情報論理学 2016 年第 6 週分ハンドアウト

担当：松田一孝

4 直観主義命題論理と型

クイズ

a^b が有理数になるような，無理数 a と b は存在するか？

証明. $t = \sqrt{2}^{\sqrt{2}}$ を考える．t が有理数か否かで場合分けする．

もし t が有理数ならば，$a = b = \sqrt{2}$ と存在する．もし t が無理数ならば，$t^{\sqrt{2}} = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = \sqrt{2}^2 = 2$ と有理数となるので，$a = t, b = \sqrt{2}$ と存在する． □

この証明は古典論理（classical logic）においては正しいものである．しかしながら，この証明からは結局 a と b は具体的に何だったのかはわからない．このような証明を**非構成的**（non-constructive）であるという．

一方，$a = \sqrt{2}$ および $b = 2\log 3$（このとき $a^b = 3$）ととればこの命題は証明できる．このような，具体的な a と b が得られる証明を**構成的**（constructive）であるという．

直観主題論理（intuitionistic logic）は構成的な証明のみを許す論理である．たとえば，$A \vee B$ が直観主義論理で証明できるのは，A が証明できるか，B が証明できるときのみである．**直観主義命題論理**（intuitionistic propositional logic）は命題論理の直観主義版である [1]．具体的には，直観主義命題論理の自然演繹の推論規則は，（古典）命題論理のものから背理法を除いたものである．前回，古典命題論理において $P \vee \neg P$ の証明をしたことを思いだしてみよう．そこでは，P が成り立つことを示したのでも，$\neg P$ が成り立つことを示したのでもなく，背理法を用いて示したのであった．上の非構成的な証明においても，$\sqrt{2}^{\sqrt{2}}$ が「有理数か無理数かのいずれかである」という排中律が用いられていることに注意する．

4.1 Curry-Howard 対応

実は，直観主義論理とプログラミング言語の型システムとは関連が深い．具体的には，ある特定のプログラミングにおいて「型 A を持つ項が存在する」ことと，ある証明体系において「命題 A が証明できる」ことが同値になる（Curry-Howard 対応）．さらには，「式の評価」と「証明の簡単化」が対応する（Curry-Howard 同型）が今回はここまでは触れない．

このことは，プログラムを書くように証明を書くことを可能にする．実際に，Coq や Agda はこの考えに基づいて設計された定理証明支援系であり，証明を「プログラム」することができる [2]．さらに，これらのシステムでは型システムが証明の正しさを保証してくれるのだ．このことは，人間が理解しづらいような，論文数百ページにおよぶ証明を持つ定理が実際に正しいことを確認するのに有効だ．また，プログラムとの親和性の高さは，プログラムとその正しさの証明を同時に行うような certified programming を可能とする．

直観主義命題論理程度の論理ならば，Standard ML の（とても小さな）サブセットを用いて証明をプログラミングすることが可能だ．本講義ではそれを紹介しよう．

[1] 直観主義述語論理というのもあり，それは（古典）述語論理（次回以降扱う）の直観主義版である．こちらでは，$\exists x.A$ が証明できるのは $A[a/x]$ を満たすような具体的な個体 a が得られるときのみである．

[2] ただし，Coq ではユーザが直接証明をプログラムするのではなく，tactic というものを用いてプログラムを構成するという多段構造になっている．形式的証明を「プログラム」するのは紙の上で形式的証明するよりは楽だが，それでも通常の自然言語による証明よりもはるかに面倒ではあるのだ．

4.2 関数抽象，適用

以下の Standard ML に似たとても小さなプログラミング言語を考えてみよう．

$$e ::= x \mid \mathbf{fn}\ x \Rightarrow e \mid e_1\ e_2$$

すなわち，この言語では式は変数か，関数抽象か適用かのいずれかである．

さて，このサブセットの式について型を付けることを考えてみよう．まず，変数である．変数が型 A を持つということは，これまでの時点で「この変数が型 A を持つ変数である」と導入されているということだし，導入されていればよい．このような「過去に導入された変数とその型」を表すのに**型環境**を用いる．型環境は，$x : A$ という形の要素からなる列である．「型環境 Γ のもので，式 e が型 A を持つ」ことを表す判断を $\Gamma \vdash e : A$ と書くことにする．すると，変数の型付け規則，つまり型付けのための推論規則は

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\ (\text{T-Var})$$

と書ける．要するに「これまでのどこかで x が型 A の変数として導入されていたならば，x は型 A を持つ」ということである．

次に関数抽象 $\mathbf{fn}\ x \Rightarrow e$ を考えてみよう．関数抽象なので関数型 $A \to B$ を持つはずだ．では，どういうときにこの型を持つのかというと，x を型 A の変数だとしたときに e が型 B を持つこととときである．つまり，型付け規則は

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \mathbf{fn}\ x \Rightarrow e : A \to B}\ (\text{T-Abs})$$

となる．

最後に関数適用を考えてみる．$e_1\ e_2$ と関数 e_1 を適用しているのだから，e_1 は $A \to B$ のような関数型を持っていなければならない．また，e_2 は e_1 の引数なので型 A を持っていなければならない．そして，このときに $e_1\ e_2$ は型 B を持つ．すなわち，型付け規則は

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ (\text{T-App})$$

となる．

注意深い学生は気づいたかもしれないが，これまでの型付け規則は自然演繹の規則にとてもよく似ている．実際に変数や式の情報を消せば対応する自然演繹の推論規則が得られる．逆に自然演繹における導出から変数や式の情報を復元可能である．このことは対応する規則を隣同士に並べてみればより顕著になるだろう．

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}\ (\text{T-Var}) \qquad \frac{}{\Gamma, A, \Delta \vdash A}\ (\text{Ax})$$

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \mathbf{fn}\ x \Rightarrow e : A \to B}\ (\text{T-Abs}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ (\Rightarrow\text{I})$$

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1\ e_2 : B}\ (\text{T-App}) \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash B}{\Gamma \vdash B}\ (\Rightarrow\text{E})$$

実際に以下が証明できる．

定理 4.1 (Curry-Howard 対応)**.** 以下の二つは同値である．

- 自然演繹で $\vdash A$ が導出できる
- この言語にて $\vdash e : A$ なる e が存在する．すなわち型 A を持つ式が存在． □

この意味ではこの小さなプログラミング言語の式は，導出においてどの推論規則を用いたかを表しているとも言える．そのため**証明項**と呼ばれることもある．

ここで扱った小さなプログラミング言語は**単純型付き λ 計算**と呼ばれるものである．ほとんどの実用的な型付き関数プログラミング言語は単純型付き λ 計算をサブセットとして含んでいる．

4.3 組

さてこのプログラミング言語を少し拡張し，二つ組に関する演算を追加してみよう．

$$e ::= \cdots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

ここで，(e_1, e_2) は第一要素と第二要素がそれぞれ e_1 と e_2 の評価結果であるような組を作成し，$\#1\ e$ は e の評価結果の組の第一要素を返し，$\#2\ e$ は第二要素を返す．型 A の要素と型 B の要素からなる組の型を $A \times B$ と書くことにする．$A \times B$ は A と B の直積型と呼ばれる．ML において，$A * B$ と書かれるのは，$\times$ が ASCII にないためであろう．

これらに対する型推論規則と，対応する自然演繹の型付け規則は以下である．

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1, e_2) : A \times B}\ (\text{T-Pair}) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge\text{I})$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \#1\ e : A}\ (\text{T-Fst}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\text{E})$$

$$\frac{\Gamma \vdash e : A \times B}{\Gamma \vdash \#2\ e : B}\ (\text{T-Snd}) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\text{E})$$

定理 4.1 は同様に成り立つ．

4.4 直和

さてこのプログラミング言語を少し拡張する．

$$e ::= \cdots \mid \mathsf{InL}\, e \mid \mathsf{InR}\, e \mid \textbf{case}\ e_0\ \textbf{of}\ x \Rightarrow e_1 \mid y \Rightarrow e_2$$

すなわち，(S)ML において以下のヴァリアント型と，その値に対する網羅的な **case** による分岐を許すことに対応する．

```
type ('a,'b) plus = InL of 'a | InR of 'b
```

ここで，(A, B) `plus` と書くかわりに見やすさのためここでは $A + B$ と書くことにする．$A + B$ は A と B の直和型と呼ばれる．

これらに対する型推論規則と，対応する自然演繹の型付け規則は以下である．

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{InL}\, e : A + B}\ (\text{T-Left}) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee\text{I})$$

$$\frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{InR}\, e : A + B}\ (\text{T-Right}) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee\text{I})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : A + B \\ \Gamma, x : A \vdash e_1 : C \\ \Gamma, y : B \vdash e_2 : C\end{array}}{\Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ x \Rightarrow e_1 \mid y \Rightarrow e_2 : C}\ (\text{T-Case}) \qquad \frac{\begin{array}{c}\Gamma \vdash A \vee B \\ \Gamma, A \vdash C \\ \Gamma, B \vdash C\end{array}}{\Gamma \vdash C}\ (\vee\text{E})$$

定理 4.1 は同様に成り立つ．

4.5 $\top$, $\bot$

$\top$ については簡単だ.

```
type top = Top
```

と要素が一つだけの型を導入するか，あるいは ML には既にそういう型 `unit` が存在するので，以下のようにそれの別名にしてしまえばよい.

```
type top = unit
```

$\bot$ については少し注意が必要だ．式としては

$$e ::= \cdots \mid \mathbf{efq}\ e$$

と拡張し，その型付け規則を

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash \mathbf{efq}\ e : A}\ (\text{T-Efq}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot\text{E})$$

と用意すればよい.

問題は Standard ML 等の具体的な関数プログラミング言語における実現である．直感的には $\bot$ は証明されるべきではないのだから，何も要素を持たない型としてしまえばよい．しかしながら，多くの言語では，0 個の構成子からなるヴァリアントの宣言や，その型の値に対する 0 個の分岐を持つ分岐等は許されていない（Agda や Coq ではできる）.

OCaml や Haskell (GHC 拡張) 等のより強力な型システムを持つ言語であれば,

```
type bot = { efq : 'a. 'a }
```

や

```
newtype Bot = Bot { efq :: forall a. a }
```

のように高ランク多相を用いればよい．いずれも $\bot$ に対応する型の値は「任意の型を持つ値」としている．もちろん，そんな「値」は存在しない.

しかしながら，SML ではいずれのアプローチも取れないので，再帰定義を用いて

```
type bot = Bot of bot
fun efq (Bot b) = efq b
```

とする.

否定 $\neg A$ は $A \Rightarrow \bot$ の別名として導入してしまえばよい.

```
type 'a not = 'a -> bot
```

なお，以下の定理が知られている.

定理 4.2. 古典命題論理の自然演繹で $\vdash A$ が導出可能ならば，そのときに限り直観主義命題論理の自然演繹で $\vdash \neg\neg A$ が導出可能. □

5 callcc と古典命題論理

一部のプログミング言語では背理法に対応する型規則が存在する．SML/NJ では SMLofNJ.Cont に定義されている二つの関数

```
val callcc : ('a cont -> 'a) -> 'a
val throw : 'a cont -> 'a -> 'b
```

を用いて背理法（正確には二重否定除去だが違いは構文上引数が必要かそうでないかぐらいである）を実現することができる．

ここで callcc e は，この式「残りの計算」（**継続**と呼ぶ）を引数として e を評価する．そして，throw k e は，今の「残りの計算」を打ち捨て，e の評価結果を「残りの計算」k に渡す．ものすごく大雑把にいえば，関数プログラミング版のラベルと goto のようなものである．

これにより，二重否定除去を実現することができる．

```
fun dne (k : ('a -> bot) -> bot) =
   callcc (fn cont => efq (k (throw cont)))
```

これによって定義可能な排中律はなかなか興味深い挙動をする．