
  cohead ones = 1
  cotail ones = ones
```

因为这里有两个解构器，所以余模式匹配也需要匹配两次。

显而易见地，对 `ones` 调用 `cohead` 会得到 1，调用 `cotail` 会返回一个无限的列表，里面还是都是 1。 余模式匹配把这个过程非常直观地描述了！

实现 `cotake`：

```
  cotake : {A : Type i} -> Nat -> Colist A -> Lis
  cotake O as = nil
  cotake (S n) as = cohead as :: cotake n (cotail
```

试试调用：

```
_ = idp :> (cotake 1 ones == 1 :: nil)
_ = idp :> (cotake 2 ones == 1 :: 1 :: nil)
_ = idp :> (cotake 3 ones == 1 :: 1 :: 1 :: nil
_ = idp :> (cotake 4 ones == 1 :: 1 :: 1 :: 1 :
_ = idp :> (cotake 5 ones == 1 :: 1 :: 1 :: 1 :
_ = idp :> (cotake 6 ones == 1 :: 1 :: 1 :: 1 :
_ = idp :> (cotake 7 ones == 1 :: 1 :: 1 :: 1 :
```

对头！我们这个函数是停机的！对无限的数据进行有限地使用，被编译器认可了！

蛤蛤蛤。

那么，我们试试实现个 `zipWith` ？

```
cozipWith : {A B C : Type i} -> (A -> B -> C)
          -> Colist A -> Colist B -> Colist C
cohead (cozipWith f a b) = f (cohead a) (cohead
cotail (cozipWith f a b) = cozipWith f (cotail
```

看着感觉问题不大……用它写个 `fib` 试试（余模式是可以嵌套的）？

```
open import lib.types.Nat using (_+_)
{-# TERMINATING #-}
cofib : Colist Nat
cohead cofib = 0
```

```
cohead (cotail cofib) = 1
cotail (cotail cofib) = cozipWith _+_ cofib (co
```

Ayayayayaya! 我们遇到了停机问题！ 编译器不知道 `cozipWith` 返回的 `Colist` 和传入的 `Colist` 的关系（函数范围内的停机信息不会保留，这也是开头说的停机性判定迷的地方）， 导致了报错！

```
Termination checking failed for the following functions:
  cofib
Problematic calls:
  cofib
```

我们如何让编译器明确地知道， `cozipWith` 返回的 `Colist` 和参数 `Colist` 一样长呢？
我们需要一个工具来保留这个余归纳时的长度信息。

无限大的大小

这就是我写这篇文章的初衷了，因为某沙想看（捂脸）。
引入一个包（不要点进去看，命名瞎眼）：

```
open import Agda.Builtin.Size
```

我们可以使用 `Size` 来保存余归纳的数据的大小关系。 `Size` 具有以下特征：

- 不能被模式匹配
- 它的类型是 `Size` ，这个类型独立于 `Set` 的类型体系

- 考虑到 Girard Paradox，我们给定 `Size` 本身的类型是 `SizeU`
- `SizeU` 也独立于 `Set` 的类型体系
- `SizeU` 需要 `--type-in-type` 这个 pragma

- 它的构造类似自然数，不过一般的用法不一样

一般的用法是：

- **函数**拿一个隐式参数 `s`，类型是 `Size`
- 我们可以把这个 `s` 填进其他参数的类型参数里
- 返回值如果和参数大小相同
  - 就给他们配上同一个 `s`
  - 否则可以使用 `Size<`、`↑` 等函数创造新的 `Size` 并描述和 `s` 的关系
- `Size` 的默认值是无限大

我们先给 `Colist` 加上 `Size`：

```
open import lib.types.Nat
```

```
record Colist {i} (s : Size) (A : Type i) : Type i
  coinductive
  constructor _:>:_
```

然后， `cohead` 常规：

```
  field
    cohead : A
```

但是 `cotail` 就不一样了——我们需要弄一个隐式参数，它的类型是『一个小于 `s` 的 `Size`』， 但值是多少就让 Agda 自己推：

```
    cotail : {ss : Size< s} -> Colist ss A
```

暴露定义，泛化一个 `Size`：

```
open Colist
variable s : Size
```

`ones` 的定义将不受影响：

```
ones : Colist s Nat
cohead ones = 1
cotail ones = ones
```

`cotake` 由于处理的对象的长度是无所谓的，所以我们不需要使用 `Size`，就直接使用无限大这个 `Size`：

```
open import lib.types.List
cotake : {A : Type i} -> Nat -> Colist ∞ A -> List
cotake O as = nil
cotake (S n) as = cohead as :: cotake n (cotail as)
```

测试：

```
_ = idp :> (cotake 1 ones == 1 :: nil)
_ = idp :> (cotake 4 ones == 1 :: 1 :: 1 :: 1 :: ni
_ = idp :> (cotake 7 ones == 1 :: 1 :: 1 :: 1 :: 1
```

`cozipWith` 可以使用一个 `Size` 来保留『返回的 `Colist` 长度和参数一致』这一细节，通过手动指定他们的 `Size` 都是同一个变量：

```
cozipWith : {A B C : Type i} -> (A -> B -> C)
          -> Colist s A -> Colist s B -> Colist s C
cohead (cozipWith f a b) = f (cohead a) (cohead b)
cotail (cozipWith f a b) = cozipWith f (cotail a) (
```

于是， `cofib` 就安全了！

```
cofib : Colist s Nat
cohead cofib = 0
cohead (cotail cofib) = 1
cotail (cotail cofib) = cozipWith _+_ cofib (cotail
```

测试！

```
_ = idp :> ((cotake 1 cofib) == 0 :: nil)
_ = idp :> ((cotake 5 cofib) == 0 :: 1 :: 1 :: 2 ::
```

试试更大的数据？

```
_ = idp :> ((cotake 15 cofib) == 0 :: 1 :: 1 :: 2 :
        3 :: 5 :: 8 :: 13 :: 21 :: 34 :: 55 :: 89 :
        144 :: 233 :: 377 :: nil)
```

再大一点就没法自动推导啦，我们得手写，但检查已有的定义也是没有问题的：

```
_ = idp :> ((cotake 28 cofib) == 0 :: 1 :: 1 :: 2 :
        3 :: 5 :: 8 :: 13 :: 21 :: 34 :: 55 :: 89 :
        144 :: 233 :: 377 :: 610 :: 987 :: 1597 ::
        4181 :: 6765 :: 10946 :: 17711 :: 28657 ::
        75025 :: 121393 :: 196418 :: nil)
```

好啦，这差不多就是 Agda 对余归纳数据结构的（几乎）全部的知识了。 是不是很强大呢？
不过像携带函数作用域的信息并到处传递，还是用 refinement type 爽一点， F⋆ 的这个就很好，它的 `Lemma` 就是 `x:unit{}` 的语法糖。

谢谢阅读。

创建一个 issue 以申请评论

Create an issue to apply for commentary