Leanにおける選択公理

神田さなえ
指導教員：松下大介

2024/1/17

Abstract

数学の諸定理の証明に関わる選択公理は、定理証明支援系においてはどのように取り扱われるのか。これを定理証明支援系Lean上でのDiaconescuの定理の証明を利用して考察していきたい。また、Leanは型理論に基づいている。このためLean上での数学的事項の記述は通常と少し異なる記述で行われる。これも併せて確認し、型理論上での論理の展開がどうなるかを把握し、証明支援系への理解を深めたい。

通常数学で扱われるものは複数の特性を持つ事が了解されている。しかし型理論では、そのようなラベリング（型付け）は概ね一対一で行われる。このため、例えば3が自然数全体の集合$\mathbb{N}$の元であると同時に整数$\mathbb{Z}$の元であることは通常の数学では了解されている。しかし、Leanでは同時にこの二つのラベルを貼ることは不可能であるため、しばしば工夫を要する。（大抵は追加の命題を用意することで性質を保証する）
型理論は集合の考え方とよく似ており、集合Aに属する元aと型Aを持つ項aが類似の関係性を持っている。（型Aも項として取り扱う事ができる。また、全ての項は型を持つ。）大きな相違点としては集合論では矛盾を回避するために制限があるが、集合の集合が集合であるように定義づけの階層に制限がある。一方型理論は階層に制限がなく、厳密に階層化されている。そのため、ラッセルのパラドックスのような状況を回避できる。この厳密な階層構造はプログラムに適切な変数が与えられ、正常に動作することを保証することにも利用される。（型安全性と言う）同様にして、プログラムとして記述された証明の正しさを担保している。
では実際にどういった形式で命題が表されるのか、これを選択公理を用いて確認する。 Leanでの選択公理はLeanではClassical.choiceという名称で扱われる。 Classical.choiceは以下のような型を持つ項である。

```
{α : Sort u} → Nonempty α → α
```

Lean内での選択公理Classical.choiceは関数型と呼ばれる項で、型Sort uの項αとα が空でないとする命題Nonempty α（これはProp型の項である）によって α型の項を導く関数という属性を持つ項である。 Leanでは同時に次のような説明も行われる。

```
Classical.choice.{u} {α : Sort u} (a : Nonempty α) : α
```

これは実際に関数に値を代入した場合の形式を表示している。aはNonempty α型の項で、上で述べたように Nonempty αはProp型の項、すなわち命題である。LeanにおいてProp型の項を型に持つ項はその命題の証明そのものとなる。このNonempty αはαが空でないという命題なので、その証明が与えられると選択公理の要件を満たし、同時にαも与えているので略記され、関数Classical.choiceに項aを与えたClassical.choice aはα型の項となる。このようにLeanでは命題は通常Prop型を持つが、定理や公理に関数型を持たせる場合もある。
下にDiaconescuの定理を示す。

Theorem 0.1. *Diaconescu's Theorem*

if $\forall\{U_\lambda\}_{\lambda\in\Lambda}(\forall\lambda\in\Lambda, U_\lambda\neq\phi)$, $\exists f:\Lambda\longrightarrow\bigcup_{\lambda\in\Lambda}U_\lambda s.t. f(\lambda)\in U_\lambda$, *then for all proposition P, satisfying P or not P.*

証明の方針としては任意の命題Pに対して、命題を要素とする集合U,Vを置く。UとVをうまく取ると選択公理を用いてU,Vそれぞれのある要素u,vを取り出し、Pならばu=v、および、Pまたはu≠v、を言う事ができるので、前者の対偶を取ることで証明が可能である。以下でLean上で選択公理に基づく排中律の証明を行う。なお、以下のプログラムは引用サイトTheorem Proving in Lean 4 の12章の節ChoiceとThe Law of Excluded Middleから抜粋した。

```
import Mathlib
open Classical
```

```
theorem em_reproof (p : Prop) : p ∨ ¬p :=
let U (x : Prop) : Prop := x = True ∨ p
let V (x : Prop) : Prop := x = False ∨ p
```

命題を要素とする集合U,Vを置いている。このU,VだがLean上ではProp型からProp型への関数型として定義した。集合は条件となる命題によって構成できる。Leanではこれはある型αからProp型への関数型の項として表現される。 Lean上での集合の定義、及びその要素が集合に属することを示す命題は以下のようになっている。

```
def Set (α : Type u) := α → Prop

protected def Mem (a : α) (s : Set α) : Prop :=
  s a
```

集合を表すSetが写像である事が確認できる。条件となる命題がkであれば、当然その集合に属する元はkを満たす。この条件はdef Memに当たる。
従って、上で定義したU,VがLean内では集合として取り扱えることがわかる。これらは当然、pが真ならば全ての命題が要素になる。証明に戻る。

```
 have exU : ∃ x, U x := ⟨True, Or.inl rfl⟩
have exV : ∃ x, V x := ⟨False, Or.inl rfl⟩
```

U、Vが空集合でない言及。Trueは'True' is a proposition and has only an introduction rule として扱われている。恒真な命題であると考えれば良い。 Falseも同様。これらのような命題は、それぞれU,Vに定められた命題を満たすから、集合としてのU,Vに属している。

```
let u : Prop := choose exU
let v : Prop := choose exV
```

前述のexUとexVにより、U,Vへの選択公理の適用が許される。chooseはNonmenptyと別の名称の命題を用いる場合の選択関数。これによりU,Vの要素u,vを選んでいる。

```
have u_def : U u := choose_spec exU
have v_def : V v := choose_spec exV
```

選択公理によって選ばれたu,vがそれぞれU,Vに属することをchoose_specにより述べる。通常の証明における$\exists u, u\in U$の$u\in U$に当たる。

```
have not_uv_or_p : u ≠ v ∨ p :=
match u_def, v_def with
```

まず、「u≠v、またはpである」ことを示そう。uとvの定義から、u,vの値でパターン分けしている。

```
| Or.inr h, _ => Or.inr h
| _, Or.inr h => Or.inr h

| Or.inl hut, Or.inl hvf =>
  have hne : u ≠ v := by simp [hvf, hut, true_ne_false]
  Or.inl hne
```

一番上の行で左辺のOr.inrはu_defより、uが命題Uを満たす、i.e. u=True ∨ pが成り立つことを用いる。Or.inrはこれの右側、つまりpが成立する場合を指す。この場合、pの成立がすでに言われているわけだからu≠v ∨ p は満たされるので、 vについて特にパターン分けは不要。 _ で適当に指定させている。（この場合はv_def、つまりvが命題Vを満たすことをそのまま使っている。）右辺のOr.inrは証明事項 u≠v ∨ p の右を証明する、と宣言している。

```
have p_implies_uv : p → u = v :=
```

p → u=v をp:Prop から u=v:Propへの関数型として捉える。 p → u=v型の項の存在を示す↔ p → u=vを示す
前述の通り、Prop型の項を型に持つ項はその命題の証明であるから、 Lean上ではp型の項をu=v型の項に写すことが、pの真からu=vの真を得ることを示すのと同値になる。

```
fun hp =>
```

fun hp => で、p型の項hpをどのようにしてu=v型の項に写すかを言う。すなわち、u=vを示す。
U=Vならば、選択関数でUとVを写した先は等しい。i.e.u=v であるから、まずU=Vを示す。

```
have hpred : U = V :=
funext fun x =>
  have hl : (x = True ∨ p) → (x = False ∨ p) :=
    fun _ => Or.inr hp
  have hr : (x = False ∨ p) → (x = True ∨ p) :=
    fun _ => Or.inr hp
```

U,Vを関数型であたえているので、関数外延性を用いている。

```
  show (x = True ∨ p) = (x = False ∨ p) from
  propext (Iff.intro hl hr)
```

関数としてのU,Vの一致を示す。すなわち、命題xをU,Vで写した先の項 x = True ∨ p:Prop と x = False ∨ p の一致を言う。命題外延性（i.e.命題の同値）から示す。

```
have h₀ : ∀ exU exV, @choose _ U exU = @choose _ V exV := by
  rw [hpred]; intros; rfl
```

選択関数によって選ばれる元の一致を言う。（どの元を選んで空集合でないと示しても）、集合そのものが一致するなら選択関数で移す先は同じであることを言う。

```
show u = v from h₀ _ _
```

これによって、u,vが一致する。
これまでの証明でp→u=vであることと、u≠v ∨ pである事がわかった。
あとはu≠v と p で場合分けをする。

```
match not_uv_or_p with
| Or.inl hne => Or.inr (mt p_implies_uv hne)
| Or.inr h   => Or.inl h
```

これによって、選択公理から排中律が示された。