Agda 中的证明，从零到一

2017, Nov 1 by Tesla Ice Zhang

类型则命题，程序则证明。这句话表达了定理证明的一个很重要的思想。

我一开始就没有搞懂这句话在说什么。 在我自认为搞懂的时候，我把我以前没有搞懂的原因归结为我看的教程太垃圾了。

一开始我理解这个问题的同时，我以为我也理解了之前一个 Haskell 关于 `IO Monad` 的问题，但实际上不是我想的那样。 Haskell 的 `IO Monad` 的实现 比我想象的要复杂一些，因此本文不谈 Haskell 。

前置知识

这是一篇面向略懂 dependent type 的人的定理证明教程，然后你要看得懂类 Haskell 的语法。 因为我在学这个的时候就是只会点 Haskell ，然后用过 GADT 和 type family 模拟过 dependent type 。

给出一些参考资料：

- 一个比较简单的 Haskell 入门教程 *Learn you a Haskell*
- 虎哥介绍的 GADT ，讲的很精彩
- 介绍 GADT 和 dependent type 的 CodeWars Kata: Singletons
- 介绍 GADT 的 CodeWars Kata: Scott Encoding

声明在前面

由于 Agda 语言的特殊性，本文将使用 LaTeX 和代码块来共同展示代码。 前者是为了保证字符的正确显示，后者是为了方便读者复制代码。

本文不讲 Agda 基本语法和 Emacs 的使用。可能以后会有另外的文章。

本文主要内容是帮助一个没接触过定理证明但是接触过 dependent type 的人（这就是我接触定理证明之前的状态）理解一个非常非常简单的定理证明的例子。

如何理解定理证明

首先，我们已经知道，我们这是要用类型表达命题，类型对应的实现来证明这个命题的正确性。

命题中的基本元素一般是值的类型(而且很多时候都是代数数据类型)，也就是 $p \rightarrow q$ 的那个 p 或者 q 。 而这个 $\rightarrow$ 对应的就是 “函数” 这一概念，它组合了两个类型，表达了 “蕴含” 这一数理逻辑中的概念。

但是这是为什么呢？

因为，当 $p \rightarrow q$ 存在后，**只要你有一个 p，你就能通过这个 $p \rightarrow q$ 拿到一个 q**。

比如，我实现了一个这样的类型的函数：

$$(p_0 \equiv q_0) \rightarrow (q_1 \equiv p_1)$$

```
(p₀ ≡ q₀) → (q₁ ≡ p₁)
```

那么这个函数的实现就是

> 如果 $(p_0 \equiv q_0)$，则 $(q_1 \equiv p_1)$

，或者说，

> $(p_0 \equiv q_0) \rightarrow (q_1 \equiv p_1)$

这个命题的证明。

再比如，我实现了一个这样的类型的函数：

$$p \rightarrow q \rightarrow r$$

```
p → q → r
```

那么这个函数的实现就是

> 如果 p 成立，那么 “如果 q 成立，那么 r 成立” 这一命题成立

，或者说，

> $p \to (q \to r)$

这个命题的证明。

其实我们原本想表达的意思是

> $p \land q \to r$

但是这个 $\land$ 关系暂时没讲所以先就这样。

这就是 “类型则命题，程序则证明” 的含义。

在 Agda 中，上面的代码应该写成这样：

$$\text{proof} : \{p\ q\ r : \text{Set}\} \to p \to q \to r$$

```
proof : {p q r : Set} → p → q → r
```

下面我们看一些实例。

refl 与相等性

之所以我没有再学习 Idris 就是因为那些教程没说 `Refl` 是啥 (Idris 叫 `Refl` ， Agda 叫 `refl`) 就直接在代码里面用了，我看的时候就一脸蒙蔽，还以为是我智商太低没看懂他 implicit 的东西。 但是好在我看了一坨很友好的 Agda 代码后民白了。

首先，我们可以定义这样一个用来表示相等关系的 GADT ，它对于任何一个 Level 的任何一个实例都成立。 这里我们用了 Universal Polymorphism 表达这个 “对于任何一个 Level “ 的概念。

然后我们使用 `refl` 这个类型构造器表达 “这个相等关系成立” 这一事实。

我们用 $\equiv$ 表示他（标准库的定义在 `Agda.Builtin.Equality` 中）：

$$\begin{aligned}&\text{data}\ \_\equiv\_\ \{a\}\{A : \text{Set}\ a\}(x : A) : A \to \text{Set}\ a\ \text{where}\\&\quad \text{refl} : x \equiv x\end{aligned}$$

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

如果你看不懂这个类型签名也没有关系，只需要接受”这个 GADT 只有一个叫 `refl` 的类型构造器”这一事实就好了。

然后我们可以用它进行一些证明。比如我们来证明相等性的传递性，也就是

> 如果 $a \equiv b$ 并且 $b \equiv c$，那么 $a \equiv c$

。然后我们来看看这个命题对应的类型：

$$\_\leftrightarrows\_ : \{A : \mathrm{Set}\}\{a\ b\ c : A\} \to a \equiv b \to b \equiv c \to a \equiv c$$

```
_⇆_ : {A : Set} {a b c : A} → a ≡ b → b ≡ c → a ≡ c
```

那么我们要怎么实现它，也就是证明它呢？

我一开始写下了这样的东西：

$$\_\leftrightarrows\_\ ab\ bc = ?$$

```
_⇆_ ab bc = ?
```

然后我就不知道该怎么办了。

事实上，这个原本就很简单的证明被我想复杂了。 因为这个定理是不证则明的，那么我们要如何表达，如何通过 `ab`, `bc` 这两个模式匹配出来的结果进行变换得到这个不证则明的定理呢？

首先这个模式匹配的参数就不应该这样通配地用 `ab`, `bc` 来表达。 我们应该把这两个相等关系 (他们的本质是 GADT) 给模式匹配出来。

由于直接写 `ab`, `bc` 什么都得不出来，我于是尝试将 `ab bc` 用模式匹配消耗掉，然后 Agda 直接在右边给我自动填入了 `refl` ，然后好像就 Q.E.D 了：

$$\_\leftrightarrows\_\ \ \mathrm{refl}\ \ \mathrm{refl} = \mathrm{refl}$$

```
_⇆_ refl refl = refl -- 编译通过！
```

这是为什么呢？我们来分别看下这两种写法的含义。

使用 ab bc

这样的话实际上是把 $a \equiv b$ 和 $b \equiv c$ 两个条件当成了”变量”而不是作为”条件”。 也就是说，当使用 `ab bc` 时，右边就需要”通过 $a \equiv b$ 和 $b \equiv c$ 这两

个条件，再对这两个条件套用一些变换，得出 $a \equiv c$”。

在这个时候，编译器并没有把 $a \equiv b$ 和 $b \equiv c$ 当成既成条件，而是当成了“变量”。

这就回到了我们原本的需求，我们原本就是需要写出一个 $a \equiv b \land b \equiv c \to a \equiv c$ 的变换。

如果要用变换强行实现的话，可以使用 `with` 语句（就是 Agda 的 `case of`，但是和 Haskell 那个又不一样）把这两个变量模式匹配出来，然后直接得证。 这里给出一个代码实现。

$$
\begin{array}{ll}
\_\leftrightarrows_1\_\ ab\ bc\ \text{with}\ ab & |\ bc \\
\ldots \quad\quad |\ \ \text{refl} & |\ \text{refl} = \text{refl}
\end{array}
$$

```
_⇆₁_ ab bc with ab   | bc
...           | refl | refl = refl
```

这种方法和下面的做法是等价的。

如果你没有看懂这一坨，可以尝试继续读下去，说不定看完下面那坨你就懂了。

使用 refl

由于 $a \equiv b$ 已经是一个条件了，我们直接把它的值取出来。 这时，右边的代码就 **已经是建立在 $a \equiv b$ 和 $b \equiv c$ 这两个既成条件下** 的了，因此这时 Agda 已经认为 `a` `b` `c` 三者相等了。

利用这一点，我们直接使用 `refl` 是没有问题的。

$$\_\leftrightarrows_0\_\ \text{refl}\ \ \text{refl} = \text{refl}$$

```
_⇆₀_ refl refl = refl
```

顺带一提

当然我们也可以这样写，这是一个语法糖：

$$\text{refl} \leftrightarrows_0 \text{refl} = \text{refl}$$

```
refl ⇆₀ refl = refl
```

之前那个比较 trivial 的模式匹配也可以这样写：

$$\begin{array}{lll} ab \leftrightarrows_1 ab \,\mathrm{with}\, ab & | \; bc \\ \ldots \quad\quad\quad | \;\; \mathrm{refl} & | \; \mathrm{refl} = \mathrm{refl} \end{array}$$

```
ab ⇆₁ bc with ab   | bc
...          | refl | refl = refl
```

另一个例子

现在你肯定有点感觉了，但是这个例子太 trivial 你又感觉自己有点没懂， 那么我们再来看看这个稍微复杂点的例子帮你加深一下理解。

首先：

$$\forall\ \{a\}\ \{A : \mathrm{Set}\, a\}$$

表示 Universal Polymorphism 。然后考虑一个函数，我们有：

$$\begin{array}{l} \ggg : \forall\ \{a\ b\}\ \{A : \mathrm{Set}\, a\}\ \{B : \mathrm{Set}\, b\}\{m\ n\}\{f : A \to B\} \to m \equiv n \to f\ m \equiv f \\ \ggg \mathrm{refl} = \mathrm{refl} \end{array}$$

```
⋙ : ∀ {a b} {A : Set a} {B : Set b} {m n} {f : A → B} → m ≡ n →
⋙ refl = refl
```

和上面一样，在建立了 $m \equiv n$ 的基础上，可以直接用 `refl` 表达他们对于同一个函数应用的结果相等。

我是在 这个 StackOverflow 问题 里看到这个代码的，下面唯一的回答里面还有更多的解释。

这个我就暂时不作过多讲解了，以后再说。

结束

这个证明太简单了，只有一步，没有什么实际意义，仅用于入门理解。 下一篇文章我们将会进行一个稍微复杂点的关于与或关系的证明。

我说完了。

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License

|