計算機輔助證明介紹及簡單 Haskell 實現

Ruozeli

June 10, 2026

1 理論基礎

計算機輔助證明的核心原理是 Curry–Howard 同構, 該同構在構造性證明與計算機程序之間建立起聯系, 從而可以把數學命題視作類型, 證明視作程序. 據此, 程序語言的類型檢查器 (當然, 前提是該語言有足夠強大的類型系統) 便可充當計算機證明助手, 我們可以通過檢查程序中變量, 函數參數和返回值的類型是否匹配來檢驗證明是否正確. 為理解該過程, 我們首先須理解直覺主義意義下的 “構造性證明”.

1.1 直覺主義邏輯和 BHK 釋義

先來考慮這樣一個問題: 什麼是一個 “可證的命題”? 這看似是一個多余的問題, 但要認真回答卻並不那麼容易.

一般而言, 我們平常所見的數學證明依據的幾乎都是古典二值邏輯, 每一個合法的數學陳述 “先驗地” 具有一個確定的真值 (0 或 1), 一旦我們證明某個陳述的否定為假, 就立即能斷言該陳述自身為真. 用命題邏輯來表達, 即 $\vdash \varphi \vee \neg\varphi$, 而不論公式 φ 的具體含義, 這也就是古典邏輯中的排中律 (*tertium non datur*).

排中律使我們能夠在不給出具體構造的情況下斷言某個數學對象的存在性, 為此只需表明該對象不存在會導致矛盾即可. 像這樣基於排中律和反證法, 不給出實際構造的數學證明被稱為**存在性證明**. 在通常觀點下, 這樣的證明當然是被承認為真的. 然而, 存在性證明包含的信息相當有限, 我們不知道一個對象的具體形象而僅僅斷言其存在, 這在很大程度上是一個空洞的真陳述. 與之相對, **構造性證明**則會明確地給出該對象的構造方法, 這種證明在應用場景中顯然會更受青睞.

20世紀上半葉, 以 Brouwer 為代表的直覺主義者拒絕承認存在性證明, 要求數學證明必須給出具體構造, 與 Hilbert 領導的形式主義陣營針鋒相對. 這種對立背後有哲學層面的緣由: Hilbert 在哲學上傾向於傳統的柏拉圖主義觀點, 該觀點認為數學對象獨立於人類心靈

而客觀存在, 所以某個數學對象存在與否是預先斷定的, 故可以為命題賦予 “絕對” 的真值, 排中律的使用在這一背景下便名正言順; 而形式主義的方法論在對象指稱問題上不做任何擔保, 所以 Hilbert 建立的演繹系統就自然采用了符合數學傳統的推演形式. 而直覺主義者認為數學對象本質上只是人類心靈的構造物, 在沒有具體構造之前, 我們不能僅憑邏輯准則空談存在. 在此, “指稱” 問題被重新強調: 在尚未完成構造時, 我們無法有意義地談論該對象.[1]

進而, 兩派數學家在技術層面也對數學證明做出了不同解讀. 古典命題邏輯允許我們基於 Boole 代數建立命題的語義, 根據可靠性和完全性定理, 檢驗一個命題是否為系統內定理, 相當於從語義層面計算真值表的最後一列是否都為 1. 而在直覺主義命題邏輯中, 由於排中律不被承認為公理, 系統中的每一個命題不再具有確定的 “邏輯真” 或 “邏輯假”, 所以我們無法寫出僅包含 0 和 1 的二值邏輯真值表. 既然不能依靠真值表, 直覺主義命題邏輯就必須找到別的依據來判定語義, 這種依據就是**構建模式**. 此即著名的 **BHK 釋義** (Brouwer-Heyting-Kolmogorov): 一個命題為真, 僅當我們能夠從前提 “構造” 出它的結論.

* * *

具體來說, 給定命題變元集 PV, 定義全體公式集 Φ 為滿足如下條件的最小集合:

1. 每個命題變元和 $\bot$ (謬) 都在 Φ 中,
2. 若 $\varphi, \psi \in \Phi$, 則 $(\varphi \to \psi)$, $(\varphi \vee \psi)$, $(\varphi \wedge \psi) \in \Phi$.

其中, $\to$ (蘊涵), $\vee$ (析取) 和 $\wedge$ (合取) 是基本命題聯結詞, 而 $\neg$ (非), $\leftrightarrow$ (等價) 和常量 $\top$ (真) 都是縮略式:

1. $\neg\varphi \equiv_{df} \varphi \to \bot$,
2. $\varphi \leftrightarrow \psi \equiv_{df} (\varphi \to \psi) \wedge (\psi \to \varphi)$,
3. $\top \equiv_{df} \bot \to \bot$.

下面給出直覺主義構造規則[7]:

1. $\varphi_1 \wedge \varphi_2$ 的一個構造由 φ_1 的一個構造和 φ_2 的一個構造構成,
2. $\varphi_1 \vee \varphi_2$ 的一個構造由一個指標 $i \in \{1, 2\}$ 和 φ_i 的一個構造構成,
3. $\varphi_1 \to \varphi_2$ 的一個構造是一個將每個 φ_1 的構造轉換為 φ_2 的構造的方法 (函數),

[1] 據說, Brouwer 曾受 Kant 范疇表中對判斷的三分法 (“肯定的, 否定的, 無限的”) 之啟發, 即對於無限大的問題域, 我們的理智不能對其進行恰當的劃分 (斷言真或假), 故在無確定根據時只能懸擱判斷.

4. 不存在 $\bot$ 的構造.

這幾條規則是符合直觀的, 如果我們需要一個 $\varphi_1 \wedge \varphi_2$ 的證明, 就必須有 φ_1 的證明和 φ_2 的證明; 如果需要一個 $\varphi_1 \vee \varphi_2$ 的證明, 則只需 φ_1 或 φ_2 的證明兩者之一即可; 如果能構造一個從前提 φ_1 到 φ_2 的證明 (即證明 $\varphi_1 \to \varphi_2$), 那就相當於是把 φ_1 的構造轉換為了 φ_2 的構造; 最後, $\bot$ 是抽象意義的邏輯假, 它一定是不可構造的.

* * *

至此, 我們僅僅是抽象地說明了 "構造" 的含義, 儘管這尚不足以得到一條真正的系統內定理, 但這一部分對後文的討論至關重要.

事實上, 直覺主義命題邏輯與古典命題邏輯的語法幾乎相同, 有一條推理規則和兩條公理[2]:

1. MP: $\{\varphi, \varphi \to \psi\} \vdash \psi$ (分離規則),
2. Ax1: $\varphi \to (\psi \to \varphi)$ (肯定後件律),
3. Ax2: $(\varphi \to (\psi \to \rho)) \to ((\varphi \to \psi) \to (\varphi \to \rho))$ (蘊涵詞分配律).

由於 $\neg$ 不是基本命題聯結詞, 直覺主義命題邏輯中沒有否定公理 (換位律). 基於此, 我們可以得到直覺主義不承認排中律的另一重理由: 命題 $\varphi \vee \neg\varphi$ 在直覺主義命題邏輯中應寫作 $\varphi \vee (\varphi \to \bot)$, 很容易看出該命題不一定是可構造的. 同樣地, 雙重否定消去律 $\neg\neg\varphi \to \varphi$ 在直覺主義命題邏輯中無效, 這意味著我們不能使用反證律 (但雙重否定引入律 $\varphi \to \neg\neg\varphi$ 仍有效, 故可以用歸謬律). 除此之外, 許多古典命題邏輯的重言式 (即語義恆真的公式) 在直覺主義命題邏輯中都無效.

然而, 在直覺主義命題邏輯中, 公式 $\neg\neg\neg\varphi \to \neg\varphi$ 仍然有效, 因為我們有如下推演:

1. $\{\varphi, \neg\neg\varphi \to \bot\} \vdash \neg\neg\varphi$ (雙重否定引入律),
2. $\{\varphi, \neg\neg\varphi \to \bot\} \vdash \bot$ (MP),
3. $\{\neg\neg\varphi \to \bot\} \vdash \varphi \to \bot$ (演繹定理),
4. $\vdash \neg\neg\neg\varphi \to \neg\varphi$ (演繹定理).

此外, 爆炸原理, 即 $\bot \to \varphi$ 在直覺主義命題邏輯中仍有效. 對此可以這樣理解: $\bot \to \varphi$ 存在一個構造僅當我們有一個方法能把 $\bot$ 的每個構造轉換為 φ 的構造, 然而事實上並不存

[2]我們可以僅根據公理和推理規則進行推演, 則每個證明都被理解為一個從前提集出發, 終止於所證命題的有限公式序列. 但這種寫法 (即 Hilbert 式演繹) 過於繁瑣, 下文將采用等價的 Gentzen 式自然演繹做進一步討論.

在 $\perp$ 的構造, 所以該陳述空洞地為真 (vacuously true), 相當於是一個無輸入的函數, 其輸出無論是什麼都將是合法的. 更進一步說, 如果系統內容納了一個矛盾, 那麼由爆炸原理, 該系統就會變成完全平凡的, 這也正是指示系統出現問題的一個重要標志, 這一點無論對古典邏輯還是直覺主義邏輯來說都是關鍵的. 一個推論是: $\top\ (= \perp \to \perp)$ 一定是可構造的, 因此 $\top$ 代表了抽象意義的邏輯真.

1.2 λ 演算

為了將證明與類型檢測聯系起來, 下面先引入 Church 的 λ 演算. 在計算機程序語言中, λ 演算一般指匿名函數, 而計算理論中的 λ 演算含義與之略有不同.

給定一個字符集 $\Sigma = \{x, y, z, \cdots\}$, 則任何一個合法的 λ-表達式都具有如下三種形式[3]:

1. 變量, 如 x,
2. 抽象 (abstraction): $\lambda x.E$, 其中 E 為函數體, x 為參數, 一個函數有且僅有一個參數. $\lambda x.E$ 即一般語境下的 $x \mapsto E$, "λ" 和 "." 之間的變量 x "捕獲" 函數體 E 中出現的 x, 將其與輸入參數綁定起來.
3. 應用 (application): $E_1\, E_2$, 其中 E_1, E_2 都是合法的 λ-表達式 (注意 E_1, E_2 之間有一個空格).

函數書寫的一般慣例為: 抽象時, 函數體盡可能向右延展, 應用時, 函數從左向右結合. 例如, $\lambda x.x\ \lambda y.x\, y\, z$ 應理解為 $\lambda x.(x\ \lambda y.((x\, y)\, z))$.

λ 演算有三條求值規則, 分別為:

1. α-重命名: 可任意更改變量名, 例如 $\lambda x.x$ 可改為 $\lambda y.y$,
2. β-歸約: 相當於代入化簡, 用傳入參數替換函數體內被綁定到該參數的變量名. 以 $(\lambda x.x)\ (\lambda y.y)$ 為例, 我們用第二個表達式替換第一個表達式中的 x, 並在替換後刪去用於綁定變量的 "$\lambda x.$", 便得到 $\lambda y.y$.
3. η-等價: 形如 $\lambda x.f\, x$ 的表達式可改為 f. 事實上, 我們對任何參數 y 應用表達式 $\lambda x.f\, x$, 經過 β-歸約後所得結果都為 $f\, y$, 由此表明等價性.

雖然函數抽象只能接受一個參數, 但 λ-表達式可以通過 Curry 化 (Currying) 接受多個參數. 例如, 函數 $f(x, y) = x + y$ 可以表示為 $f = \lambda x.(\lambda y.x + y)$, 如果我們輸入 $f\, 1\, 2$, 則參數 1 和 2 先後與函數體中的 x 和 y 綁定, 得到結果為 3. 如果只輸入 $f\, 1$, 則參數 1 與 x 綁定, 將返回一個可用的函數 $\lambda y.y + 1$, 這就是多參數函數的 Curry 化. 之所以規定函數只能

接受一個參數, 是因為 λ 演算系統內的所有值本質上都是函數, 如果不這樣規定, “無論填入多少個參數都無法得到一個 ‘最終’ 的結果”[3]. [3]

* * *

定義三個組合子 S, K, I 如下:

1. $I = \lambda x.x$ (即恆同函數),
2. $K = \lambda x.\lambda y.x$ (輸入兩個參數, 僅返回第一個),
3. $S = \lambda x.\lambda y.\lambda z.x\,z\,(y\,z)$ (輸入三個參數 f, g, h, 先將 f 應用於 h, 再將 g 應用於 h, 並把 $g \circ h$ 作為參數輸入給 $f \circ h$. 換句話說, 組合子 S 給出了如下映射: $f \circ g \circ h \mapsto (f \circ h) \circ (g \circ h)$ 的 λ 演算表達)

這三個組合子構成了 λ 演算的一個 “完全組”, 一般地, 任何 λ-表達式都可表為它們的組合, 因為:

1. $\lambda x.x$ 可寫為 I,
2. $\lambda x.A$ 可寫為 $K\,A$, 其中 A 不含變量 x,
3. $\lambda x.A\,B$ 可寫為 $S\,(\lambda x.A)\,(\lambda x.B)$.

很容易看出 K 和 S 組合子與直覺主義構造模式的類似性. 事實上, 這兩個組合子分別與直覺主義命題邏輯的 Ax1 和 Ax2 相對應. 但在具體討論之前, 必須先定義函數的類型.

* * *

考慮這樣一個 λ-表達式

$$\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x)$$

試著對其進行化簡, 把第二個表達式代入第一個, 則會得到

$$\Omega = (\lambda x.x\,x)\,(\lambda x.x\,x) = \Omega$$

可見這是一個無限循環的項, 故無類型的 λ 演算不能保證自身可停機. 該項的構造類似於 Russell 悖論, 正如公理集論通過引入公理排除 Russell 悖論, λ 演算也應通過進一步規范化排除這種不停機的項. 為此, Church 為 λ 演算引入了類型系統. 這種做法與 Russell 在 *Principia Mathematica* 中提出的類型論方案相似.

在簡單類型 λ 演算 $(\lambda \to)$ 中, 每個項都必須有一個類型, 記作 $x : A$, 意即變量 x 的類型為 A. 例如, 在無類型 λ 演算中, 一個表達式寫作 $f = \lambda x.\lambda y.x$, 給它加上類型約束後就變為

[3] 函數是 λ 演算的 “一等公民”, 在函數式編程中也是如此.

$f = \lambda x : \mathrm{Int}.\, \lambda y : \mathrm{Int}.\, x$, 此時, f 只能接受整數型的輸入. 進而我們可以定義函數的類型: 若一個函數 f 接受 A 類型的參數並輸出 B 類型的結果, 則定義 f 的類型為 $A \to B$, 這與數學中的函數定義 (定義域 $\to$ 值域) 是一個道理. 而對於有 n 個參數的函數 g, 它的類型可以寫成

$$A_1 \to (A_2 \to (A_3 \to \cdots (A_{n-1} \to (A_n \to B)) \cdots))$$

這種寫法是由於多參數函數的 Curry 化, 一個 n 元函數在輸入一個參數後便返回一個 $n-1$ 元函數, 以此類推, 最終約化為一個一元函數. 一般規定 "$\to$" 是右結合的, 所以上述類型可以直接寫成 $A_1 \to \cdots \to A_n \to B$.

有了類型系統, 上面定義的 Ω 就不再合法了. 假定該表達式中 x 的類型為 T, 為了讓函數應用 $(x\,x)$ 合法, x 又必須是以 T 類型為輸入的函數, 這破壞了類型的分層結構 (一個函數當然只能以較低層次類型的項為輸入)[4], x 不能有確定的類型, 矛盾. 如此一來, 系統的不一致性就被排除了. 簡單類型 λ 演算中的每個表達式一定可以化為正規形式, 對應於一個會停機的計算.

* * *

在 Haskell 這樣的靜態類型語言中, 原則上每個變量和函數都必須有一個確定的類型, 實際使用中, Haskell 強大的類型推斷系統能根據上下文自動匹配類型, 所以不一定非要手動寫出顯式的類型簽名. 比方說, 如果我不加類型聲明地定義如下函數, 自動類型推斷就會給出第一行所示的類型簽名 (在 `ghci` 中定義函數後可用 `:t` 命令查看其類型):

```
1 f :: Num a => a -> a -> a
2 f x y = x + y
```

其中 `Num a` 表示類型 `a` 屬於 `Num` 這個類型類 (typeclass), 該類型類由所有支持算術操作的數值類型構成. 而後面的 `a -> a -> a` 與我們上文所述的函數類型完全一致. Haskell 在編譯程序前會事先檢查程序中的類型是否一致, 如果程序能通過編譯, 那麼至少在類型層面一定是沒問題的, 這也正是能用它來進行簡單定理證明的原因. 反之, 在 Python 這樣的動態類型語言中, 類型檢測大多是在運行時才發生, 所以在程序執行中可能出現 TypeError.

* * *

有了類型的概念, 我們再來看上面定義的 K 和 S 組合子. K 組合子接收兩個輸入參數, 輸出第一個, 所以它的類型為 $A \to B \to A$; 同樣容易看出, S 組合子的類型為 $(A \to B \to C) \to (A \to B) \to A \to C$. 顯而易見, 這兩個函數類型與直覺主義命題邏輯的兩條公理:

1. Ax1: $\varphi \to (\psi \to \varphi)$,

[4] 這裡的表述可能不那麼嚴格, 但大體就是這個意思.

2. Ax2: $(\varphi \to (\psi \to \rho)) \to ((\varphi \to \psi) \to (\varphi \to \rho))$.

之間具有驚人的對應性. 一般規定基本聯結詞 $\to$ 是右結合的, 這就意味著, 我們幾乎可以把兩個 "$\to$" 看作是同一回事![5] 組合子理論可進一步發展為組合邏輯 (combinatory logic), 其類型系統恰可對應於 Hilbert 演繹系統. 這是邏輯與類型之間對應性的一個重要例子, 它暗示我們可以把命題看作類型, 用函數 (λ 演算) 來表示命題間的構造模式. 例如, 在 λ 演算中, 我們有 $I = ((S\,K)\,K)$, K 對應於 Ax1, S 對應於 Ax2, 而 I 對應於恆同命題 $\varphi \to \varphi$, 所以這個等式給出了一個用 Hilbert 式演繹證明命題 $\varphi \to \varphi$ 的方法, 而要直接從 Hilbert 演繹系統中構造這個證明則並不容易.

1.3 Curry–Howard 同構

Curry–Howard 同構 (Curry–Howard correspondence) 是 Haskell Curry 和 William Alvin Howard 發現的一類現象: 類型與邏輯命題之間存在一種一一對應的關係. 這一點在上文中已經初露端倪, 下面就來具體說明這種對應性.

采用 Gentzen 式自然演繹, 我們重寫直覺主義命題邏輯的構造規則如下[7]:

$$\Gamma, \varphi \vdash \varphi (\mathrm{Ax})$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} (\to \mathrm{I}) \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi} (\to \mathrm{E})$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} (\wedge \mathrm{I}) \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} (\wedge \mathrm{E}) \quad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi} (\vee \mathrm{I}) \quad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \vee \psi} \qquad \frac{\Gamma, \varphi \vdash \vartheta \quad \Gamma, \psi \vdash \vartheta \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \vartheta} (\vee \mathrm{E})$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} (\bot \mathrm{E})$$

图 1: Intuitionistic Natural Deduction

每個命題聯結詞 $\star$ 分別對應一條引入規則 ($\star I$) 和一條消去規則 ($\star E$), 最上方的是恆同規則, 即從一個命題可以構造出其自身, 這條規則很容易用很容易用組合子 $I = \lambda x.x$ 構造出; 最下方的是爆炸原理, 其含義上文已經解釋過了. 容易驗證, 表中的自然推演規則正是上文中構造規則的形式化.

首先, 為實現 ($\to I$) 對應的類型演算, 只需構造一個類型為 $\varphi \to \psi$ 的函數即可. 對於 $x : \varphi$ 和 $M : \psi$, 我們可以構造 $(\lambda x : \varphi.\, M) : (\varphi \to \psi)$, 於是有如下推演規則:

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash (\lambda x : \varphi.\, M) : (\varphi \to \psi)}$$

[5] 所以使用同一個符號是有道理的.

反之, 為實現 $(\to E)$ 的類型演算, 只需把 $\varphi \to \psi$ 類型的函數 M 應用到 $N : \psi$ 即可, 於是有:

$$\frac{\Gamma \vdash M : (\varphi \to \psi) \quad \Gamma \vdash N : \varphi}{\Gamma \vdash (M\,N) : \psi}$$

而要實現 $(\wedge I)$, 我們同時需要 φ 的證明和 ψ 的證明, 於是很自然就會想到用 Descartes 積把兩個類型配成一對, 所以在本質上, 類型 $\varphi \wedge \psi$ 就相當於類型 $\varphi \times \psi$, 這被稱為**積類型** (Product Type), 據此就有推演規則:

$$\frac{\Gamma \vdash M : \varphi \quad \Gamma \vdash N : \psi}{\Gamma \vdash (M, N) : (\varphi \wedge \psi)}$$

反過來使用一個投影函數 π_i $(i = 1, 2)$ 即可從一個有序對中 "復原" 出合取支的證明:

$$\frac{\Gamma \vdash P : (\varphi \times \psi)}{\Gamma \vdash \pi_1(P) : \varphi}$$

類似合取規則, 析取類型 $\varphi \vee \psi$ 即為 $\varphi + \psi$ (稱為**和類型** Sum Type), 故 $(\vee I)$ 的實現非常簡單:

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash M : (\varphi \vee \psi)}$$

而 $(\vee E)$ 的構造則略有難度, 需要分情況考慮. 若 $x : (\varphi + \psi)$, 則或者 $x : \varphi$ 或者 $y : \psi$, 前一種情況應用 $\lambda x.Q$, 後一種情況應用 $\lambda y.R$, 用 | 算符可寫成:

$$\frac{\Gamma \vdash P : (\varphi \vee \psi) \quad \Gamma, x : \varphi \vdash Q : \vartheta \quad \Gamma, y : \psi \vdash R : \vartheta}{\Gamma \vdash (\lambda x.Q \mid \lambda y.R) : \vartheta}$$

這裡僅做形式表示, 具體函數定義見下文. 在構造類型推演規則時, 我們定義了新的類型, 其中使用的類型構造函數稱為**類型構造子** (Type Constructor), 在 Curry–Howard 同構中, 類型構造子對應於命題聯結詞.

此外, $\bot$ 對應於空類型, 它一定是不可構造的; $\top$ 對應於單元類型, 它一定是可構造的, 因為它恰有一個平凡的實例, 換句話說, 該命題有一個平凡的證明.

至此, 直覺主義自然演繹的8條規則都有了對應的類型演算規則.

一個類型稱為 "數據可居留的" (inhabitable), 當且僅當它至少有一個實例, 而這也意味著它對應的命題是可證的 (provable). 對於變量 $x : A$, 如果 A 表達一個命題, 那麼 x 就叫做 A 的可證性的一個 "見證" (witness), 換句話說, 也就是 A 為真的一個證據. 與集合論不同 (在集合論中, 先有元素, 再構成集合), 在類型論中, 先定義類型 (命題), 隨後才能給出它的實例 (證據). 而在一個程序中, 我們組織起所有這些 "證據", 就構造出了 A 的一個證明.

因此, Curry–Howard 同構可以概括為: "類型即命題, 程序即證明". 下表更詳細地總結了對應性:

邏輯	**類型論**
命題	類型
證明	程序

公理模式	組合子
假設 (前提)	變量
分離規則 (MP)	函數應用
邏輯聯結詞	類型構造子
邏輯真 ($\top$)	單元類型
邏輯假 ($\bot$)	空類型
可證性	類型可居留
證明檢驗	類型檢查

有了 Curry–Howard 同構, 下面我們就可以在 Haskell 中實現一些簡單的定理證明了.

* * *

這裡我們或許還有必要回答這樣一個問題, 即為什麼計算機輔助證明是可靠的? 畢竟我們至此似乎只是展示了數學證明和類型論之間一種名義上的關聯. 事實上, 由於證明與程序在構造過程上具有精確對應性, 程序設計中的每一個函數定義和類型轉換都嚴格反映了數學證明的一個步驟, 而函數式編程的特性確保程序每一次執行得出的結果必是唯一確定的, 這就保證了程序類型檢查的可靠性至少與形式演繹系統 (如 Hilbert 演繹系統和 Gentzen 自然推演) 的可靠性是相當. 而在具體實踐中, 編程語言的類型系統一般達不到定理證明所需的嚴格程度, 即便如 Haskell 這樣的語言, 充其量也只能進行一些簡單的定理證明[6]. 因此, 計算機輔助一般不會使用通用程序語言, 而會使用支持**依值類型** (Dependent Type, 其含義將在下文解釋) 的定理證明助手 (Proof Assistant) 來完成. 常用的定理證明助手包括 Lean, Agda (建立在 Haskell 之上), Rocq (原名 Coq) 等.

相對於數學證明的人工檢驗, 計算機輔助證明的優勢是很明顯的. 首先, 人在閱讀數學證明時容易疲勞, 粗心或是犯計算錯誤, 這對於計算機而言是完全可以避免的. 其次, 數學家寫出的證明往往是缺省形式, 根據前置知識和數學直觀省略了大量細節, 這就導致有可能出現不嚴格定義的問題, 也增加了讀者理解證明的難度, 而若要將證明形式化為證明助手可接受的形式, 就必須把省略的中間過程一一補全, 這就進一步排除了可能的錯誤或遺漏. 再者, 一些證明涉及大量機械檢驗, 其難度之大本就是人力所不能及, 最典型的例子如四色定理的形式證明, 即便在簡化後也留有600余種情況需逐一驗證, 如果不依靠計算機輔助, 這個證明根本是無法完成的. 事實上, 四色定理證明的原理是窮舉驗證, 與本文介紹的依賴於類型論的輔助證明不是同一回事, 但為了能夠相信 “計算機驗證” 的結果是正確的, 我們必須用證明助手對這個證明進行形式化[7]. 最後, 計算機輔助證明意味著數學證明的驗證不再需要付出高昂的人力成本, 如果一名數學家把自己的證明形式化, 那麼這個證明就應被承認為真, 這有助於打破學術出版界的資本壟斷, 有利於建立更加開放的學術社區.

[6] 復雜的定理證明原則上也是可以實現的, 但需要一些技巧和特殊處理, 效率會很低

[7] 可以理解為兩個領域的 “耦合”.

* * *

注: 關於 Curry–Howard 同構的更多歷史細節, 參見[8]. “程序員可能時常認為程序語言多少是隨意的, 但 Curry–Howard 同構表明, 程序在某些方面是絕對的 (absolute)”. 有人認為 Curry–Howard 同構是反對計算機軟件專利權的一大論據: 既然算法本質上是數學證明, 那麼對算法的專利權就意味著相應證明的專利權, 這暗示數學證明也可以被私有化, 專利化——而這無疑是荒謬的.[11]

2 Haskell 實現

2.1 命題邏輯

下面在 Haskell 中實現一些簡單的命題邏輯證明. (本部分主要參考[5])

首先, 我們引入 `Void` 數據類型和 `absurd` 函數:

```
1 import Data.Void (Void, absurd)
```

`Void` 是 Haskell 內置的空類型 (無法構造出 `Void` 類型的值), 對應於命題邏輯的 $\bot$ (謬); 而 `absurd` 函數的類型簽名為 `absurd :: Void -> a`, 對應於命題邏輯的爆炸原理 (“absurd” 正是 “荒謬” 的英文). 同樣, 單元類型 `()` 也是內置的, 唯一一個屬於該類型的值就是 `()`, 其類型簽名為 `() :: ()`. 單元類型對應於命題邏輯的 $\top$.

Haskell 也有內置的恆同函數 `id :: a -> a`, 該函數給出了恆同規則.

我們用遞歸類型 (Recursive Types) 來定義和類型與積類型:

```
1 data Prod a b = Pair a b
2 data Sum a b = Inl a | Inr b
```

其中 `Pair` 和 `Inl`, `Inr` 是自定義的類型構造子, 前者接受兩個參數將其配對; 後兩者各接受一個參數, `|` 算符表示 “或”, 所以 `Sum a b` 可理解為 `a` 或 `b`. 和類型與積類型建立在 “+” 與 “×” 的基本代數結構之上, 是函數式編程語言中定義新數據的基本方式, 統稱為**代數數據類型** (ADT).

使用 `Pair` 可以實現合取引入 ($\wedge I$). 反過來, 我們定義如下兩個函數來 “取出” 一個 `Pair a b` 中的第一項或第二項[8]:

```
1 fst' :: Prod a b -> a
```

[8] 函數名定為 `fst'` 是為了避免與內置函數 `fst` 沖突

```
fst' (Pair x _) = x

snd' :: Prod a b -> b
snd' (Pair _ y) = y
```

這樣就實現了合取消去 ($\wedge E$). 同樣道理, 一個變量是 `Sum a b` 類型的, 則它或者是 `a` 類型或者是 `b` 類型的, 定義一個函數對兩種情況分別處理即可:

```
sumCase :: Sum a b -> (a -> c) -> (b -> c) -> c
sumCase (Inl x) f _ = f x
sumCase (Inr y) _ g = g y
```

函數定義中的 `(Inl x)` 實際上是個語法糖, 對應的 `case` 表達式可寫成

```
sumCase :: Sum a b -> (a -> c) -> (b -> c) -> c
sumCase s f g = case s of
    Inl x -> f x
    Inr y -> g y
```

這樣就實現了析取消去 ($\vee E$).

在直覺主義命題邏輯中, $\neg$ 和 $\leftrightarrow$ 都是縮略式, 所以這裡也用函數類型的縮略式來表達:

```
type Not a = a -> Void
type Iff a b = Prod (a -> b) (b -> a)
```

其中, `type` 關鍵詞的作用是定義類型別名, 方便後續使用.

* * *

為證明假言三段論 (Hypothetical Syllogism, HS), 即

$$\vdash (\varphi \to \psi) \to (\psi \to \rho) \to (\varphi \to \rho)$$

只需構造如下函數:

```
hs :: (a -> b) -> (b -> c) -> (a -> c)
hs atob btoc =
    \a -> btoc $ atob a
```

或者更簡單地, 可以不用 λ 演算, 直接用函數復合, 寫成:

```haskell
hs :: (a -> b) -> (b -> c) -> (a -> c)
hs atob btoc =
    btoc . atob
```

這兩種寫法是等價的, 都是把 `b -> c` 的函數應用到 `a -> b` 的函數上, 得到 `a -> c` 的函數.

* * *

下面是 $\leftrightarrow$ 的一些性質:

1. 單側消去: $\vdash (\varphi \leftrightarrow \psi) \to (\varphi \to \psi)$.

```haskell
elimLeft :: Iff a b -> (a -> b)
elimLeft (Pair atob _) = atob
```

2. 自反性: $\vdash \varphi \leftrightarrow \varphi$.

```haskell
refl :: Iff a a
refl = Pair id id
```

3. 對稱性: $\vdash (\varphi \leftrightarrow \psi) \to (\psi \leftrightarrow \varphi)$.

```haskell
symm :: Iff a b -> Iff b a
symm (Pair atob btoa) = Pair btoa atob
```

4. 傳遞性: $\vdash (\varphi \leftrightarrow \psi) \to (\psi \leftrightarrow \rho) \to (\varphi \leftrightarrow \rho)$.

```haskell
trans :: Iff a b -> Iff b c -> Iff a c
trans (Pair atob btoa) (Pair btoc ctob) =
    Pair (hs atob btoc) (hs ctob btoa)
```

5. $\vdash (\varphi \leftrightarrow \psi) \to (\rho \leftrightarrow \vartheta) \to (\varphi \wedge \rho \leftrightarrow \psi \wedge \vartheta)$.

```
andIff :: Iff a b -> Iff c d -> Iff (Prod a c) (Prod b d)
andIff (Pair atob btoa) (Pair ctod dtoc) =
    Pair (\ac -> Pair (atob $ fst' ac) (ctod $ snd' ac))
    (\bd -> Pair (btoa $ fst' bd) (dtoc $ snd' bd))
```

尤其值得注意的一點是, 在直覺主義命題邏輯中, "↔" 不滿足結合律, 即

$$((\varphi \leftrightarrow \psi) \leftrightarrow \rho) \leftrightarrow (\varphi \leftrightarrow (\psi \leftrightarrow \rho))$$

一般是不可構造的[7]. 因此這一直觀上顯然的命題在直覺主義中也不被承認, 這再一次反映出直覺主義邏輯與古典邏輯的巨大差異.

* * *

最後, 我們來討論關於 "¬" 的一些定理. 這是直覺主義邏輯與古典邏輯差異最顯著的部分, 這裡的函數構造也會頗具難度.

1. $\vdash \top \vee \top \to \top \vee \bot$.

```
tfIff :: Iff (Sum () ()) (Sum () Void)
tfIff = Pair (\(Inl a) -> Inl a) (\(Inl a) -> Inl a)
```

2. 雖然無法在直覺主義邏輯中證明排中律, 但我們可以證明排中律是不可拒絕的, 即 $\vdash \neg\neg(\varphi \vee \neg\varphi)$. 為此, 我們需要構造一個類型為 `Not (Not (Sum a (Not a)))` 的項, 按 `Not` 的定義展開, 可知要求的類型是 `((Sum a (a -> ⊥)) -> ⊥) -> ⊥`, 即要求這樣一個函數, 對它輸入一個類型為 `(Sum a (a -> ⊥)) -> ⊥` 的函數後, 輸出為 `Void`. 假定這個 `(Sum a (a -> ⊥)) -> ⊥` 類型的函數為 `f`, 我們想要構造一個 `(Sum a (a -> ⊥))` 類型的項並將 `f` 應用到其上. 由於 `a` 是未定的, 我們沒法直接從 `a` 出發構造該項, 所以先試著構造一個 `a -> ⊥` 類型的函數 `g`, 借用 `f` 輸出 `Void` 的性質, 只要用 λ 演算令 `g = \x -> f $ Inl x` 就行了. 有了 `g`, 就可以反過來用 `Inr g` 構造所需的項, 並得到所需函數. 於是便有如下構造:

```
emIrrefutable :: Not (Not (Sum a (Not a)))
emIrrefutable f = f $ Inr $ \x -> f $ Inl x
```

3. 排中律與雙重否定消去律等價, 所以雙重否定消去律是不可構造的. 但如果有排中律的前提, 就可證明雙重否定消去律, 即 $\vdash (\varphi \vee \neg\varphi) \to (\neg\neg\varphi \to \varphi)$. 這裡的函數的構造與證明析取消去時差不多, 都是分兩種情況分別處理, 如果變量為 `a` 類型, 就直接返

回該變量, 否則它在 `f :: ((a -> ⊥) -> ⊥)` 的作用下轉換為 `Void`, 再用爆炸原理即可.

```
emToDne :: Sum a (Not a) -> (Not (Not a) -> a)
emToDne (Inl x) f = x
emToDne (Inr x) f = absurd $ f x
```

4. 雙重否定引入律: $\vdash \varphi \to \neg\neg\varphi$. 翻譯成類型論即是要構造類型為 `a -> (a -> ⊥) -> ⊥` 的函數, 這實際是分離規則的特例:

```
dni :: a -> Not (Not a)
dni x f = f x
```

5. 最後, 我們來證明上文已經用邏輯方法證明過的結論 $\vdash \neg\neg\neg\varphi \to \neg\varphi$. 翻譯一下, 即是要構造類型為 `(((a -> ⊥) -> ⊥) -> ⊥) -> (a -> ⊥)`, 它接受一個 `(((a -> ⊥) -> ⊥) -> ⊥)` 類型的函數 `f` 和一個 `a` 類型的變量 `x`, 輸出 `Void`. 這裡的想法與邏輯推演是一樣的, 先用 `dni` 引入雙重否定, 再應用 `f`:

```
dne :: Not (Not (Not a)) -> Not a
dne f x = f $ dni x
```

* * *

上文我們在這個程序中定義了一些抽象的類型和函數, 它們不接受具體輸入, 也不產生輸出, 所以這不是一個傳統意義上可執行的程序. 那麼要怎麼 "證明" 這些定理呢? 答案是, 如果這個程序能通過編譯, 它所表述的定理就被證明了, 而不需要實際運行. 更簡單地, 如果使用 `VSCode + Haskell Language Server` 環境, 只要編輯器下沒有劃紅線報錯就可以了 (類型檢查是自動的). 反過來, 如果我們確實定義一些數據輸入給這個程序, 那就基本可以斷定它是可運行的, 換句話說, 我們也證明了這個程序寫得是正確的. 然而, 在一些更復雜的情形下, 可能出現的錯誤不只涉及某個類型的項是如何構造的, 還涉及這些項具體的值, 簡單類型 λ 演算不足以排除這些錯誤, 需引入依值類型這樣更強的類型系統, 而這在 Haskell 中是不支持的.

2.2 依值類型

首先來看這樣一段代碼:

```haskell
List1 :: [Int]
List1 = [1, 2, 3]

a :: Int
a = List1 !! 5
```

這段代碼在類型方面是完全正確的, 所以能通過 Haskell 的類型檢查, 但在運行時會報錯, 因為 `List1 !! 5` 取列表中第5個值, 但該列表總共只有3個值. 這一般被稱為越界訪問 (Out-of-bounds access) 問題, 靠 Haskell 的類型系統是無法在編譯階段就排除掉的.

而在支持依值類型的語言中, 我們可以定義形如 `Vec n a` 的向量, 限制其長度為 `n`, 類型為 `a`, 這樣就可以規避越界訪問的錯誤了.依值類型本質上就是允許我們依據一個具體的值來構建類型, 在此, 這個值就是向量的維數.

* * *

在定理證明方面, 依值類型使謂詞邏輯的形式化成為可能. 比方說如下例子[1]:

$$\exists x : \mathbb{N}.x + 1 = 5$$

其中, 命題函數 $p(x) := (x + 1 = 5)$ 是一個自然數到命題的映射, 命題 $p(x)$ 的類型取決於 x 具體的值, 寫成映射即

$$p : \mathbb{N} \to \star$$
$$p(x) : (x + 1 = 5), x : \mathbb{N}$$

於是整個命題 $\exists x : \mathbb{N}.x + 1 = 5$ 就可以寫成兩個類型的積類型, 其中每一項由某個 x_0 和 $x_0 + 1 = 5$ 的證明構成. 換句話說, 其類型為:

$$\mathbb{N} \times (x + 1 = 5)$$

由於第二部分的 x 依賴於第一部分選擇的 x, 這不是一個簡單的積類型, 我們稱這樣的依值類型為 Σ 類型. 一般地, 任意存在性命題 $\exists a : t.p(a)$, 其對應的類型可表示為:

$$\sum_{a:t} p(a)$$

類型推導規則為:

$$\frac{\Gamma \vdash t : * \quad \Gamma, a : t \vdash b : p}{\Gamma \vdash (a, b) : \sum_{a:t} p}$$

其中 "$*$" 標記了 "類型的類型", 換言之: "$t : *$" 表示 "t 是一個類型", 這裡用到了高階類型. 直觀來看, 對任意類型 t, 只要有一個 $a : t$ 可推出 $b : p$, $\sum_{a:t} p$ 就是可構造的, 這也是稱之為 Σ 類型的原因.

類似地, 對於全稱命題 $\forall a : t.p(a)$, 我們需要構造從 $a : t$ 到 $p(a)$ 的證明的一個函數, 所以整個命題的類型為 $t \to p$. 我們稱這樣的依值類型為 Π 類型, 表示為:

$$\prod_{a:t} p(a)$$

類型推導規則為:

$$\frac{\Gamma \vdash t : * \quad \Gamma, a : t \vdash b : p}{\Gamma \vdash (\lambda a.b) : \prod_{a:t} p} \tag{1}$$

$$\frac{\Gamma, t : * \vdash M : \prod_{a:t} p \quad \Gamma, t : * \vdash n : t}{\Gamma, t : * \vdash (M\, n) : p[n/a]} \tag{2}$$

其中 $p[n/a]$ 表示用 n 替換 p 中出現的所有 a.

依值類型給出了 Curry-Howard 同構的一個更廣泛的形式, 此前的命題邏輯類型演算可以看作依值類型演算的特殊情況. 例如, 命題邏輯公式 $p \to q$ 可以看作謂詞邏輯的 $\forall x : p.q$, 其中 q 不依賴於 x.

* * *

雖然 Haskell 自身不支持依值類型, 但我們可以用廣義代數數據 (GDATs) 來 "模擬" 依值類型, 以實現一些簡單的帶量詞定理證明.

這裡以 `Codewars` 上的一道題為例[6]. 我們想要證明: 在自然數中, 一個偶數加上一個奇數得到的一定是奇數. 為此, 首先定義自然數類型:

```
1 data Nat = Z | S Nat
```

其中 `Z` 代表 `0`, 構造子 `S` 代表後繼運算. 然後遞歸定義 "偶數" 和 "奇數": `Z` 是偶數, 若 `n` 是偶數, 則 `S (S n)` 是偶數, 奇數同理:

```
1 data Even (n :: Nat) where
2     ZeroEven :: Even Z
3     NextEven :: Even n -> Even (S (S n))
4
5 data Odd (n :: Nat) where
6     OneOdd :: Odd (S Z)
7     NextOdd :: Odd n -> Odd (S (S n))
```

這兩個類型的定義都使用了 `GDAT`, 即用一個顯式的類型簽名來定義類型構造子, 把返回結果的類型固定下來.

題目的要求是構造這樣一個函數 `evenPlusOdd :: Even n -> Odd m -> Odd (Add m n)`, 為此先得自己定義一個表示加法的 `type family` 即 `Add`. 這個定義基本上是標准寫法, 但由於要求函數的第二個參數與結果都是 `Odd`, 我們也讓 `Add` 在遞歸時匹配第二個參數.

```haskell
type family Add (n :: Nat) (m :: Nat) :: Nat where
    Add n Z = n
    Add n (S m) = S (Add n m)
```

注意這個 Add 是類型層面的函數, 而不是通常意義的加法, 如果用在 n :: Nat 上會出現 Illegal term-level use 錯誤.

而函數的定義同樣是利用遞歸, 用 ZeroEven 去加一個奇數得到奇數自身, 用偶數 n 的下一個偶數去加奇數 m 則相當於 n + m 的下一個奇數. 這麼說可能有些囉嗦, 但看代碼是很簡潔的:

```haskell
evenPlusOdd :: Even n -> Odd m -> Odd (Add m n)
evenPlusOdd ZeroEven n = n
evenPlusOdd (NextEven n) m = NextOdd $ evenPlusOdd n m
```

能這樣寫的關鍵是我們調整了 Add 在遞歸時匹配參數的位置. 如果按通常寫法讓它匹配第一個參數, 即

```haskell
type family Add (n :: Nat) (m :: Nat) :: Nat where
    Add Z m = m
    Add (S n) m = S (Add n m)
```

那麼這個函數的實現就會相當麻煩. 下面是 AI 給的代碼 (甚至使用了一個 “引理”):

```haskell
evenToOddS :: Even n -> Odd (S n)
evenToOddS ZeroEven = OneOdd
evenToOddS (NextEven en) = NextOdd (evenToOddS en)
-- Lemma: The successor of an even number is odd.

evenPlusOdd :: Even n -> Odd m -> Odd (Add m n)
evenPlusOdd en OneOdd = evenToOddS en
evenPlusOdd en (NextOdd om) = NextOdd (evenPlusOdd en om)
```

同樣, 如果函數中要求的類型是 Odd (Add n m) 而非 Odd (Add m n), 那我們采用的簡潔寫法也就無效了, 在函數的參數匹配和 type family 的遞歸方向之間的確有重要的關聯 (這是由編譯器的自動類型匹配決定的). 對此, AI 給的總結 (Takeaway) 是: “型別家族對哪一邊做遞回, 你的函式就必須對哪一個參數做模式匹配.”

這個例子看起來與我們前面的理論表述不太相像, 但該例確實用到了依值類型. 在 evenPlusOdd 函數中, 輸入的 m, n 本質上是數值, 而結果的類型 Odd (Add m n) 正是由這兩個具體

值決定. 把諸如 `Even Z` 的類型看作諸如 0 是偶數的證明, 那麼函數類型給出了形如 Even → Odd → AddToOdd 的命題類型, 如上文所述, 這是一個 Π 類型, 對應於一個全稱量詞命題.

儘管可以在 Haskell 中模擬依值類型, 但通過例子也能看出, 這種寫法非常困難且具有相當程度的技巧性. 在真正的依值類型語言中, 類型 (type) 和值 (value) 之間沒有明確的分界線, 在 Haskell 中我們需要在值層面和類型層面分別定義兩個 `Add` 函數 (前者是常規函數而後者作為 `type family`), 而在依值類型語言中, 我們只需要定義一個值層面的函數 `Add : Nat -> Nat -> Nat`, 編譯器就能自動將其推導到類型上, 這當然極大地方便了程序證明.

那麼, Haskell 是怎麼模擬依值類型的呢? 換句話說, 它是如何在常規函數和 `type family` 之間建立對應性的呢? 事實上, 我們在諸如 `Even n` 中使用的 `n` 不是真正的值 `n :: Nat`, 而是我們透過 `{-# LANGUAGE DataKinds #-}` 擴展, 在類型層面復制了一份名叫 `Nat` 的 `Kind` (相當於類型的類型), 裡面的類型叫做 `Z` 和 `S (S ...)`, 它們與 `data Nat = Z | S Nat` 定義的數據截然不同. 所以問題就轉化為: 如何在類型層次的自然數和值層次的自然數之間建立聯系. 答案是通過**單例類型** (Singleton)[4]:

```
data Z
data S n

data Nat a where
    Zero :: Nat Z
    Succ :: Nat a -> Nat (S a)
```

前兩行代碼完全抽象地定義了類型 `Z` 和 `S n`, 我們甚至無法具體構造出這兩個類型的相應值. 而後面我們用 `GDAT` 把具體的值與這些類型聯系起來, 並保持了同構關係. 例如 `Succ (Succ Zero) :: Nat (S (S Z))`, 每個類型恰好有一個實例, 並且從實例的形象一眼就能看出它屬於哪個類型, 這樣兩個層次就對應起來了. 但是, 如果要進一步定義 “自然數” 的運算, 我們又得在兩個層次分別工作, 定義兩套截然不同的函數操作. 這一切在專門的定理證明助手中都是自動完成的.

3 總結

上文概述了計算機輔助證明的原理, 並基於 Haskell 給出了一些簡單的實現. 簡單來說, 通過 Curry-Howard 同構, 我們在直覺主義邏輯與類型論之間建立起對應關係, 一個類型系統越復雜, 它能 “表述” 的邏輯命題也越廣泛. 使用簡單類型 λ 演算就足以形式化直覺主義命題邏輯, 而在引入依值類型的系統中, 我們可以進一步實現謂詞邏輯. 依值邏輯的一種常見實現方式是**構造演算** (Calculus of Constructions, CoC), 這是一種高階類型 λ 演算, 也被

記作 λC, 它既支持定義依賴於項的項和類型, 也支持依賴於類型的項和類型. 由於這種高表達力, 它是證明助手如 Lean, Rocq 等的理論基石.

還需要注意一點, 計算機輔助證明不同於現在很流行的 AI 輔助證明, 即便兩者間有很大的相關性. 我們在這裡介紹的輔助證明, 其實更接近於 “驗證”, 是由數學家先寫好一個證明, 然後使用程序語言形式化, 來確保其正確性, 在寫出證明的過程中, 原則上並不需要計算機輔助. 而廣義的計算機輔助證明則包括自動推理技術, 例如使用啟發式搜索 (heuristic search), 就有可能證明新的結果, 或是找到原先結果的新證明. 但是, 我認為這類自動推理技術的應用不會降低數學家的重要性, 儘管計算機能完全機械地尋找定理及證明, 這不意味著它能夠獨自判斷哪一種結果是有價值的, 所以它搜索出的結果很可能是平庸瑣碎的, 必須由人來讀懂這些結果, 從中挑選出真正有價值的內容, 並指導計算機自動搜索的方向. 這就像是計算機的出現對數學帶來的影響, 如今任何人都能用計算機計算數百萬個素數, 但不是每個人都能像 Gauss 一樣從中看出素數定理. 至於這種變化是否會使數學轉型為一種類似天文學的觀察學科, 或一種 “解釋學”? 對此學界尚有爭議.

4 參考

1. 直觉主义逻辑和程序定理验证 Link
2. 计算机辅助证明简介 Link
3. 无类型 λ 演算 Link
4. Haskell 中的“定理证明” Link
5. Shanghai Tech ACM A0019. Haskell 不是一款你的定理证明器 Link
6. Odd + Even = Odd? Prove it! Link
7. Morten Heine Sørensen, Pawel Urzyczyn. Lectures on the Curry-Howard Isomorphism (2006). Elsevier. (及其翻譯版 Link)
8. Philip Wadler. Propositions as Types (2014). Link
9. Rob Nederpelt, Herman Geuvers. Type Theory and Formal Proof: An Introduction (2014). Cambridge University Press.
10. Wikipedia, Brouwer–Heyting–Kolmogorov interpretation. Link
11. Wikipedia, Curry–Howard correspondence. Link
12. Wikipedia, Combinatory logic. Link
13. Wikipedia, Calculus of constructions. Link
14. Wikipedia, Computer-assisted proof. Link