后记

想说的差不多都说了，引用 Belleve 大神对推理系统和程序语言相似性的解释[0]作为结语吧。

- 程序语言的语言构造同构为推理系统的推理规则
- 程序的类型同构为逻辑命题
- 闭合程序（不依赖环境的程序）可以同构为一条定理的证明过程，其类型就是一条定理
- 逻辑上下文同构为自由变量类型指派
- Lambda 演算和 Gentzen 的自然演绎
  - 函数调用就是蕴含消除
  - 函数抽象就是蕴含介入
  - 参数多态就是全称量化
  - 模板类型就是谓词
  - 结构类型就是合取
  - 联合类型就是析取
  - 收参数但不返回就是否定
  - 高洋上[1]的 call/cc 就是双重否定消除
- SK 组合子演算同构为直觉 Hilbert 推理系统
  - S 和 K 就是演算系统的两条公理

[0]: https://www.zhihu.com/question/22959608/answer/24770830
[1]: 高端洋气上档次

目录

序一

我初次接触“形式验证”一词，是我受冰封邀请为他的编程语言写一个更好的前端的时候。我在代码中加入大量的断言，美其名曰“防御式编程”，而冰封看到后把我的断言删了。他告诉我形式验证能够绝对地证明程序的正确性，有了这种思想和方法，任何断言都只是徒增运行时开销。

当时我并不完全理解他所说的许多术语，也对“证明”一词将信将疑，但是我仍然抱着浓厚的兴趣，尝试打开形式验证的大门。但直到冰封的这本小册子出现，而我有幸成为“内测成员”，我才得以真正开始对形式验证的学习。在一系列简单明了的文字中，我理解了什么是“类型即命题，程序即证明”，也理解了如何证明而非测试程序的正确性，为什么冰封要删我的断言——为什么要断言一个不可能存在的异常不存在呢？

尽管软件工程中没有银弹，尽管多数人没有机会深入研究理论，但是学习形式验证确实能开拓视野：不了解形式验证的人也可以在工作中游刃有余，但是了解形式验证就有机会更上一层楼。冰封撰写的这本小册子为广大没有英文阅读和数学基础的同学提供了一扇门一个桥梁。也许你看不懂数学名词，读不了外语 paper，但你仍然能愉快地阅读下去。希望这本小册子能将你带入形式验证的世界。

——ICEY，2018 年秋，中国青岛

延伸阅读

PLFA[0]、Alfa[1]、HoTT-Agda[2]、Agda Wiki[3]（Agda 资源最丰富的网站）、类型即命题，程序即证明[4]、Agda 文档[5]、源码[6]和标准库[7]、一篇用 Literate Agda 写的教程[8]（代码过时，内容质量高）。

[0]: https://plfa.github.io/
[1]: http://www.cse.chalmers.se/~hallgren/Alfa/
[2]: https://ncatlab.org/homotopytypetheory/show/Agda
[3]: http://wiki.portal.chalmers.se/agda/pmwiki.php
[4]: http://scienceblogs.com/goodmath/2009/11/17/types-in-haskell-types-are-pro/
[5]: https://agda.readthedocs.io/
[6]: https://github.com/agda/agda
[7]: https://github.com/agda/agda-stdlib
[8]: http://people.inf.elte.hu/divip/AgdaTutorial/Index.html

其他定理证明器

仅为笔者个人推荐。

F ★_____ https://www.fstar-lang.org/
Idris _____ https://www.idris-lang.org/
Isabelle__ https://isabelle.in.tum.de/
Coq _____ https://coq.inria.fr/
Shen _____ http://www.shenlanguage.org/

术语表

右边的数字是第一次出现的页码。

序二

认识 qlbf，是在知乎上。我因为吹爆了 JetBrains 的产品，引得同为 JetBrains 粉丝的 qlbf 的关注，循着我主页上的群号加入了我的群。从此事至今，约莫一年有余。

其间又幸得 qlbf 组织起 ZJU Lambda 的周练活动，我接触了 CodeWars，并于其上习得诸多用 Haskell 作证明的 practice，并进而接触了 Coq、Idris、Agda 以及它们在中国的社区。可以说，qlbf 是我在函数式编程与形式化验证上的引路人。

在我看来，形式化的意义巨大，它能直接将逻辑思辨从手工作坊时代演进到工业化、自动化时代。即使自动化的推理看起来依然遥远，但自动化的、坚实的验证，在命题和证明越来越复杂、也越来越多的趋势下，不可谓意义不大。

然而，形式化验证在中国（也许在世界），并不像一般编程技术那样普及。数学界的人往往对其表示不解，持友善态度者寥寥，计算机工业界的应用，也往往限于少数领域的少数项目，这不能不说是一种遗憾。

Qlbf 的这本小册子，不能说内容丰富、深入，但姑且算是五脏俱全，堪为基本的、可操作的入门导引。万事开头难，倘若能引人入门，便有很大意义。

星星之火，可以燎原。希望 qlbf 不仅作我的引路人，也作世界的引路人。

——Colliot，2018 年国庆假期，于紫金港

引言

看看这本书是否适合你

类型即命题，程序即证明（Types are propositions, programs are proofs）。

这句话言简意赅地描述了使用编程语言证明定理的核心思想，也叫**柯里-霍华德同构**（Curry-Howard Isomorphism）。

笔者在开始专门学习形式验证前多次试图理解这句话，一直没有看懂这句话到底在说什么，即使这本该是很直观的道理——就像“单子和可应用函子都是一个自函子范畴上的幺半群”这句话一样。

本书是笔者的一系列博客[0]的完全重写，删除了大量冗余的内容，增加了更多原本缺失的内容。主要内容是帮助对 Haskell 和函数式编程有基本认知的读者（也就是笔者接触定理证明之前的状态）理解定理证明的原理。也就是说，要读懂本书，读者需要事先了解这些编程概念：

完成目标 1：

```
suc (n + m) ≡⟨ cong suc (comm n m) ⟩ { }2
```

目标 2 等式左边为 `suc (m + n)`，由 $\texttt{lemma}_1$ `m n`（也就是 `suc (m + n) ≡ n + suc m`），可以完成目标 2：

```
suc (n + m) ≡⟨ lemma₁ m n ⟩ { }3
```

现在目标 3 等式左边已经和等式右边完全一致、可以自动证明或者直接填入 `refl` 了。但为了可读性，我们可以把结论再写一遍后配合 `_QED`（这样可以获得一个特化的 `refl`，和直接写 `refl` 语义完全相同）来表达“证毕”。方便起见，我们使用自动证明来填写结论。向目标 3 填入 `?`，`QED`，得到目标 4 然后直接自动证明：

```
m + suc n QED
```

ὅπερ ἔδει δεῖξαι[0].

完整 `comm` 实现（可以换行来避免单行过长）：

```
comm : ∀ n m → n + m ≡ m + n
comm zero n = sym (lemma₀ n)
comm (suc n) m = suc n + m
  ≡⟨ refl ⟩ suc (n + m)
  ≡⟨ cong suc (comm n m) ⟩ suc (n + m)
  ≡⟨ lemma₁ m n ⟩ n + suc m
  QED
```

以上证明全过程可以参考这个视频（体积约 `900kb`）：
http://ice1000.org/assets/plus-comm.mp4

[0]: “证明完毕”的希腊语原文。

根据内置 Nat 类型的_+_运算符的定义：

```
_+_ : Nat → Nat → Nat
zero + n = n
(suc n) + m = suc (n + m)
```

有引理 n+0 ≡ n，方便起见称之为 $lemma_0$。使用 cong 函数归纳证明：

```
lemma₀ : ∀ n → n + 0 ≡ n
lemma₀ zero = refl
lemma₀ (suc n) = cong suc (lemma₀ n)
```

同样可以归纳证明的还有(n+m)+1 ≡ n+(m+1)：

```
lemma₁ : ∀ n m → suc (n + m) ≡ n + suc m
lemma₁ zero _ = refl
lemma₁ (suc n) m = cong suc (lemma₁ n m)
```

我们可以使用这两个引理证明 n+m ≡ m+n：对一个参数分类讨论，当它为 0 时原命题变成 n ≡ n+0，和 $lemma_0$ 对称，其他情况则想办法归纳：

```
comm : ∀ n m → n + m ≡ m + n
comm zero n = sym (lemma₀ n)
comm (suc n) m = { }0
```

考虑目标 0 的类型 suc n + m ≡ m + suc n，我们可以用递等式证明它。根据_+_的定义，原式左边 suc n + m 等于 suc (n + m)。于是可以通过对目标 0 进行“完成目标”，变成：

```
suc n + m ≡⟨ refl ⟩ { }1
```

目标 1 等式左边就变成了 suc (n + m)。由一致关系，递推假设 comm n m（也就是 n + m ≡ m + n）可以转换成 suc (n + m) ≡ suc (m + n)。根据这个等式，可以

0. 代数数据类型（Algebraic Data Type）
1. 模式匹配（Pattern Matching）
2. 简单的命题逻辑，全称量词和存在量词
3. 数学归纳法
4. 少量 Haskell 知识：函数、运算符、数据类型
5. 包管理工具 Stack/Cabal，二选一
6. 柯里化（Currying）
7. 一种**快捷键记法**[1]，形如 C-c C-l

使用“代数数据类型”一词而不是缩写 ADT 是因为 ADT 还可以指“抽象数据类型（Abstract Data Type）”。代数数据类型的应用非常广泛，Haskell 中的 **data**，Rust 中的 **enum**，Scala 和 Kotlin 中的 **sealed class**，F#和 OCaml 中的 **type** 声明的都是代数数据类型。本书将会在代数数据类型的基础上讲解 **GADT**（泛化代数数据类型，Generalized Algebraic Data Type）。

考虑到 Agda 基本部分的语法非常类似 Haskell，本书用了 Haskell 的一个子集来介绍 Agda 语言。读者即使不熟悉 Haskell 也不必担心，只需简单地了解一下语法就足以应付本书的要求了。

全文代码中 Unicode 部分使用宋体，ASCII 部分使用开启连字（Ligature）的 Fira Code（关键字开启粗体），其他文本使用 Cambria Math 和宋体。

全文 Haskell 代码需要 GADTs 和 UnicodeSyntax 两个扩展。即，在代码文件开头加上：

```
{-# LANGUAGE UnicodeSyntax, GADTs #-}
```

本书的创作目的是帮助理解函数式编程概念的读者进一步踏入形式验证领域，属于衔接书。类似的教程很喜欢大量使用未介绍的语法（点名批评*《七周七语言(卷 2)》*[2]的 Idris 部分），本书杜绝了这种情况。

本书使用的编程语言是 Agda，是因为它很接近另一门热门语言 Haskell，对普通的函数式编程爱好者来说比较容易上手。不过，**语言相关的知识并不是本书的重点，笔者希望读者通过本书了解的是定理证明，而不是 Agda**。类似的语言也不少，没有选择 F ★是因为它暂不支持**自动模式匹配**（Case-Split）和**自动证明**（Auto Proof-Search）。Idris 是另一个很好的选择，但是比 Agda 它也没有明显的优点，所以就选用了 Agda。

第一章介绍 GADT 和简单的 Agda 语法，第二章引入证明和相等性的概念并介绍一个使用等量代换的证明。到这里笔者都不希望读者进行实际的编程，而是到第三章才开始上手操作。第三章讲解环境搭建、语法糖、Emacs 的使用。请注意，本章不需要前置 Emacs 知识，读者也不需要写一行 Emacs Lisp 代码，请非 Emacs 用户千万不要紧张。第四章讲解数学归纳法，第五章讲解如何证明一个命题是假命题，第六章介绍了一套 DSL（Domain Specific Language，领域特定语言）并用它证明了加法交换律。

笔者有很多想写的内容限于篇幅和时间没有介绍，是一个很大的遗憾，希望以后有机会能弥补。这些特性包括但不限于：**`with`** 抽象（With-Abstraction）、**`rewrite`**、运行时证明、FFI、元编程、Dependent Record、Instance Argument、**`postulate`**、**`module`**、Copattern 与 Coinduction 等。

本书提供手册打印版和电子阅读版并优先考虑打印版的阅读体验。任何形式的阅读完全免费，在 CC BY-NC-SA 协议[3]

递等式 DSL

还记得小学数学的脱式计算吗

这是 DSL 的定义——混缀运算符形式的相等性的传递性和自反性（实现平凡，略）：

```
_≡⟨_⟩_ : ∀ {A : Set} (x : A) {y z : A}
       → x ≡ y → y ≡ z → x ≡ z
_QED : ∀ {A : Set} (x : A) → x ≡ x
infixr 2 _≡⟨_⟩_
infix 3 _QED
```

于是我们就可以用 `a ≡⟨ b ⟩ c ≡⟨ d ⟩ e QED` 证明命题 `a ≡ e`：原式（目标命题）左边 `a`，由类型为 `a ≡ c` 的**引理**（Lemma）`b` 转换为 `c`，再由类型为 `c ≡ e` 的引理 `d` 转换为原式右边 `e`。`n QED` 是 `refl` 对 `n` 的特化，在证明的最后一步使用“[原式右边] `QED`”比 `refl` 更可读。标准库中 `Relation.Binary.EqReasoning` 包有一份这样的封装。

比如，证伪 4 ⩽ 3：

```
4lt3 : 4 ⩽ 3 → ⊥
4lt3 (nltm (nltm (nltm ())))
```

可以通过不断对参数自动模式匹配来实现“自动证伪”。比如，可以先把 4lt3 写成：

```
4lt3 a = { }0
```

然后对 a 自动模式匹配四次就可以了。

为使类型更直观，标准库在 Relation.Nullary 包下定义了命题的否定符号：

```
infix 3 ¬_
¬_ : ∀ {a} → Set a → Set a
¬ P = P → ⊥
```

这个函数接收一个类型 P，返回“接收 P 的实例，不返回”这个函数类型。于是刚才的 4lt3 函数的类型就可以表达为：

```
¬ (4 ⩽ 3)
```

我们还可以进一步定义不等关系：

```
_≠_ : ∀ {a} {A : Set a} → A → A → Set a
x ≠ y = ¬ x ≡ y
```

Idris 中的谬模式写作 **impossible**。

本书使用 Microsoft Publisher 2016 这个极不专业的排版软件编写。

截止本书发布，最新的 Agda 版本是 2.5.4.1。本书没有使用标准库，所有代码已经放在 GitHub[4]。本书是第二个修订版，修复了很多之前版本中的问题，包括格式、翻译和代码几方面。最重要的一点是关于 Emacs 的——**很多人对使用陌生的编辑器抱有抵触情绪，但在这里是完全不必要的。** Emacs 可以使用上下左右键移动光标，可以通过键盘输入字母和数字，知道这些就够了。

如有发现错误、过时的内容，欢迎联系斧正。笔者联系方式： ice1000kotlin@foxmail.com

感谢编写过程中为我审稿的各位，尤其是指出大量排版、翻译、遣词造句方面问题及提出修改建议的欧林猫，对内容提出质疑和为我作序的 ICEY 和 Colliot，提出内容修改建议的魔理沙，指出其他问题的磷、Re、楼大海，以及第一版的各位读者。

敬请期待续集：《什么？是形式验证！》！

——2018 年秋，美国宾夕法尼亚州

[0]: http://ice1000.org/categories#agda
[1]: https://agda.readthedocs.io/en/v2.5.4.1/tools/emacs-mode.html
[2]: https://www.amazon.cn/dp/B01MTXAK6P
[3]: https://creativecommons.org/licenses/by-nc-sa/4.0/
[4]: https://github.com/ice1000/Books/

何谓 GADT

多出来的一点点灵活性

先看看我们只有代数数据类型的时候都能写些什么，比如一个 Maybe：

```
data Maybe a = Nothing | Just a
```

这定义了两个数据构造器（Data Constructor）——接收一个参数的 Just、不接收参数的 Nothing，还有接收一个类型参数的一个类型构造器（Type Constructor）——Maybe（其实还有两个**模式**（Pattern），随类型数据构造器一起出现）。把这三个构造器根据它们的类型写成函数，用 Haskell 表达就是：

```
Maybe :: * → *
Nothing :: Maybe a
Just :: a → Maybe a
```



```
ridiculous : ⊥ → ⊥
ridiculous a = a
```

这是一个虚无缥缈的函数，因为它无法被调用，因此也就可以返回无法返回的值。但是，同样的理论却不能用在这个函数上：

```
ridiculous' : 1 ≡ 0 → ⊥
ridiculous' a = { }0
```

目标 0 处不能填 a，因为 a 的类型不是⊥。

但逻辑上这明明是合理的——它同样无法被调用，为什么不能让它随意返回一个值呢？

Agda 为这种情况提供了特殊语法**谬模式**（Absurd Pattern），用于表达当前分支不存在。在目标 0 中对 a 自动模式匹配就能看到，Agda 直接把这个分支删了：

```
ridiculous' ()
```

这正是假命题应该有的样子，无法被调用的函数就不应该需要写实现。谬模式就是指使用一对空括号匹配不存在实例的类型，然后丢弃这个分支的实现。不需要实现的函数返回什么类型都可以，所以有（标准库 Data.Empty）：

```
⊥-elim : ∀ {a} {A : Set a} → ⊥ → A
⊥-elim ()
```

对此我们可以有一些推论，比如函数可以返回⊥类型当且仅当参数模式匹配的每个分支都能找到谬模式。证明函数无法实现也是一种实现，此乃反证法。而假命题的定义也就呼之欲出了，只要实现一个函数接收一个命题，返回⊥类型，就证伪了这个命题。

证伪

我可能在证一个假命题

如何用类型表达假命题呢？

既然程序是证明，写不出程序的类型就是假命题。“无实例的类型”的概念在很多语言中都存在——Rust 的`!`，Kotlin 和 Scala 的`Nothing`，Haskell 的`Void`。Agda 中也有一个这样的类型，读作**空类型**或者**底类型**（Bottom）写作⊥，在标准库`Data.Empty`包中，定义如下。

```
data ⊥ : Set where
```

由于⊥没有数据构造器，我们无论如何都无法获取它的实例。在实际的证明中经常遇到类似的情况，比如类型 1 ≡ 0，在唯一可能的数据构造器`refl`不适用的情况下就是一个没有实例的类型。

由于⊥没有实例，参数中有⊥的函数就无法被调用，返回类型为⊥的函数就无法返回。但是，这样的函数却存在：

可以看到作为数据构造器的后两者的返回类型都是`Maybe a`。GADT 和代数数据类型唯一不同的地方就是返回类型，我们不一定必须写`Maybe a`，而是可以把`a`替换为另外的类型。比如说我们可以有一个只能产生`Maybe Int`的数据构造器：

```
IntMaybe :: Int → Maybe Int
```

这个构造器多提供了一种生成类型为`Maybe Int`的值的方式（因此我们在模式匹配一个类型为`Maybe Int`的值时需要多处理一种情况）。

使用 Haskell 语法表达，代码是这样的：

```
data Maybe a where
  Nothing :: Maybe a
  Just :: a → Maybe a
  IntMaybe :: Int → Maybe Int
```

Agda 和 Haskell 语法相似，这里提供一个等价版本，读者可以先感受一下（类型的类型叫`Set`，和 Idris 语言里的`Type`一样）：

```
data Maybe (a : Set) : Set where
  Nothing : Maybe a
  Just : a → Maybe a
  IntMaybe : Int → Maybe Int
```

那么 GADT 提供的这个功能有什么好处呢？ Wikipedia 上有一个很好的例子——假设我们要定义一个语法树求值器，在没有 GADT 的时候只能定义出这样的语法树（为了简化，省略了 if else、eq 等结构）：

```
data Expr = ILit Int
          | BLit Bool
          | Add Expr Expr
```

我们会发现，这个定义会无法排除一些很明显的运行时错误，比如这样的表达式在类型上是合法的：

```
Add (ILit 233) (BLit False)
```

然而如果要实现一个 `eval` 函数对这个表达式进行求值，就需要对这种不合法的情况进行运行时报错，就必须返回一个 `Either` 或者 `Maybe` 值。而即使表达式是编译期已知且一定正确的，也需要处理一下这个返回的 `Either`，比如使用 `fromRight` 函数，这本质上就是断言一个异常的不存在。

而**一个编程语言在任何情况下需要程序员断言一个不可能存在的异常的不存在，都是这门编程语言类型系统不够强大的体现**。让我们来看看强大的 Haskell 是怎么处理这个错误的。定义 GADT：

```
data Expr a where
   ILit :: Int → Expr Int
   BLit :: Bool → Expr Bool
   Add :: Expr Int → Expr Int → Expr Int
```

这样就可以让上面的表达式在类型上变得不合法——因为 `BLit False` 的类型就从一个 `Expr` 变成了 `Expr Bool`，自然就不符合 `Add` 的签名里第二个参数的类型了。这就是控制数据构造器返回类型的妙用。

可能会有读者提出这样一个非 GADT 的 `Expr` 定义：

```
data Expr a = ILit Int
            | BLit Bool
            | Add (Expr Int) (Expr Int)
```


上下文（Context，也叫环境（Environment））功能可以获取这样的信息，快捷键 C-c C-e，即查看当前目标的上下文中所有非全局变量的类型。Emacs 会显示 5 条结果，最后两条显示 `ab` 的类型，是“变量 0 ⩽ 变量 2”，`bc` 的类型则是“变量 2 ⩽ 变量 1”，它们正好可以使用我们正在证明的传递性来推出我们需要的类型“变量 0 ⩽ 变量 1”！

我们可以把分类讨论的第一个情况看作数学归纳法的**起始步骤**（Base Case），第二个情况看作**递推步骤**（Induction Case），把：

```
abc ab bc
```

作为**递推假设**（Induction Hypothesis），来证明：

```
abc (nltm ab) (nltm bc)
```

这个过程用 Agda 代码表达就是：

```
abc (nltm ab) (nltm bc) = nltm (abc ab bc)
```

这就是**归纳证明**，在形式验证中无处不在。

Haskell 用户看到这段代码，一定感觉无比熟悉——很多 Haskell 函数都有着相似的结构，比如 `zip`：

```
zip :: [a] → [b] → [c]
zip [      ] bs = []
zip (a : as) (b : bs) = (a, b) : zip as bs
```

对于相等关系，可以通过 `cong` 函数来实现归纳证明。比如，对于自然数 `n`、`m`，假设有命题 `n ≡ m` 的证明 `p`，那么可以得到 `suc n ≡ suc m` 的证明 `cong suc p`。对于自然数是如此，对于其他可归纳的结构亦是如此。

我们还可以证明不等关系的传递性，这在交互式证明的帮助下非常简单。先写下命题和参数：

```
abc : ∀ {a b c} → a ⩽ b → b ⩽ c → a ⩽ c
abc ab bc = ?
```

由于 a ⩽ b 被模式匹配为 0ltn 时 a 一定为 0，此时结论 a ⩽ c 直接成立，因此我们先对 ab 进行自动模式匹配，Agda 会生成分别对应两个数据构造器的分支：

```
abc 0ltn bc = { }0
abc (nltm ab) bc = { }1
```

这是我们第一次见到有多个分支的证明，但只要分别对这些分支进行证明就好——此乃分类讨论。

目标 0 直接成立，因此直接 C-c C-a 或者手动填入 0ltn 后 C-c C-SPC，证毕。

考虑目标 1，尝试直接自动证明，失败。尝试模式匹配 bc，获得一个新分支（没有生成两个分支的原因会在下一册说明）。这个分支其实是可以自动证明的，但为了知其所以然，我们应该分析一下为什么应该如此证明。根据目标区，目标 1 的类型是“suc 变量 0 ⩽ suc 变量 1”（原变量名是编译器生成的，书中改用更可读的命名），正好符合 nltm 的返回类型。这时我们可以利用“完成目标”功能，在目标 1 中填入“nltm ?”，然后 C-c C-SPC 生成目标 2，类型是“变量 0 ⩽ 变量 1”，正合我们所意。

先 C-c C-l 重新加载一下文件，所有目标被重新按顺序编号，目标 2 变成目标 0，代码如下：

```
abc (nltm ab) (nltm bc) = nltm { }0
```

要利用变量 0 和变量 1，首先应该知道它们到底是什么。

这个定义也不能满足我们的需求（即使它看起来似乎很美好），因为 BLit False 的类型在这个定义下依然是 Expr a 而不是 Expr Bool。

本章是*《什么是 Haskell 中的 GADT（广义代数数据类型）？》*[0]的重写版。之所以重写是因为原文篇幅较长、用语相对非正式而且有一些不必要的代码，在重写中都修正了。

Agda 语言所有的代数数据类型都使用 GADT 语法定义，抛弃了 Haskell 的代数数据类型语法。当然，Agda 的 GADT 还有很多值得讨论的特性，后文会提及。

[0]: https://colliot.me/zh/2017/11/what-is-gadt-in-haskell/

何谓证明

天平、炼金术与等量代换

要理解命题的证明，首先需要理解如何表达命题。我们已经知道，这是要用类型表达命题，用类型对应的表达式的存在性来作为这个命题的正确性的证明。

在命题逻辑中，我们有变量（Variable）（即原子命题）和一些逻辑运算符（合取（Conjunctive，∧），析取（Disjunctive，∨），蕴涵（Implication，→））。

变量以类型表达时一般是一个值的类型，用字母表示。而蕴涵关系 p → q 对应的就是“函数”这一概念，它组合了两个类型 p 和 q。但是这是为什么呢？

假如有函数 f 的类型是 p → q，并且它有不抛异常、永远停机的实现，那么只要有一个命题 p 的证明，就能通过这个蕴涵关系 p → q 得到一个 q 的证明。而如果我们有类型为 a → b → c（蕴涵运算符右结合，柯里化）的函数 g，那么 g

归纳证明

对所有的偏序关系都有效哦

相等性作为一种性质，在代码里以 GADT 形式表达。其他性质也可以表达为 GADT，比如我们可以把 ⩽ 关系定义成“0 ⩽ 任意自然数”和“当 n ⩽ m，有 suc n ⩽ suc m”两条规则的组合，于是有 GADT：

```
data _⩽_ : (a b : Nat) → Set where
  0ltn : ∀ {n}   → 0 ⩽ n
  nltm : ∀ {n m} → n ⩽ m → suc n ⩽ suc m
```

于是，2 ⩽ 3 就可以表达为 `nltm (nltm 0ltn)`了。我们可以用“自动证明”功能随意证明一些常量之间的 ⩽ 关系，比如 7 ⩽ 13（证明结果略，太长了）：

```
7lt13 : 7 ⩽ 13
7lt13 = ?
```

完成目标（Fill Goal），快捷键 C-c C-SPC，用于对手动填入目标的表达式（可以是带有新目标的表达式）进行类型检查。如检查通过，就用这个表达式替换目标并更新目标区，否则在目标区显示错误信息。这个快捷键将在下一章大量使用。

精化（Refine），快捷键 C-c C-r，用途是自动填入一个带有新目标的表达式到当前目标中。

跳转到定义，光标在一个标识符上时即可使用，快捷键 M-.。跳转到定义后，Emacs 25.1 及以上版本可以使用快捷键 M-,**跳回去**，其他版本则是 M-*。

Emacs 光标前后移动分别是 C-f 和 C-b，因此在它们前面分别加上 C-c，就有了**跳转到下一个目标** C-c C-f 和**跳转到上一个目标** C-c C-b。

其他特殊符号的输入可以参考官方文档[1]。

[0]: https://tio.run/
[1]: https://agda.readthedocs.io/en/v2.5.4.1/tools/emacs-mode.html

的实现就是“如果 a 成立，那么‘如果 b 成立，那么 c 成立’这一命题成立”。再考虑其他逻辑运算符，a ∨ b 只需要 a 或 b 中的一个成立，对应 Haskell 的 `Either` 类型。a ∧ b 需要 a 和 b 同时成立，对应元组（Tuple）。

这个命题读起来比较拗口，我们可以使用 Haskell `Prelude` 里的函数 `uncurry` 来将它变得顺口一些：

```
uncurry :: (a → b → c) → ((a, b) → c)
```

`uncurry g` 的类型就变成了“如果 a ∧ b 成立，那么 c 成立”。在 Agda 中，`g` 写作：

```
g : {a b c : Set} → a → b → c
```

这里用到的语法和 Haskell 只有三点区别：类型声明必须写出来、同类型的变量可以合并以及 `::` 变成了 `:`。Agda 可以自动传入可推导的参数，以**大括号**标记，名曰**隐式参数**，无括号或者小括号里的叫显式参数。

对于类型中连续出现的同类型、同显隐式的变量，如

```
(a : A) → (b : A) → (c : A)
```

可以把它们的定义放到一起，变成：

```
(a b c : A)
```

Haskell 和 Idris 会自动声明类型变量，但由于 Agda 的类型签名里除了有类型还可以有值、类型的类型，这带来了很强的不确定性，所以干脆就不自动声明类型变量了。

有的函数返回类型受前面参数的影响，所以需要在返回类型签名里直接使用参数，为此我们使用`(a : A)`这样的语法把参数引入到类型中。这种函数被称为**依赖函数**（Dependent Function）。

Agda 的“类型”这一概念比 Haskell 的复杂很多。在 Agda 中，`Set` 类型是一个特殊的类型——它其实有一个参数，类型是 `Level`（定义于 `Agda.Primitive` 包下，在标准

库的 Level 包中二次封装），原定义较为繁琐，这里提供一个语义等价的版本：

```
data Level : Set where
  lzero : Level
  lsuc  : Level → Level
```

在没有直接传递这个参数的情况下编译器会自动将一个 lzero 应用到 Set，可以算是一种语法糖。否则就至少需要 **import** 一个库才能定义类型了。

Level 类型提供了一个空构造器和一个单参数构造器，这两个构造器分别代表一个叫 0 的单例 Level 和 Level 的后继操作。用这种方法可以表达所有自然数，因为任何一个自然数 n 都可以通过 0 进行 n 次后继操作得到。Agda 将一个类似的定义作为自然数放在 Agda.Builtin.Nat 包下，标准库于 Data.Nat 包中二次封装。这种自然数被称为**皮亚诺数**（Peano Numbers）。Level 可能是语法糖的缘故，被做成了单独的类型并提供了特殊的定义。

Set lzero 表示值的类型，也就是 Haskell 大部分时间讨论的类型； Set (lsuc lzero)表示类型的类型，在 Haskell 中写作*读作 Kind ； Set (lsuc (lsuc lzero)) 表示类型的类型的类型，Haskell 中没有这个东西；以此类推。有了 Level，我们就有了高阶逻辑，类型的类型无穷匮也。

之所以 Agda 需要把 Level 做成 Set 的参数，是为了在更高的层次上泛化函数的抽象能力。回忆一下 Haskell 中的 flip 函数——我们可以写出等价的 Agda 代码：

```
flip : {a b c : Set lzero}
       → (a → b → c) → (b → a → c)
```


先尝试 C-c C-a，也就是直接自动证明，编译器会在目标区说找不到证明，和预期一致。于是我们考虑对参数 ab 进行模式匹配：先在目标里输入 ab（也就是需要被模式匹配的变量），然后 C-c C-c，对它进行自动模式匹配，于是 ab 就被替换成了 refl，但这时 b 也被替换成了.a。后者是一种叫点模式的特殊模式。

之所以需要点模式，是因为在有的情况下，一个模式的匹配结果会对其他模式进行限制，比如当 a ≡ b 被匹配为 refl 时，b 和 a 就可以互相代换了，上下文中所有 b 就可以被换成 a 来简化类型（默认策略是把等号右边换成左边），于是匹配 b 的模式就可以被写成 a。但这个模式用到了另一个模式匹配出来的变量，所以需要在模式前面加一个点来标记这不是重名变量。

现在的代码应该是这样的（{ }0 的背景应该是绿色）：

```
trans' a .a c refl bc = { }0
```

再尝试自动证明，会发现 bc 被自动填入了目标，这也意味着证明完成了。这是因为，当上下文中的 b 都被换成 a 后，bc 的类型 b ≡ c 中的 b 也被换成了 a，也就是说 bc 的类型变成了 a ≡ c，正好就是目标的类型。

通过自动证明，我们发现了之前证明过的命题的另一种证法。边阅读目标区的类型边证明的过程就是交互式证明的过程，这可以同时发挥程序精确的优点和人类思路清晰的优点。

读者可以改变证明流程（比如，先对 bc 进行模式匹配，可以看到目标区类型的变化。再比如，把 ab 和 bc 都匹配出来，可以看到目标区变成 a ≡ a，也就是我们最初的做法）来进一步体验整个过程。

```
trivialEq' : ∀ {a} {A : Set a} → A ≡ A
trivialEq' = refl
```

如果这个参数不是隐式参数，那么还可以省略括号：

```
trivialEq'' : ∀ a b → a + b ≡ a + b
trivialEq'' a b = refl
```

Agda 的 Emacs 插件还有两个很棒的附加功能：自动模式匹配和自动证明，分别对应快捷键 C-c C-c 和 C-c C-a。比如上一章中 `trans` 的例子，其实就可以使用 Agda 的自动证明和自动模式匹配功能完成。首先把类型和参数都写出来，并使用问号替代未完成代码：

```
trans' : ∀ {a} {A : Set a} (a b c : A)
     → a ≡ b → b ≡ c → a ≡ c
trans' a b c ab bc = ?
```

然后使用 C-c C-l 加载文件，可以发现 Emacs 把问号变成了一个绿色背景的`{ }0`，表示这有一个编号为 0 的**目标**（Goal），目标即一个未完成的程序。

未完成的程序也就是未完成的证明，Agda 可以辅助我们完成证明。证明的过程我们不断和 Agda 编译器交互，因此这一过程被称为**交互式证明**。理想情况下应该由 Agda 独立完成整个证明过程，但自动证明实在过于复杂，因此需要人工辅助。在这个例子中，之所以需要把 `a b c` 写成显式参数是为了让 Agda 更好地在目标区显示证明的状态。

证明时，需要把光标放在目标里。目标区会显示目标的编号和类型，此时可以看到第 0 号目标的类型：

```
?0 : a ≡ c
```


再次提醒大括号表示隐式参数。而考虑类型构造器 `Either`，它的定义可以看作：

```
Either : Set → Set → Set
```

我们就不能对这个 `Either` 使用 `flip` 把它的参数反过来，因为参数 `Set` 的类型是 `Set (lsuc lzero)`。而如果把这个 `Level` 参数化，就可以做到：

```
flip' : {l : Level} → {a b c : Set l}
      → (a → b → c) → (b → a → c)
```

没有 `Level` 概念的 Haskell 就做不到这种抽象了。

同一个函数接口处理不同类型参数的能力被称为**多态性**（Polymorphism），而像这种对于所有 `Level` 的类型都有处理能力的多态性被称为**通用多态**（Universe Polymorphism），把类型作为参数传入的多态叫**参数化多态**（Parametric Polymorphism）。这些性质并不是互斥的。

Haskell 支持通过 **`infix`** 系列关键字自定义中缀（Infix）运算符以及它们的优先级和结合性（如 `<*>`，`>>=`，`<|>`，`^>>`），是 Haskell 语法灵活的体现。

Agda 则进一步支持了**混缀**（Mixfix）运算符并保留了指定优先级结合性的能力，使用和 Haskell 相同的一套关键字（**`infix`**，**`infixl`**，**`infixr`**）。我们可以在定义运算符时使用下划线作为“参数占位符”（但不能有两个连续的）。举个例子，Haskell 直接调用中缀运算符的语法 `( <+> ) a b` 在 Agda 中对应 `_<+>_ a b`。

而 Agda 可以定义 `if_then_else_`这样的运算符，然后以 `if a then b else c` 的语法调用，其中 `a b c` 是参数；以及前缀运算符 `a_`和后缀运算符`_b`。也就是说 Agda 的运算符支持完全是 Haskell 的超集。

我们来看一个更复杂的例子：这是用来表达相等性（严格来说叫**直觉等价**（Propositional Equality））的 GADT，它使用了刚才介绍的通用多态和中缀语法：

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

可见类型签名中相邻括号之间的箭头可以省略。

它定义于内置的 Agda.Builtin.Equality 中，标准库于 Relation.Binary.PropositionalEquality 包中二次封装。这个类型签名有些复杂，我们来仔细研究一下它的含义。

之前的 GADT 定义都属于类型构造器的定义，也就是 **data** ［名字］［类型参数］： Set 这样的结构。 但这个定义并不是一个 Set 或者 Set a，而是一个类型层面的函数 A → Set a，意味着这个类型构造器构造出来的是一个函数而不是一个类型。而它本身还有一个类型为 A 的参数。这意味着这个类型需要首先通过一个类型 A 的实例构造出来（这是第一个参数，它被命名为 x），这时如果再对它应用一个类型 A 的实例（第二个参数，没有命名），就可以得到一个可以被视为表达相等性的命题的类型了。

前文说过依赖函数可以让函数的返回类型声明中直接使用参数的值，在这里我们向 GADT 中引入类似的能力——在数据构造器的类型声明中使用 GADT 的参数。在_≡_的定义中可以看到 refl 这个单例构造器使用了参数 x，它来自类型构造器的第一个参数。这并不是一个复杂特性，它在 Haskell 中甚至是隐式的。

现在这个定义中所有语法都已知了，但我们还是不知道

包引入了_≡_，第二个引入了自然数相关的定义，比如 Nat 这个 GADT 和 suc、zero 两个数据构造器，相关函数+，-，*（分别是自然数的加、减和乘）等。这两个包里的内容足够我们做一些简单的证明了。

既然要真的开始编程，那么我们有必要了解一些语法糖了。如果因为任何原因（比如需要不在作用域里的变量）想让 Agda 自己推导一个表达式的值，在这个表达式的合法值只有一种可能性的情况下（否则可能导致推出不想要的值），可以使用**下划线**代替，表示让 Agda 自行推导。比如我们想表达命题 1+1 ≡ 2，而皮亚诺数 2 的表示太长，就可以省略（按“\==”可以输入≡）：

```
theorem : suc zero + suc zero ≡ _
theorem = refl
```

顺带一提，Agda 支持使用数值字面量表达对应的皮亚诺数，比如上面的 theorem 可以写作：

```
theorem' : 1 + 1 ≡ 2
theorem' = refl
```

数值字面量的映射规则可以自定义，篇幅所限，略过不提。我们使用标准库的定义就好了。

再比如我们没有 **import** Agda.Builtin.Level 但是需要通用多态，使用下划线代替就可以了（按“\r”或“\to”或“\->”可以输入→）：

```
trivialEq : {a : _} {A : Set a} → A ≡ A
trivialEq = refl
```

类型签名中可以用∀（按“\all”输入）简化：

Emacs 模式带来了一种全新的编程方式，它大大减少了编程者的工作量，而这个过程中 Emacs 本身只充当了一个支持显示彩色字符的记事本的作用。本章介绍的这种编程方式对普通的 Emacs 用户来说也是陌生的。

Agda 安装库的过程较为繁琐，考虑到本书并不依赖标准库就略过这部分了。安装完成后可以这样测试：

```
agda --version
```

如果输出了版本号，说明安装正常。

现在用 emacs 打开一个扩展名为 agda 的文件（比如 Hello.agda），可以看到 mode-line 已经显示这是一个 Agda 文件了。然后我们先输入一些代码：

```
module Hello where
```

这个 **module** 的名字必须和文件名匹配。然后使用快捷键 C-c C-l 来让 Agda 编译器加载这个文件并显示高亮和报错。Emacs 最下方会出现一个小窗口（方便起见，称为**目标区**），里面什么也没有。目标区用于显示编译器提供的信息。高亮出现代表基本的代码检查通过了，每次刷新高亮都需要按这个快捷键。

如果代码有语法错误，那么出错的地方会有红色高亮，其他地方是黑色的。如果代码在**全面性**（Exhaustiveness，即模式匹配没有处理所有情况）、**停机性**（Termination，即可能无限递归）上有错误，那么出错的地方会有黄色背景，其他地方正常高亮。这两种情况下，目标区会显示错误信息。

导入之前介绍的几个 GADT：

```
open import Agda.Builtin.Equality
open import Agda.Builtin.Nat
```

open import 和 Haskell 的 **import** 语义相同。第一个

它的用法。我们可以写一些简单的类型表达式来理解_≡_和 refl 的用法（方便起见，用了 Level）：

```
levelEq : lzero ≡ lzero
```

不仅有 0 ≡ 0，还可以有 1 ≡ 1：

```
levelEq' : lsuc lzero ≡ lsuc lzero
```

是不是有些眉目了？先不考虑实现，这两个函数分别是 0 ≡ 0 和 1 ≡ 1 这两个命题写成类型的形式。

而由于这两个命题是平凡正确的，所以不需要任何步骤就能“证明”他们。根据 DRY（Don't Repeat Yourself，不写重复代码）原则，我们应该会有把所有代表“自己等于自己”的类型提取成一个函数。先把类型写出来：

```
trivial : {a : Level} {A : Set a} → A ≡ A
```

有没有发现这其实和 refl 的类型完全一样？

也就是说，**refl 其实就是“自己等于自己”这一命题**，它的全名是**自反性**（Reflexivity）。而刚才提到的函数 levelEq，levelEq'都属于对某个类型特化的 refl。

然而这些命题都属于**原子命题**。前面说过，蕴涵关系对应函数抽象，那么组合原子命题，就可以得到更有趣的命题。

比如，相等性的**传递性**（Transitivity）——如果命题 a ≡ b 和 b ≡ c 成立，那么 a ≡ c 成立。如果使用函数来表达这个命题，就是（问号表示代码未完成）：

```
trans : {l : Level} {Q : Set l} {a b c : Q}
      → a ≡ b → b ≡ c → a ≡ c
trans ab bc = ?
```

接下来怎么办？如果用 refl 代替问号，Agda 会给出类型错误。似乎我们需要把 ab 和 bc 两个变量利用起来。

其实，只需要把这两个变量模式匹配出来就好了：

```
trans refl refl = refl
```

这个实现可以通过类型检查。因为 Agda 有个推理规则——如果在一个上下文中（此处指问号处）一个类型为 `a ≡ b` 的变量被模式匹配为 `refl`，那么 `a` 和 `b` 在这个上下文就被视为可互相代换的，此乃 **J 规则**（J-Rule），在大多数定理证明器（Proof Assistant，例如 Idris，F ★）中也适用。

读者有兴趣的话可以看看 J 规则的 Agda 代码定义，使用了下一章要介绍的语法**点模式**（Dot Pattern）：

```
J : {A : Set} (P : (x y : A) → x ≡ y → Set)
  → ((x : A) → P x x refl)
  → (x y : A) (xy : x ≡ y) → P x y xy
J P p x .x refl = p x
```

J 规则可证明[0]相等性的**一致关系**（Congruence）（标准库中定义在_≡_的包里），在最后一章被多次使用：

```
cong : {A B : Set} (f : A → B)
     → {m n : A} → m ≡ n → f m ≡ f n
cong f refl = refl
```

相等性还满足**对称性**（Symmetry）：

```
sym : {A : Set} {m n : A} → m ≡ n → n ≡ m
sym refl = refl
```

[0]: https://stackoverflow.com/a/27100693/7083401

交互式证明

梦回80年代，聆听编译器的想法

Agda 目前唯二推荐的编辑器是 Atom 和 GNU/Emacs，本书仅介绍后者。临时运行代码可以选择网站 TIO[0]。推荐的安装方式是编译源码，只需两步（编译时间长，请耐心等待），Stack 用户把 `cabal` 换成 `stack` 即可：

```
cabal install alex happy cpphs
cabal install Agda
```

然后配置 Emacs 插件（需要提前安装 Emacs）：

```
agda-mode setup
agda-mode compile
```

本章不会涉及一行 Emacs Lisp 代码，不会涉及 Emacs 的高级功能，不涉及对. emacs 文件的操作。Agda 通过它的