Agda による Zorn の補題の証明

河野真治
琉球大学工学部
Shinji KONO
Faculty of Engineering, University of the Ryukyus

Abstract

集合論は数学では古典的な基礎なので避けて通れない。Agda は Curry Howward 対応で証明を記述できる純関数型言語である。Agda 向きの集合論の公理を提案し、その有用性示すの例題として Zorn の補題の証明を行った。Zorn の補題はチコノフの定理などに使われる重要な定理であり、それを証明付きデータ構造を用いて Agda で証明する。直観主義論理での集合論について考察する。

Set theory is important because it is a classical foundation in mathematics. Agda is a pure functional language, which can describe proofs based on the Curry Howward correspondence. We use axiom suitable for Agda and show a proof of Zron lemma as an example. The proof methods of set theory in Agda are discussed.

1 Agda での集合論

Agda[5] はプログラミング言語なので、数学を証明付きのデータ構造として構成していく。集合論もこの方法で構築できる。集合論が使えるようになると、プログラムの証明などで、高次の無限大が使えるので、並列実行なのでかっこよいことができるかも知れない。

ZF 集合論 [4] は巧妙に作られた公理系を持つが、古典論理に基づいており、関数を中心とした直観主義論理とは合わないところがある。これを順序数定義可能集合を使って、Agda に合うようにする。この方法は Topos を用いる圏論的な集合論 [6] とは異なる。集合論などの基本的な数学を証明支援系で記述する作業は活発に行われている [2]。この論文はそのアプローチの一つである。

Agda の証明は論理式と型の対応である Curry Howard 対応に基づいてあり、当然限界がある。再帰や単一化では、停止性や値の実在性 (instaciation) による複雑な制限がある。Agda が正しい型だと判断する能力の限界も存在する。直観主義論理なので排中律の証明はない。証明できないものは postulate を使って仮定してよい。これは入力とするのと同じである。排中律あるいは選択公理を仮定しても Agda の証明には問題ない。それは非構成的な古典論理での証明になる。つまり、Agda は古典論理を包含していると考えてよい。

ここでは、実際にこの集合論が使えることを示すのと、比較的難しく、初学者の躓きやすい Zorn の補題を理解することを目標に Agda で Zorn の補題の証明を行う。

まず、自然数の集合の扱いを考えることから始めよう。

2 Agda での自然数の集合

Agda は型変数を持つ純関数型言語で、Curry Howard 対応により推論と証明を行う。

たとえば、**record** は Agda の構造体であり、複数の要素を持つ直積に相当する。これにより ∧ を型として表現できる。field が要素を表す。

```
record _∧_ {n m : Level} (A : Set n) ( B : Set m )
   : Set (n ⊔ m) where
  field
    proj1 : A
    proj2 : B

p = record { proj1 = 1 ; proj2 2 }
```

ここで、p は 1 と 2 のペアになる。proj1 p で 1 を取り出すことができる。これは、Haskell の普通のプログラムだと見ても良い。p は単なるデータ構造である。ただし、Set には Level が付いている。

この言語の中で集合論を考える。まず、自然数の集合を考えよう。これは素朴集合論に相当する。

```
record NSet : Set (suc zero) where
  field
    def : ℕ → Set
```

ここで **def : ℕ → Set** は、自然数 ℕ を含む論理式を表す。NSet の Level は ℕ より一つ上 **(suc zero)** あることに注意しよう。ここでの **Set** は、集合論の Set ではなくて、Agda の型、つまり、任意の (ある固定された Level 、この場合は Level 0) 論理式である。それに **def** という名前が付けられていて、**NSet** は、それを持っている。例えば、

```
eqa1 : NSet
eqa1 = record { def = λ x →  x * x + 6 ≡ 5 * x   }
```

これは、**{ x ∈ ℕ | x^2 + 6 = 5 x }** という集合に対応する。型 **Set** に式をそのまま書くことができる。この式にはこの式を評価する環境での変数を任意に使ってよい。**x** だけしか使えないわけではない。正確には Set と同じレベルの式を書く必要がある。NSet は一つレベルが上なので、そこには書けない。つまり、同じ record を再帰的に使うことはできない (工夫すれば書けることもある)。

これは NSet の intro 推論規則に相当する。つまり、record の constructor が推論規則になる。型 NSet を作ることが NSet を証明することになっている。

```
_∋_ : NSet → ℕ → Set
S ∋ x = def S x
```

と定義すると、

```
eq1∋2 : eqa1 ∋ 2
eq1∋2 = refl

eq1∋3 : eqa1 ∋ 3
eq1∋3 = refl
```

などが成立する。**def** は NSet の elim 推論規則に相当する。NSet を式から削除して NSet の要素の論理式に置き換える推論規則である。この場合、取り出されているのは型 (Set) であり、取り出した型/論理式を証明している。

ここで、refl は、

```
data _≡_ {A : Set } : A → A → Set where
  refl :  {x : A} → x ≡ x
```

と定義されている Agda の等式を生成する constructor である。ここで、x は評価されてから λ 項として等しいかどうかが調べられる。これは単一化なので、実際の計算はかなり複雑になる。等式は refl の他に対称律 sym 推移律 trans があり、refl を使って証明される。

ここで、**{A : Set }** は省略可能な変数を表している。Agda は依存型と呼ばれるが、型その物を first class つまり、値として用いることができる。Haskell[7] と異なり、使える型の範囲がほぼ無制限になっている。推論できる型を省略することにより人にとって見やすい証明になる。

これは等式の Agda での証明の例になっている。この集合は

```
eqa2 : NSet
eqa2 = record { def = λ x →  (x ≡ 2) ∨ ( x ≡ 3 )   }
```

と定義することもできる。ここで ∨ は

```
data  _∨_  {n m : Level} (A  : Set n) ( B : Set m ) : Set (n ⊔ m) where
  case1 : A → A ∨ B
  case2 : B → A ∨ B
```

である。これは Sum type と呼ばれる。例えば List は **[]** と **_::_** の Sum type である。**case1** は constructor であり、これを使って ∨ な型/論理式を作れる。Sum type はパターンマッチで取り出すことができる (elim 規則)。取り出すと、**A** または **B** が新しい変数に入っている値として取れてくる。もし、それが命題ならば、取れてくるのは証明である。排他的論理和は Sum type の一つして証明される。

集合は、以下のように等号を定義できる。

```
record _==_  ( a b :  NSet  ) : Set  where
  field
    eq→ : ∀ { x : ℕ } → def a x → def b x
    eq← : ∀ { x : ℕ } → def b x → def a x
```

これは外延性による集合の等しさの定義になっている。Agda では **(){}** 以外の連続した (空白が入らない)Unicode を含む単語は単なる名前として扱われる。つまい **eq←** は式ではなく名前である。

⊆ を以下のように定義することができて、片方は簡単に証明できる。

```
_⊆_ : (a b : NSet ) → Set
_⊆_ a b  = ∀ {x : ℕ} → def a x → def b x

eq2⊆eq1 : eqa2 ⊆ eqa1
eq2⊆eq1 {2} (case1 refl) = refl
eq2⊆eq1 {3} (case2 refl) = refl
```

ここでは **(case1 refl)** がパターンであり、その値は **def a x** という命題である。この命題をさらに **refl** というパターンで受けている。これで等式の受け方、つまり等式の使い方がわかる。**refl** あるいは **case1 x** をパターンに置けるかどうかは、そこでの値が十分にそろっているか (instanciation されているか) による。いつでもできるわけではない。逆方向の証明は難しい。二次方程式を解かないといけないので。例えば、ある自然数以下であることを証明して、帰納法で示してやれば良い。一般的に証明可能なわけではない。証明可能でないと示せれば否定になる。

```
data ⊥ : Set where

⊥-elim : {A : Set } → ⊥ →  A
⊥-elim ()

¬_ : Set → Set
¬ A = A → ⊥
```

consturctor のない data として否定を表すことができる。入力パターンとしてありえないことを示せれば、その関数の否定を証明できる。

例えば、NSet の空集合は

```
n∅ : NSet
n∅  = record { def = λ y → y < 0 }
```

と書ける。

```
¬n∅∋x : {x : ℕ } → ¬ ( n∅ ∋ x  )
¬n∅∋x ()
```

可能でない入力は () と書ける。あるいは同じことだが

```
¬n∅∋x' : {x : ℕ } → ¬ ( n∅ ∋ x  )
¬n∅∋x' {x} nx = ⊥-elim ( n1 nx ) where
  n1 : x < 0 → ⊥
  n1 ()
```

と明示しても良い。Agda での否定の証明はこのようにして行う。

これで論理の結合子、∧,∨,¬,≡ を導入することができた。量化記号は、関数の入力はすべて ∀ が入っていると考えてよい。∃ は、入力の record の field が対応する。例えば、ある集合 **(s : NSet)** が入力ならば、その集合の定義が存在して def で取り出せる。つまり、∃ を論理式の任意の場所に書くことはできない。これにより排中律を証明することはできないなどの直観主義論理の制約が生じる。

集合自体の集合を考えることもできる。

```
record NSetSet : Set (suc zero) where
  field
    ndef : NSet → Set

open NSetSet

record _=n=_ ( a b :  NSetSet  ) : Set (suc zero) where
  field
    eq→ : ∀ { x : NSet } → ndef a x → ndef b x
    eq← : ∀ { x : NSet } → ndef b x → ndef a x

eqa3 : NSetSet
eqa3 = record { ndef = λ x →  x == eqa2  }
```

NSet は自然数の部分集合なので実数、つまり非可算集合になっている。有限集合なら素朴集合になり、ZF の公理が成立する。**NSetSet** は実数の集合ということになる。

実数の性質を調べるのは数学の重要な目標の一つなので、これを統一的に扱いたいというのが集合論の目標になる。**NSetSetSET ...** では残念な感じである。

3 順序数定義可能集合 (Ordinal Definable Set)

集合は集まりなので、自然数と自然数の全体、自然数の部分集合の全体という風にできていると思いたい。とりあえず、これを順序数 Ordinal として受け入れることにする。あとで、これを Agda での公理系として record で定義する。これはもちろん高次の無限である。順序数には順序 **o<** があるとする。

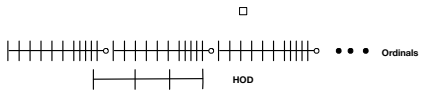

Figure 1: HOD and Ordinal

自然数の集合と同じように

```
record OD : Set (suc n ) where
  field
    def : (x : Ordinal  ) → Set n

record _==_ ( a b :  OD  ) : Set n where
  field
    eq→ : ∀ { x : Ordinal  } → def a x → def b x
    eq← : ∀ { x : Ordinal  } → def b x → def a x
```

と定義する。OD は順序数定義可能集合 (Ordinal Definable Set[1, 9]) である。OD は順序数上の部分集合になる。

順序数は「たくさん入っているもの」という直観で集合とみなせる。もし OD と順序数が対応すれば、さまざまなレベルの無限集合を均等に扱える集合論が得られると思われる。実際、**Set n** は Agda の論理式全体を含む高次の無限大なので、OD と $Ordinal$ が iso があると考えられる。集合の要素は、当然、順序数として小さいことが期待される。

```
& : OD  → Ordinal
* : Ordinal  → OD
c<→o< : {x y : OD }  → def y ( & x ) → & x o< & y
```

しかし、それはうまくいかない。順序数には自然数とおなじ一つ大きな順序数があり、それは順序数の順序 (**o<**) として大きい。逆の順序があれば矛盾となるのが期待される。これは自然数の公理の一部である。

実際、順序数全体を

```
data One {n : Level } : Set n where
  OneObj : One

Ords : OD
Ords = record { def = λ x → One }
```

として OD とすることができる (One は要素一つの data)。**c<→o<** があるとすると、以下の順序数の性質

```
<-osuc  : { x : Ordinal  } → x o< osuc x
o≤> : { x y : Ordinal  } → y o< osuc x  → x o< y → ⊥
```

から、

```
¬OD-order : ( & : OD  → Ordinal ) → ( * : Ordinal  → OD )
   → ( { x y : OD  } → def y ( & x ) → & x o< & y) → ⊥
¬OD-order & * c<→o< = o≤> <-osuc (c<→o< {Ords} OneObj )
```

なので、OD と Ordinal が iso ではありえない。これはラッセルの逆理の一つの形である。つまり、OD は単なる述語であって集合としては扱えない。集合ではない OD を class と呼んでも良い。

なので、集合論の集合を定義するには、もう少し工夫が必要である。

4　HOD 遺伝的順序数定義可能集合

集合論の本では **HOD = { x | TC x ⊆ OD \}** を使う[1]が、ここでは、単純に OD の要素に順序数的な最大がある

[1] x は V、つまり、順序数と平行に作られた ZF の天下りなモデルの要素である。x は集合なので、その要素、さらに、その要素の要素が OD なものを集めたものを HOD と定義している。通常の集合論では ZF の公理と V の存在を仮定して始めるので、こうなっている。

ことを要求する。

```
record HOD : Set (suc n) where
  field
    od : OD
    odmax : Ordinal
    <odmax : {y : Ordinal} → def od y → y o< odmax
```

歴史的な理由で HOD を使うが、Bounded OD の方がより相応しい名前になる。これなら Ordinal との iso を持って良い。その根拠はゲーデル数、つまり、扱うものはメモリに制限のない Agda のデータ構造しか型と値にならないということだが、それはここでは示さない。

```
record ODAxiom : Set (suc n) where
  field
    & : HOD  → Ordinal
    * : Ordinal  → HOD
    c<→o< : {x y : HOD }  → def (od y) ( & x ) → & x o< & y
    ⊆→o≤  : {y z : HOD }  → ({x : Ordinal} → def (od y) x → def (od z) x )
       → & y o< osuc (& z)
    *iso  : {x : HOD }     → * ( & x ) ≡ x
    &iso  : {x : Ordinal }  → & ( * x ) ≡ x
    ==→o≡ : {x y : HOD }  → (od x == od y) → x ≡ y
```

つまり、ZF の集合は、C 言語みたいなもので、集合は順序数のアドレスを持ち、**&, *** で相互に変換できる。外延的に等しい集合は同じアドレスを持つ。

```
odef : HOD → Ordinal → Set n
odef A x = def ( od A ) x

_∋_ : ( a x : HOD  ) → Set n
_∋_  a x  = odef a ( & x )
```

で、要素の存在を表す命題を取り出すことが出きる。これは命題なので証明する必要がある。実際に使うには **odef** の方が便利なので、それを使う。

(od x == od y) → x ≡ y は同じ要素を含むなら record として等しいことを要求するが、論理式を簡単にする機能がある。Agda では返す値が同じ関数なら **≡** として同じとみなす仮定 **functiona-extentinality** があるが、それと同じである。

例えば pair (x , y) は

```
record { def = λ t → (t ≡ & x ) ∨ ( t ≡ & y ) } ; odmax = omax (& x)  (& y)
```

と書けるが、これは x と y の max が要素の最大となる。なので、HOD になる。もし、

```
& (x , x) ≡ osuc (& x)
```

なら、**c<→o<** は **⊆→o≤** から導出される。例えば Russel Paradox がないことは、以下のように証明される。

```
∅6 : { x : HOD  } → ¬ ( x ∋ x )
∅6  {x} x∋x = o<¬≡ refl ( c<→o<  {x} {x} x∋x )
```

ここで、o<¬≡ は

```
o<¬≡ :  { ox oy : Ordinal } → ox ≡ oy  → ox o< oy  → ⊥
```

である。

これだと、HOD の Power Set が HOD であることは保証されないので、別に Power Set の大きさを制限する公理を足す必要がある。これには自由度があり、連続体仮説に対応する。これを付け加えることで、正則公理と選択公理以外の ZF の公理を証明できる。ただし、排中律が成立しないので、修正する必要がある。の集合論の本では、それを考慮した定義が採用されている。ここでは、ODAxiom で十分なので、Power Set は導入しない。

ZF の公理よりも、これらを使う方が Agda に向いている。これを使って Zorn の補題を示すことにする。

5　順序数の公理

順序数は順序数の存在物と、その存在物間の関係を別に記述するのが Agda 流であるらしい。次の順序数 osuc (直後順序数) と順序数の順序があるのは既に述べた。順序数の公理化は竹内のもの [8] あるが、これはもっと簡単なものである。

```
record Ordinals {n : Level} : Set (suc (suc n)) where
  field
    Ordinal : Set n
    o∅ : Ordinal
    osuc : Ordinal → Ordinal
    _o<_ : Ordinal → Ordinal → Set n
    isOrdinal : IsOrdinals Ordinal o∅ osuc _o<_ next
```

順序数には直前の順序数があるものとそうでないものがある。あるなら、次の record がある。

```
record Oprev {n : Level} (ord : Set n)
     (osuc :  ord → ord ) (x : ord ) : Set (suc n) where
   field
    oprev : ord
    oprev=x : osuc oprev ≡ x
```

順序数の順序を調べる Trichotomous と 直前があるかどうかを判定する述語が必要になる。

```
record IsOrdinals {n : Level} (ord : Set n)  (o∅ : ord ) (osuc : ord → ord )
   (_o<_ : ord → ord → Set n) (next : ord → ord ) : Set (suc (suc n)) where
  field
   ordtrans : {x y z : ord }  → x o< y → y o< z → x o< z
   trio<    : Trichotomous {n} _≡_  _o<_
   ¬x<0     : { x : ord  } → ¬ ( x o< o∅  )
   <-osuc   : { x : ord  } → x o< osuc x
   osuc-≡<  : { a x : ord  } → x o< osuc a  →  (x ≡ a ) ∨ (x o< a)
   Oprev-p  : ( x : ord  ) → Dec ( Oprev ord osuc x )
   o<-irr   : { x y : ord } → { lt lt1 : x o< y } → lt ≡ lt1
   TransFinite : { ψ : ord  → Set (suc n) }
      → ( (x : ord)  → ( (y : ord  ) → y o< x → ψ y ) → ψ x )
      →  ∀ (x : ord)  → ψ x
```

さらに、超限帰納法 TransFinite を要求する。これらの公理は、自然数の二次元配列で実現できる。つまり、モデルがあるので、順序数は無矛盾であることがわかる。HOD は可算順序数上で無矛盾であることが期待されるが、HOD と Ordnal の iso は簡単には示せない。詳細なコードは [3] にある。

Power Set を含むには Ordinals に無限の階層が必要になる。これは順序数の公理だが、Zorn の補題の証明には使わない。**o<-irr** は技術的な理由で必要であり、可算順序数なら証明できる。TransFinite は通常の超限帰納法だが、可算順序数では単なる帰納法であり、証明できる。つまり、この公理は、可算モデルを持つ。

つまり、Agda での HOD な集合論は可算モデルで議論していると思って良い。一方で順序数が非可算、つまり、**ℕ → Bool** を含むとしても良い。ただし、それで矛盾がでないかどうかは証明していない。

前述の **o<¬≡** は以下のように証明される。

```
o<¬≡ :  { ox oy : Ordinal } → ox ≡ oy  → ox o< oy  → ⊥
o<¬≡ {ox} {oy} eq lt with trio< ox oy
o<¬≡ eq lt | tri< a ¬b ¬c = ¬b eq
o<¬≡ eq lt | tri≈ ¬a b ¬c = ¬a lt
o<¬≡ eq lt | tri> ¬a ¬b c = ¬b eq
```

with 文は、Agda でのパターンマッチの構文である。| で場合分けを行う。Trichotomous は **tri<** の三つのパターンを返す。**¬b** は **¬ (ox ≡ oy)** の証明を含んでいる変数、つまり、**(ox ≡ oy) → ⊥** なので、それを使って ⊥ を返せば良い。

Trichotomous が通常の if 文と異なるのは、値だけではなく、その場合分けが成立するための証明が返ってくるところである。証明が値として得られる。可算順序数を構成する時には、あるいは一般的に Trichotomous を作る時には、その証明を作る必要がある。

6 選択公理

排中律と同等。Agda では仮定して使う。

OD には順序はあるので、その判定ができれば選択関数が作れる。判定には排中律を使う。

選択公理があれば、排中律を証明できる。なので、Agda での HOD な集合論では排中律と選択公理は同値である。

さらに正則公理が選択公理と同値になる。正則公理は自身と交わらない要素を取ってこれない要素がある、つまり、∋ で遡れない要素があることを要求するが、直観主義論理では空でない集合から要素を取ってくる関数を用意する必要がある。これは選択公理そのものになる。

```
minimal : (x : HOD  ) → ¬ (x =h= od∅ )→ HOD
x∋minimal : (x : HOD  ) → ( ne : ¬ (x =h= od∅ ) )
  → odef x ( & ( minimal x ne ) )
minimal-1 : (x : HOD  ) → ( ne : ¬ (x =h= od∅ ) ) → (y : HOD )
  → ¬ ( odef (minimal x ne) (& y)) ∧ (odef x (&  y) )
```

選択公理を要求しない直観主義論理上の正則公理は以下の形式を用いる。

```
ε -induction : { ψ : ZFSet → Set (suc m Level.⊔ n)}
      → ( {x : ZFSet } → ({ y : ZFSet } →  x ∋ y → ψ y ) → ψ x )
      → (x : ZFSet ) → ψ x
```

これは、順序数の TransFinite から示せるので公理ではなく定理となる。

正則公理が古典論理では選択公理と同値にならないのは、minimal の関数を明示しないで「あるとすれば」で正則公理を記述できるからだが、非構成的な選択と構成的な選択を区別しているからでもある。古典論理による集合論は、このような余計な区別が導入されていると見なせる。

Agda では排中律は使ってはいけない、あるいは、禁止されているわけではなくて、明示的に仮定して使うだけである。つまり、直観主義論理でできない数学があるわけではない。ただ、非構成的な存在物を明示しているだけである。

7 Zorn の補題とその重要性

Zorn の補題は集合論の初期でててきて、チコノフの定理の証明に使われる。まったく初等的ではないので、初学者がつまずく部分でもある。定理は「半順序集合 A で、全順序部分集合が上界を持つなら、A は極大元を持つ」というように記述される。これらの用語を正確に理解する必要がある。この定理は、部分の性質だけから、A 全体の性質を導いているところに特色がある。自然数の帰納法と同じ。これは当然、高次の無限集合でも成立する。また、この定理は選択公理と同値でもある。つまり選択公理/排他律を用いて証明され、これから選択公理を証明できる。

この定理の順序は、集合の自然な順序 ⊆ を使うこともできて、その場合は極大部分集合の存在を示すのに使える。たとえば、極大フィルターである。極大フィルターを使って、選択公理の成立しない集合論のモデルを作ることができるので、その意味でも Zorn の補題は重要である。

8 極大元、上界、全順序集合、半順序集合

HOD での Zorn の補題は以下のように書ける。ここでは順序数とは別の集合の順序 < を使う。これは外から与えられる。

```
Zorn-lemma : { A : HOD }
  → o∅ o< & A
  → ( ( B : HOD) → (B⊆A : B ⊆ A) → IsTotalOrderSet B → SUP A B   )
  → Maximal A
```

o∅ o< & A は A が空でないことを表している。IsTotalOrderSet は、順序 _<_ が部分集合の中で判定できることを表している。

```
IsTotalOrderSet : ( A : HOD ) → Set (Level.suc n)
IsTotalOrderSet A = {a b : HOD} → odef A (& a) → odef A (& b)
  → (a < b) ∨ (a ≡ b) ∨ (b < a )
```

正確には Agda の Trichotomous を使うので、もう少し複雑である。

上界 SUP と極大 Maximal は微妙に違う。上界は、部分集合 B の要素 x が (B の要素とは限らない A の要素である) sup よりも小さい順序が「ある」ことを主張する。

```
record SUP ( A B : HOD )  : Set (Level.suc n) where
  field
    sup : HOD
    as : A ∋ sup
    x≤sup : {x : HOD} → B ∋ x → (x ≡ sup ) ∨ (x < sup )
```

極大では A の中で, maximal よりも順序的に大きなもの x が A に「ない」ことを主張する。そういう maximal があると言っている。

```
record Maximal ( A : HOD ) : Set (Level.suc n) where
  field
    maximal : HOD
    as : A ∋ maximal
    ¬maximal<x : {x : HOD} → A ∋ x  → ¬ maximal < x
```

例えば、実数の集合 **{x | x < 0 }** を考えよう。これには極大元はない。これは、$x < 0$ 自体が、全順序部分集合なのに、上界が存在しない。なので、Zorn の補題は成立しない。**{x | x < ≤ 0 }** ならば成立する。

9 証明の方針

さまざまな書籍に証明が載っている。手抜きの本だと証明してない場合がある。それだけ厄介ならしい。基本的は A の要素の順序列を作り、それが単調増加な列があり、その各要素の順序数のアドレスが A 以下なことから、列が停止することを示し、さらに、極大元がない場合には無限単調列があることを示して矛盾を導く。あるいはこの列の濃度が A よりも大きなことを示す。これで納得するなら証明を調べる必要はない。

ここで、列は無限な可能性があるので、リストなどを使うことはできない。そこで、順序数から順序数への関数を使う。

```
f : Ordinal → Ordinal
```

まず、<-単調と ≤-単調な関数の性質を記述する。

```
≤-monotonic-f : (A : HOD) → ( Ordinal → Ordinal )
  → Set (Level.suc n)
≤-monotonic-f A f = (x : Ordinal ) → odef A x
   → ( * x ≤ * (f x) ) ∧  odef A (f x )

<-monotonic-f : (A : HOD) → ( Ordinal → Ordinal ) → Set n
<-monotonic-f A f = (x : Ordinal ) → odef A x
   → ( * x < * (f x) ) ∧  odef A (f x )
```

関数 f はそういう性質のものを仮定する。<-monotonic-f な関数が存在することを極大元がないことから導ける。

```
cf : ¬ Maximal A → Ordinal → Ordinal

is-cf : (nmx : ¬ Maximal A ) → {x : Ordinal}
   → odef A x → odef A (cf nmx x) ∧ ( * x < * (cf nmx x) )

cf-is-<-monotonic : (nmx : ¬ Maximal A ) → (x : Ordinal)
   →  odef A x → ( * x < * (cf nmx x) ) ∧  odef A (cf nmx x )

cf-is-<-monotonic nmx x ax = record { proj1 =  proj2 (is-cf nmx ax )
  ; proj2 =  proj1 (is-cf nmx ax )
```

cf と is-cf の証明は選択公理からでる。

これが止まることを示せば矛盾になる。そのために、最小上界 MinSUP と、最大鎖 chain を作ることにする。

最大鎖 chain は、自身の要素 a よりも大きい A の要素 b で、chain の f 閉包か、または、chain の上界であるものは全部含んでいるとする。

```
zf-is-max :  {a b : Ordinal } → (ca : odef chain a ) → b o< (& A)
    → (ab : odef A b )
    →  HasPrev A chain f b  ∨  ( { y : Ordinal }
    → odef chain y → (y ≡ b ) ∨ (y << b ) )
    → * a < * b  → odef chain b
```

最大鎖は f では増加しないはず。

```
fixpoint :  ( f : Ordinal → Ordinal ) → (mf : ≤-monotonic-f A f )
     → (sp1 : MinSUP A .chain )
     → f (MinSUP.sup sp1)  ≡ MinSUP.sup sp1
```

ここで、MinSUP は SUP に minsup 付け加えて最小性を要求してる。

```
record MinSUP ( A B : HOD ) : Set n where
  field
    sup : Ordinal
    asm : odef A sup
    x≤sup : {x : Ordinal } → odef B x → (x ≡ sup ) ∨ (x << sup )
    minsup : { sup1 : Ordinal } → odef A sup1
      →  ( {x : Ordinal  } → odef B x → (x ≡ sup1 ) ∨ (x << sup1 ))
      → sup o≤ sup1
```

補題の仮定から SUP があれば、それから MINSUP を超限帰納法と 排中律から導出することができる。

```
minsupP :  ( B : HOD) → (B⊆A : B ⊆' A) → IsTotalOrderSet B → MinSUP A B
```

ただし、fixpoint の証明は、最大性と、<-単調性に依存する。なので、ここで記述したように綺麗に分離すること派できない。

10 f 閉包の性質と上界関数

A が可算無限なら、f の繰り返し適用で話は終わるが、そうはいかない。そこで、f の閉包と上界を考える。

```
data FClosure (A : HOD) (f : Ordinal → Ordinal ) (s : Ordinal)
     : Ordinal → Set n where
   init : {s1 : Ordinal } → odef A s → s ≡ s1  → FClosure A f s s1
   fsuc : (x : Ordinal) ( p :  FClosure A f s x ) → FClosure A f s (f x)
```

これは自然数と同じ構造をしている。二つの constructor があり、init が zero で、fsuc が suc に相当する。ここで、

```
init : {s1 : Ordinal } → odef A s → s → FClosure A f s s
```

と書いても良いのだが、s ≡ s1 と書いておくと、ここを ? として、プログラミング技術的に便利になる。

f が ≤-単調ならば、順序を決定することができる。

```
fcn-cmp : {A : HOD} (s : Ordinal) { x y : Ordinal }
  (f : Ordinal → Ordinal) (mf : ≤-monotonic-f A f)
   → (cx : FClosure A f s x) → (cy : FClosure A f s y )
   → Tri (* x < * y) (* x ≡ * y) (* y < * x )
```

この証明は、f が途中で止まる可能性があるので、少し工夫がいる。ただ、最初から <-単調を仮定する方が楽かもしれない。

f の閉包の値は、順序数の (高次無限な) 直線上を飛び回ることになる。

f の domain 値域は非可算な可能性があり、閉包では取りきれない。閉包の上界よりも大きな値を取る f の値がある可能性がある。

それを与える関数 supf を考える。

```
order : {x y w : Ordinal } → x o< y → FClosure A f (supf x) w
  → w <= supf y
```

supf は初期値 y ∈ A から、f に依存して作っていく。

supf が満たす性質はいろいろあるので、それを ZChain record で定義する。

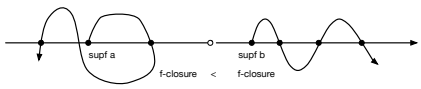

Figure 2: Closure of f and supf

11 supf が定義する集合列

supf は単調増加な整列された (全順序を持つ) 集合列 ⊆ A を定義する。

```
data UChain  { A : HOD } { f : Ordinal → Ordinal }
     {supf : Ordinal → Ordinal} (x : Ordinal) : (z : Ordinal) → Set n where
   ch-is-sup  : (u : Ordinal) {z : Ordinal }  (u<x : u o< x)
     (supu=u : supf u ≡ u) ( fc : FClosure A f (supf u) z ) → UChain x z

UnionCF : ( A : HOD )  ( f : Ordinal → Ordinal )  ( supf : Ordinal → Ordinal )
  ( x : Ordinal ) → HOD
  UnionCF A f supf x
     = record { od = record { def =
λ z → odef A z ∧ UChain {A} {f} {supf} x z } ;
       odmax = & A ; <odmax = λ {y} sy → ∈∧P→o< sy }
```

∈∧P→o< sy は、odef A z なので、odmax が自明に **& A** なことを証明する入項である。

12 ZChain と 上界関数 supf

supf の条件を recofrd ZChain で記述して、それを超限帰納法で作ってやればよい。

```
record ZChain ( A : HOD )  ( f : Ordinal → Ordinal )
   (mf : ≤-monotonic-f A f)
   {y : Ordinal} (ay : odef A y)  ( z : Ordinal ) : Set (Level.suc n) where
 field
      supf :  Ordinal → Ordinal
```

で、以下は基本的な性質になる。

```
asupf :  {x : Ordinal } → odef A (supf x)
supf-mono : {x y : Ordinal } → x o≤  y → supf x o≤ supf y
supfmax : {x : Ordinal } → z o< x → supf x ≡ supf z
supf0 : supf o∅ ≡ y
```

まだわかってないところは、分かっているところにあわせる。これで、変更ないときには、supf をそのまま再利用できる。

```
order : {x y w : Ordinal } → x o< y → FClosure A f (supf x) w
  → w <= supf y
sup=u : {b : Ordinal} → (ab : odef A b) → b o≤ z
  → IsSUP A (UnionCF A f supf b) ab
   ∧ (¬ HasPrev A (UnionCF A f supf b) f b )
  → supf b ≡ b
```

これは UChain で既に要求していた。

```
minsup : {x : Ordinal } → x o≤ z  → MinSUP A (UnionCF A f supf x)
supf-is-minsup : {x : Ordinal } → (x≤z : x o≤ z)
  → supf x ≡ MinSUP.sup ( minsup x≤z )
```

supf は最小上界である必要がある。さらに以下が最大性の根拠になる。

```
cfcs : (mf< : <-monotonic-f A f)
   {a b w : Ordinal } → a o< b → b o≤ z → supf a o< b
     → FClosure A f (supf a) w → odef (UnionCF A f supf b) w
```

これから、最大性

```
zf-is-max :  {a b : Ordinal } → (ca : odef chain a ) → b o< (& A)
 → (ab : odef A b )
 →  HasPrev A chain f b  ∨  ( { y : Ordinal }
 → odef (UnionCF A f (ZChain.supf zc) b) y → (y ≡ b ) ∨ (y << b ) )
 → * a < * b  → odef chain b
```

を導出できる。

13　ZChain の超限帰納法による構成

超限帰納法は、x 以下の ZChain があることを仮定して、ZChian x を作ればよい。

```
ind : ( f : Ordinal → Ordinal ) → (mf : ≤-monotonic-f A f )
  {y : Ordinal} (ay : odef A y)
   → (x : Ordinal) → ((z : Ordinal) → z o< x → ZChain A f mf ay z)
   → ZChain A f mf ay x
```

x は極限順序数か直後順序数なので、それで場合をわける。

14　直後順序数の場合

supf x は、直前の ZChain には入ってない。これは、直前の ZChain の最小上界のはずである。spuf x の f 閉包を sp1 とする。sp1 が x 以下か、それより大きい場合で、さらに場合分けする。

14.1　sp1 ≤ x

sp1 ≤ x なら、sp1 は今の ZChain に含まれる。したがって、supf を拡張する必要がある。前の supf を supf0 として、以下のようにする。

```
supf1 : Ordinal → Ordinal
supf1 z with trio< z px
... | tri< a ¬b ¬c = supf0 z
... | tri≈ ¬a b ¬c = supf0 z
... | tri> ¬a ¬b c = sp1
```

ここで、supf0 は x 直前の px での supf である。

最大性

```
cfcs : (mf< : <-monotonic-f A f) {a b w : Ordinal }
   → a o< b → b o≤ x →  supf1 a o< b
   → FClosure A f (supf1 a) w
   → odef (UnionCF A f supf1 b) w
```

を示すには、a と supf a と px の関係で場合分けすればよい。

14.2　x < sp1

sp1 が x より大きいなら、supf を拡張する必要はない。しかし、supf px は新しい鎖に含まれるので、ZChain record は、それに対応して証明する必要がある。

最大性

```
cfcs : (mf< : <-monotonic-f A f) {a b w : Ordinal }
   → a o< b → b o≤ x →  supf1 a o< b
   → FClosure A f (supf1 a) w
   → odef (UnionCF A f supf1 b) w
```

を示すには、a と supf a と px の関係で場合分けすればよい。

15　極限順序数の場合

この場合は、直前の鎖が存在しない。なので、x 以下の supf の和集合を取る。

x より小さい z があれば、osuc x は z が極限順序数なので、z よりも小さくなる。そこで、任意の z o< x に対して x 以下の supf を使って値を定義できる。

```
pzc : {z : Ordinal} → z o< x → ZChain A f mf ay z
pzc {z} z<x = prev z z<x

ysp =  MinSUP.sup (ysup f mf ay)

supfz : {z : Ordinal } → z o< x → Ordinal
supfz {z} z<x = ZChain.supf (pzc  (ob<x lim z<x)) z
```

x での値は以下の record を考えると、これを満たす A の要素全体は順序付けられるので最小上界を取れる。これを spu とする。

```
record IChain  (A : HOD)  ( f : Ordinal → Ordinal ) {x : Ordinal }
     (supfz : {z : Ordinal } → z o< x → Ordinal) (z : Ordinal ) : Set n where
  field
    i : Ordinal
    i<x : i o< x
    fc : FClosure A f (supfz i<x) z
```

ただし、各点での supfz はお互いに関係しないように思われる。ところが最小上界なので、結局は一致することを超限帰納法で示すことができる。

```
  supf-unique :  ( A : HOD )  ( f : Ordinal → Ordinal )  (mf : ≤-monotonic-
f A f)
        {y xa xb : Ordinal} → (ay : odef A y) →  (xa o≤ xb )
      → (za : ZChain A f mf ay xa ) (zb : ZChain A f mf ay xb )
      → {z : Ordinal } → z o≤ xa → ZChain.supf za z ≡ ZChain.supf zb z
```

spu を使って以下のように supf1 を定義する。

```
supf1 : Ordinal → Ordinal
supf1 z with trio< z x
... | tri< a ¬b ¬c = supfz a
... | tri≈ ¬a b ¬c = spu
... | tri> ¬a ¬b c = spu
```

最大性

```
cfcs : (mf< : <-monotonic-f A f) {a b w : Ordinal }
   → a o< b → b o≤ x →  supf1 a o< b
   → FClosure A f (supf1 a) w
   → odef (UnionCF A f supf1 b) w
```

を示すには、b ≡ x か b o< x かに注目する。b ≡ x なら a < x なので、前の鎖が使える。b o < x でも前の鎖が使える。

これで証明できた。詳細は github の Agda のコードを参照して欲しい。

16　めんどくささのポイント

f が作る列は ≤-単調でもあり、<-単調でもあるのだが、停止するなら ≤-単調 だと思われるがそれでは最大性を示せない。

supf は、順序数的に単調でもあり、< の順序的にも単調。しかし、その idempotent (supf (supf x) ≡ supf x) を示すには、<-単調 が必要になる。

直後順序数の場合分けが複雑。

もしかすると、< に関する超限帰納法を使うと、もっと簡単になるかも。

17　Agda によるめんどくささ

record の中に同じ record を入れることはできない。しかし集合は本質的に集合を含む。それを避けるには「集合の定義 def では、順序数ですべてを記述する」と良い。ある順序数が存在すると記述する場合には record / data を定義して使うことができる (UChain はそういうもの) が、その Level は n である必要がある。対応する順序数があれば、record の Level は n よりも大きくてよい。

<な順序は、集合に対して定義されており、順序数ではない。そこに複雑な集合 (例えば最小上界) を入れると、Agda の型検査でメモリ爆発を起こすことがある。この場合は、変換

```
≤to<= : {x y : Ordinal } → * x ≤ * y → x <= y
≤to<= (case1 eq) = case1 ( subst (λ j k → j ≡ k )
  &iso &iso  (cong (&) eq) )
≤to<= (case2 lt) = case2 lt
```

を入れると解決する。理由はわからない。Agda の bug かもしれない。

Agda では本来、証明は関数に閉じている。触らない部分は再評価する必要はなく、実際、.agdai というのが生成されて再評価を避けるようにできている。しかし、一つのファイルないではそうはならない。なので、こまめなファイル分割が必要になる。

18　OD 公理系によるめんどくささ

上のコードでもどうだが、集合と順序数の変換が頻繁に必要になる。そこにある順序数がどんな集合に対応するかは文脈からしかわからない。これは アセンブラのポインタが何を指しているかは、文脈からしかわからないのと同じである。つまり、集合論は C というよりはアセンブラレベルだとも言える。

排中律と選択公理は明示的に使用する必要がある。完全に避ける必要はないが、多用する意味はない。関数の入力は ∀ が付いているのと同じであり、∃ を使いたければ record を使うことになる。集合の定義で使いたい時には、そのたびに record を定義することになる。一般的な record を存在記号用に作ることもできるが、構文的にはいまいちな感じになる。

19　チコノフの定理の考察

コンパクト集合の直積はコンパクトという簡単な定理だが Zorn の補題を使う必要がある。コンパクトは有界閉集合の拡張だが、点列コンパクトというものもある。

1. 有界閉集合
2. 点列コンパクト (任意の無限部分列が集積点を持つ)
3. コンパクト (X の閉集合の任意の集合 F 有限交差性を満たせば ∩ **F** は空ではない)

これらの差は微妙だが、距離空間では消滅する。Zorn の補題は、鎖を作ることにより、対称ではないが距離的なものを提供している。

コンパクトの概念は、Zorn の補題と同じく、集合 A の任意の部分集合の性質と、集合 A 全体の性質を結びつけている。これらの差のいくつかは、正則公理と選択公理の差のようなものかも知れない。直観主義論理上の集合論は、決定性を持たない集合の順序をがあるので、これらの差が消滅する可能性はある。

一方で、これらの差を議論したければ、集合の順序付を与える部分関数上で議論することが考えられる。Zorn の補題は「任意の全順序部分集合」を仮定として要求するが、例えば可算の場合とか、可算被覆とかの仮定があると、精密な議論を行うことが可能になる。

20 付録 Agda での 順序数 の公理

```
record Oprev {n : Level} (ord : Set n) (osuc :  ord → ord )
    (x : ord ) : Set (suc n) where
 field
   oprev : ord
   oprev=x : osuc oprev ≡ x

record IsOrdinals {n : Level} (ord : Set n)  (o∅ : ord ) (osuc : ord → ord )
   (_o<_ : ord → ord → Set n) (next : ord → ord ) : Set (suc (suc n)) where
 field
   ordtrans : {x y z : ord }  → x o< y → y o< z → x o< z
   trio<    : Trichotomous {n} _≡_  _o<_
   ¬x<0     : { x : ord  } → ¬ ( x o< o∅  )
   <-osuc   : { x : ord  } → x o< osuc x
   osuc-≡<  : { a x : ord  } → x o< osuc a  →  (x ≡ a ) ∨ (x o< a)
   Oprev-p  : ( x : ord  ) → Dec ( Oprev ord osuc x )
   o<-irr   : { x y : ord } → { lt lt1 : x o< y } → lt ≡ lt1
   TransFinite : { ψ : ord  → Set (suc n) }
      → ( (x : ord)  → ( (y : ord  ) → y o< x → ψ y ) → ψ x )
      →  ∀ (x : ord)  → ψ x

record IsNext {n : Level } (ord : Set n)  (o∅ : ord ) (osuc : ord → ord )
    (_o<_ : ord → ord → Set n) (next : ord → ord ) : Set (suc (suc n)) where
 field
   x<nx :   { y : ord } → (y o< next y )
   osuc<nx : { x y : ord } → x o< next y → osuc x o< next y
   ¬nx<nx :  {x y : ord} → y o< x → x o< next y →  ¬ ((z : ord) → ¬ (x ≡ osuc z))

record Ordinals {n : Level} : Set (suc (suc n)) where
 field
   Ordinal : Set n
   o∅ : Ordinal
   osuc : Ordinal → Ordinal
   _o<_ : Ordinal → Ordinal → Set n
   next :  Ordinal → Ordinal
   isOrdinal : IsOrdinals Ordinal o∅ osuc _o<_ next
   isNext : IsNext Ordinal o∅ osuc _o<_ next
```

21 付録 Agda での ZF の公理

```
module zf where

record IsZF {n m : Level }
   (ZFSet : Set n)
   (_∋_ : ( A x : ZFSet  ) → Set m)
   (_≈_ : Rel ZFSet m)
   (∅  : ZFSet)
   (_,_ : ( A B : ZFSet  ) → ZFSet)
   (Union : ( A : ZFSet  ) → ZFSet)
   (Power : ( A : ZFSet  ) → ZFSet)
   (Select :  (X : ZFSet  ) → ( ψ : (x : ZFSet ) → Set m ) → ZFSet )
   (Replace : ZFSet → ( ZFSet → ZFSet ) → ZFSet )
   (infinite : ZFSet)
    : Set (suc (n ⊔ suc m)) where
 field
   isEquivalence : IsEquivalence {n} {m} {ZFSet} _≈_
   -- ∀ x ∀ y ∃ z ∀ t ( z ∋ t → t ≈ x ∨ t  ≈ y)
   pair→ : ( x y t : ZFSet  ) →  (x , y)  ∋ t  → ( t ≈ x ) ∨ ( t ≈ y )
   pair← : ( x y t : ZFSet  ) →  ( t ≈ x ) ∨ ( t ≈ y )  →  (x , y)  ∋ t
   -- ∀ x ∃ y ∀ z (z ∈ y ⇔ ∃ u ∈ x  ∧ (z ∈ u))
   union→ : ( X z u : ZFSet ) → ( X ∋ u ) ∧ (u ∋ z ) → Union X ∋ z
   union← : ( X z : ZFSet ) → (X∋z : Union X ∋ z )
     →  ¬  ( (u : ZFSet ) → ¬ ((X ∋  u) ∧ (u ∋ z )))
 _∈_ : ( A B : ZFSet  ) → Set m
 A ∈ B = B ∋ A
 _⊆_ : ( A B : ZFSet  ) → ∀{ x : ZFSet } →  Set m
 _⊆_ A B {x} = A ∋ x →  B ∋ x
 _∩_ : ( A B : ZFSet  ) → ZFSet
 A ∩ B = Select A (  λ x → ( A ∋ x ) ∧ ( B ∋ x ) )
 _∪_ : ( A B : ZFSet  ) → ZFSet
 A ∪ B = Union (A , B)
 field
   empty :  ∀( x : ZFSet  ) → ¬ ( ∅ ∋ x )
   -- power : ∀ X ∃ A ∀ t ( t ∈ A ↔ t ⊆ X ) )
   power→ : ∀( A t : ZFSet  ) → Power A ∋ t → ∀ {x}  →  t ∋ x
      → ¬ ¬ ( A ∋ x ) -- _⊆_ t A {x}
   power← : ∀( A t : ZFSet  ) → ( ∀ {x}  →  _⊆_ t A {x})  → Power A ∋ t
   -- extensionality : ∀ z ( z ∈ x ⇔ z ∈ y ) ⇒ ∀ w ( x ∈ w ⇔ y ∈ w )
   extensionality :  { A B w : ZFSet  } →
     ( (z : ZFSet) → ( A ∋ z ) ⇔ (B ∋ z)  ) → ( A ∈ w ⇔ B ∈ w )
   -- regularity without minimum
    ε -induction : { ψ : ZFSet → Set (suc m Level.⊔ n)}
     → ( {x : ZFSet } → ({ y : ZFSet } →  x ∋ y →  ψ y ) →  ψ x )
     → (x : ZFSet ) → ψ x
   -- infinity : ∃ A ( ∅ ∈ A ∧ ∀ x ∈ A ( x ∪ { x } ∈ A ) )
   infinity∅ :  ∅ ∈ infinite
   infinity :  ∀( x : ZFSet  ) → x ∈ infinite →  ( x ∪ ｛ x ｝ ) ∈ infinite
   selection : { ψ : ZFSet → Set m } → ∀ { X y : ZFSet  } →
     ( ( y ∈ X ) ∧ ψ y ) ⇔ (y ∈  Select X ψ )
   -- replacement : ∀ x ∀ y ∀ z ( ( ψ ( x , y ) ∧ ψ ( x , z ) ) → y = z )
   --    → ∀ X ∃ A ∀ y ( y ∈ A ↔ ∃ x ∈ X ψ ( x , y ) )
   replacement← : {ψ : ZFSet → ZFSet} → ∀ ( X x : ZFSet  )
     → x ∈ X → ψ x ∈  Replace X ψ
   replacement→ : {ψ : ZFSet → ZFSet} → ∀ ( X x : ZFSet  )
     →  ( lt : x ∈  Replace X ψ ) → ¬ ( ∀ (y : ZFSet)  →  ¬ ( x ≈ ψ y ) )


   -- ∀ X [ ∅ ∉ X → (∃ f : X → ⋃ X ) → ∀ A ∈ X ( f ( A ) ∈ A ) ]
   choice-func : (X : ZFSet ) → {x : ZFSet } → ¬ ( x ≈ ∅ ) → ( X ∋ x ) → ZFSet
   choice : (X : ZFSet  ) → {A : ZFSet }
     → ( X∋A : X ∋ A ) → (not : ¬ ( A ≈ ∅ )) → A ∋ choice-func X not X∋A

   -- mimimul and x∋minimal is an Axiom of choice
   minimal : (x : HOD  ) → ¬ (x =h= od∅ )→ HOD
   -- this should be ¬ (x =h= od∅ )→ ∃ ox → x ∋ Ord ox  ( minimum of x )
   x∋minimal : (x : HOD  ) → ( ne : ¬ (x =h= od∅ ) ) → odef x ( & ( minimal x ne ) )
   -- minimality (proved by  ε -induction with LEM)
   minimal-1 : (x : HOD  ) → ( ne : ¬ (x =h= od∅ ) ) → (y : HOD )
     → ¬ ( odef (minimal x ne) (& y)) ∧ (odef x (&  y) )
```

References

[1] Kurt Gödel. Remarks before the princeton bicentennial conference on problems in mathematics. In Solomon Feferman, John Dawson, and Stephen Kleene, editors, Kurt Gödel: Collected Works Vol. Ii, pages 150–153. Oxford University Press, 1946.

[2] Jose Grimm. Implementation of bourbaki's elements of mathematics in coq: Part one, theory of sets. Journal of Formalized Reasoning, 3(1):79–126, 2010.

[3] Shinji Kono. https://github.com/shinji-kono/zf-in-agda, 2019.

[4] Kenneth Kunen. Set theory - an introduction to independence proofs. page 1 – 325, Nov 2008.

[5] Ulf Norell. Dependently typed programming in agda. In Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[6] Gerhard Osius. Categorical set theory: A characterization of the category of sets. iJournal of Pure and Applied Algebra, (4):79–119, 1974.

[7] Bryan O'Sullivan, Don Stewart, and John Goerzen. Real world haskell, 2008.

[8] Gaisi Takeuti. A formalization of the theory of ordinal numbers. The Journal of Symbolic Logic, 30(3):295–317, 1965.

[9] 田中 尚夫. **公理的集合論**. 培風館, 1982.