# copattern 的其他意义

这篇文章放飞自我，会用一些英文术语了，因为有朋友反应中文看不懂。

```
{-# OPTIONS --no-unicode #-}
{-# OPTIONS --without-K  #-}
{-# OPTIONS --copattern  #-}
module CopatternIsMoreThanCoinduction where

open import lib.Basics

variable a b : ULevel
variable
  A : Type a
  B : Type b
```

上次心血来潮写了篇文章：

```
import MuGenHackingToTheGate using ()
```

URL 玩了个命运石之门的梗，然而没有人吐槽我，这另我十分伤熏。 并且，有个 Haskell 微信群的朋友对我进行了这样的反

问：

> 如果 Haskell 支持全局模式匹配，也就是说 `Just bla = Just 233`
>
> 这样的代码可以通过编译，那是不是可以说 Haskell 也有余模式了？

我理一下他的意思。实际上他是指：

```
bla :: Int
Just bla = Just 233
```

然后可以拿到 `bla` 的值是 233。

然而这 too simple，他产生这样的误会，很明显是我没说清楚（就是该强调的重点没有强调）。 我整节数学课都在反省自己，最后才理解了他到底是产生了什么误解， 以及怎么跟他解释他这个是错的。

首先，copattern 是通过定义函数的返回值被解构的行为来定义 coinductive 数据结构的语法， 这并不是一个简单的语法糖，因为它具有辅助 termination check 的作用， 前文给出的 `ones` 和 `cofib` 就已经非常说明问题了。

我们使用 copattern 定义函数时，函数体是每个 destructor 返回的东西。因此， 即使我们忽略那位读者的两个过于明显的错误（ `Maybe` 有两个 constructor， 因此不能只使用『一组 destructor』定义它；以及用 copattern 定义的函数返回的都是一个 `record`， 。这个问题要是问得有诚意一点的话，应该使用 `Identity` 类型。算了，我们还是原谅他吧）， 我们假设他口中的 `Maybe` 只有 `Just` 这一个 constructor 和 destructor。

我们也应该这样定义这个函数：

```
bla :: Maybe Int
Just bla = 233
```

首先，返回类型是应该『被 destruct』的对象—— `Maybe`，以及函数体是 destruct 后返回的东西——233。 写成 Agda，也是可以的。先定义这个很傻的 `Maybe`：

```
module StupidDefinition where
  record Maybe {i} (A : Type i) : Type i where
    inductive
    field Just : A
  open Maybe
```

然后写出 `a`：

```
  bla : Maybe Nat
  Just bla = 233
```

LGTM!

## 余模式定义 `record`

Agda 默认 `record` 构造语法不是一般的丑，因此我们有了 `constructor` 关键字来自定义构造函数的语法（注意是自定义语法，不是自定义构造函数的名字，2333）。

有个 Dark♂ 问题，我们在定义带函数成员的 `record` 的时候，非常非常的痛苦。 要么写一个辅助函数填进去，要么使用 lambda，不能使用函数定义（就是对参数模式匹配什么的） 的语法来定义这种函数成员。

一个典型的例子：各种 typeclass。

关于 Agda 的 typeclass 的详细解释，请看这篇文章：

```
import Typeclassopedia using ()
```

Agda 拥有验证各种 Law 的能力，但这不是重点，我就定义个简单的。

```
module CopatternWithRecord where

  record Monoid {i} (A : Type i) : Type i where
    inductive
    field
      mempty : A
      _<>_   : A -> A -> A
```

我们如果使用平常的方法定义 `Monoid` 的类型实例，很猥琐：

```
  open import lib.types.List


  module StupidMonoid where
    {-# TERMINATING #-}
    _ = unit where
        open Monoid {{ ... }}
        instance
          {-# TERMINATING #-}
          ListMonoid : Monoid (List A)
          ListMonoid = record
            { mempty = nil
            ; _<>_ = λ where
```

```
  ; _<>_ = λ where
    nil b -> b
    (a :: as) b -> a :: (as <> b)
  }
```

（Instance Argument 的查找似乎出了点问题，只能把这个定义藏在一个 `where` 里面。 但我相信这不影响阅读，毕竟只是在外面加了一层 `_ = unit` ）

当然了，出现了跨函数定义，因为 `<>` 的定义和使用（也就是调用 `ListMonoid` ）产生了递归。 这导致我们必须手动让 termination check 闭嘴。

我们可以使用辅助函数，which means 需要另外再给这个函数想一个名字：

```
module MonoidWithHelperFunction where
    ListMonoid : Monoid (List A)
    ListMonoid = record
      { mempty = nil
      ; _<>_ = whoAmI?Dunno,LOL
      }
      where
        whoAmI?Dunno,LOL : (a b : List A) -> List
        whoAmI?Dunno,LOL nil b = b
        whoAmI?Dunno,LOL (a :: as) b =
          a :: (whoAmI?Dunno,LOL as b)
```

这种时候，除了 copattern 还有什么别的语言特性能用吗？

```
open Monoid {{ ... }} public

instance
  {-# TERMINATING #-}
  ListMonoid : Monoid (List A)
  Monoid.mempty ListMonoid = nil
  Monoid._<>_   ListMonoid nil b = b
  Monoid._<>_   ListMonoid (a :: as) b = a :: (as
```

至于这个的 termination 问题我就不管它了，有一万种方法说服编译器这个东西是停机的。

简单的使用：

```
_ = mempty <> mempty :> List Nat
```

Lambda 表达式也是可以用 copattern 的：

```
implementation : Monoid (List Nat)
implementation = λ where
  .mempty -> nil
  ._<>_ a b -> a ++ b
```

偷懒用库函数定义啦。

---

创建一个 issue 以申请评论

Create an issue to apply for commentary