Home Archive Resume LLVM# PRs Gists LAgda About

Agda 中的证明，从一到一点五

2017, Nov 2 by Tesla Ice Zhang

上一篇我们说到了一个只有一步的证明，这一篇我们来看一个稍微复杂点的，组合命题的例子。 到目前为止，按理说所有的字符都还能正常显示。

为什么是一点五？看完你就知道啦。

前置知识

- 上一篇文章

以及，由于 Agda 语言的特殊性，本文将继续使用 LaTeX 和代码块来共同展示代码。 代码块唯一的作用在于便于复制，主要的呈现途径为 LaTeX 。 ~~（其实是因为我的手机显示不出来很多字符，我又要自己看自己写的东西）~~

关于复合命题

这里修正一个概念。 前文说的 “条件” ，即前文一直强调的 “类型则命题” 中命题的最基本组成元素（好像 Wikipedia 上也称之为 “命题变元” ，反正我对这个名称不负责，就是用来表示命题 $p \to q$ 中的 `p` 和 `q` 的东西）， 其实也是一种命题，而我之前称为命题的东西则是 “复合命题” 。

下文将使用 “命题” 统称他们。

介绍符号

都是初中数学里面的，并且是只需要小学数学就可以看懂的符号。

与和或

我们知道，门电路里面都有与门和或门，对应逻辑上的与和或。与的符号是：

$$\wedge$$

，或的符号是：

$$\vee$$

。比如，$p \wedge q \to r$ 表示 p 和 q 都必须成立， r 才成立。而 $p \vee q \to r$ 表示 p 和 q 中任意成立一个， r 就成立。

充要条件

我们知道，如果两个条件 p q 能使 $p \to q$ 和 $q \to p$ 同时成立，我们称他们互为充要条件，使用：

$$\Leftrightarrow$$

表示，比如 $p \Leftrightarrow q$ 。

我们将在接下来的代码里面使用这些符号。

定义 GADT

首先定义 ∧ 对应的 GADT ：

$$\begin{gathered}\text{data }\ \_\wedge\_\ (P\ Q : \text{Set}) : \text{Set where} \\ \wedge{-}\text{intro} : P \to Q \to (P \wedge Q)\end{gathered}$$

```
data _∧_ (P Q : Set) : Set where
  ∧-intro : P → Q → (P ∧ Q)
```

这个命题是两个其他命题的组合，它拿到两个命题变成一个新命题。 这也体现在 Agda 代码中， `_∧_` 这个类型拿到两个 `Set` 作为 类型 _∧_ 的参数， 返回一个新类型。 对应的类型构造器我们称之为 $\wedge-\text{intro}$ 。

有了这个类型，我们首先可以做一些很简单的证明。

例〇：充要条件

比如，根据充要条件 $(p \to q) \wedge (q \to p)$ 的定义，我们可以把它表达成一个函数：

$$\_\Leftrightarrow\_ : (P\ Q : \text{Set}) \to \text{Set}$$
$$p \Leftrightarrow q = (p \to q) \wedge (q \to p)$$

```
_⇔_ : (P Q : Set) → Set
p ⇔ q = (p → q) ∧ (q → p)
```

这里我们在函数体（证明）里面使用了 $\to$ ，这样的话 $(p \to q)$ 就是一个类型为 Set_1 的东西。 因此，这实际上是一个 “命题组合” ，有种 “高阶函数” 的味道（顺带一提，这个名词也是我为了便于理解自己编的，不知道有没有其他人在用（顺带一提，类型的阶（顺带一提， Agda 中表示类型的阶的类型正好是 `Level` ，中文意思就有阶的意思， 因此这个说法可以说是很通用了 ）在 dependent type 里面已经变得很模糊了，因此这个 “高阶” 的比喻是不太恰当的，这里就拿 Haskell 之类的简单语言的概念将就一下 ）)。

再根据前文已经讲过的：

> 只要有 $p \to q$ 这个函数成立，那么就证明了 “ $p \to q$ “ 这个命题

这个函数的作用便变得很清晰了。 不理解没关系，下面会用到这个东西，然后你或许能从它的应用看懂它的意义。

另外，看到没有？函数体（证明）（下文不再进行这样的强调，感觉很辣鸡）和定义 $(p \to q) \land (q \to p)$ 写起来都是完全一样的。 这里可以体现一些 Agda 语言的优势，就是因为 Unicode 语法的存在，它可以把代码写的很接近数学语言。

不过这并不代表 Agda 就只能用于学术，毕竟类型安全的社区和人气火爆的社区结合起来才是最好的， Idris 都用强大的 ffi 和官方强推的 `Control.ST` 了，为什么 Agda 不能写成 imperative language 呢。

例一：定义

比如，在 $p \land q$ 成立的时候， `p` 和 `q` 分别成立（就是 $\land$ 的定义啦，很简单的）。 用数学语言表达的话，就是（几乎就是废话）：

$$p \land q \to p$$
$$p \land q \to q$$

写成代码的话，就是（这里关于这个证明讲的比较略，是因为下文有个更详细的讲解， 已经完全覆盖了这个证明所需要用到的知识，这个证明放在前面只是因为它本身很简单，用 Haskell 知识即可理解， 如果读者看不懂这个证明可以先看后面的，不过我觉得应该都看得懂，因为它太简单了）：

$$\text{proof}_3 : \forall\{P\ Q\} \to (P \land Q) \to P$$
$$\text{proof}_3\ \ (\land{-}\text{intro}\ p\ q) = p$$

$$\text{proof}_4 : \forall\{P\ Q\} \to (P \land Q) \to Q$$
$$\text{proof}_4\ \ (\land{-}\text{intro}\ p\ q) = q$$

```
proof₃ : ∀ {P Q} → (P ∧ Q) → P
proof₃ (∧-intro p q) = p


proof₄ : ∀ {P Q} → (P ∧ Q) → Q
proof₄ (∧-intro p q) = q
```

例二(详)：交换律

然后还有一个很简单的例子——交换律（ Commutative Law ）。 用数学语言表达的话，就是（几乎也是废话）：

$$(P \land Q) \Leftrightarrow (Q \land P)$$

这个命题写成 Agda 代码，就是这样的类型（我们称之为 $\land{-}\text{comm}$ ）：

$$\land{-}\text{comm} : \forall\{P\ Q\} \to (P \land Q) \Leftrightarrow (Q \land P)$$

```
∧-comm : ∀ {P Q} → (P ∧ Q) ⇔ (Q ∧ P)
```

这里我们就已经使用到之前的定义—— $\Leftrightarrow$ 啦。

如何证明它呢？

首先，我们的证明需要返回一个由 $\Leftrightarrow$ 组合的两个类型（命题）。 由于这个组合类型是一个由 $\land$ 组合而成的两个类型，我们可以先把类型构造器写上，然后两个参数留白：

$$\wedge\text{−comm} = \wedge\text{−intro} \; ? \; ?$$

```
∧-comm = ∧-intro ? ?
```

我们发现，在 `p q` 两个变量可以互相交换的情况下， 这两个参数的类型（复合命题）都是 $(p \wedge q) \rightarrow (q \wedge p)$ 。
因此，为了代码复用，我们不妨把这两部分提取出来，作为一个单独的命题去证明它。 这个命题写成 Agda 代码，就是:

$$\wedge\text{−comm}' : \forall \{P \; Q\} \rightarrow (P \wedge Q) \rightarrow (Q \wedge P)$$

```
∧-comm′ : ∀ {P Q} → (P ∧ Q) → (Q ∧ P)
```

它的第一个显式参数（隐式参数就自动传递了，我们不用管）是 $(P \wedge Q)$ ，我们可以使用模式匹配将它拆开:

$$\wedge\text{−comm}' \; (\wedge\text{−intro} \; p \; q) = \; ?$$

```
∧-comm′ (∧-intro p q) = ?
```

然后我们把 `p q` 换个顺序，重新使用类型构造器把它们组合起来:

$$\wedge\text{−comm}' \; (\wedge\text{−intro} \; p \; q) = (\wedge\text{−intro} \; q \; p)$$

```
∧-comm′ (∧-intro p q) = (∧-intro q p)
```

然后再把这个命题填入刚才的 $\wedge\text{−comm}$ 中:

$$\wedge\text{−comm} = \wedge\text{−intro} \; \wedge\text{−comm}' \; \wedge\text{−comm}'$$

```
∧-comm = ∧-intro ∧-comm′ ∧-comm′
```

然后我们就可以喊 Q.E.D. 啦。

例三：结合律

这个结合律（ Associative Law ）的例子其实已经不是例子了（因为我不想详细讲（因为思路和交换律差不多）），我就只给出类型签名就可以了。

$$\wedge\text{–assoc} : \forall\{P\ Q\ R\} \to (P \wedge (Q \wedge R)) \Leftrightarrow ((P \wedge Q) \wedge R)$$

```
∧-assoc : ∀ {P Q R} → (P ∧ (Q ∧ R)) ⇔ ((P ∧ Q) ∧ R)
```

为什么我只给类型签名呢？因为这个证明啊，

> 即得易见平凡，仿照上例显然。留作习题答案略，读者自证不难。

为什么是一点五

因为原计划是把 $\wedge$ 和 $\vee$ 放在一起讲的，但是我发现到目前为止的证明在模式匹配上都只有一个分支，到了 $\vee$ 就有两个了，因此关于它和另外几个有两个分支的证明就单独再开一篇吧 ~~（其实是因为这篇写长了，我写博客会控制篇幅的）~~。

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

| | |