Agda 上での ZF 集合論の構成

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

Abstract

Agda は省略可能な型変数を持つ純関数型言語であり、カーリーハワード対応に基づく証明支援系として使うことができる。数学的な命題は型として記述され、その証明は型を実現する λ 項として記述される。ここでは直観主義論理に適した非構成的な仮定を導入し、ZFC 集合論の公理を証明する。集合はデータ構造である順序数上の述語 (Ordinal definable) として導入する。

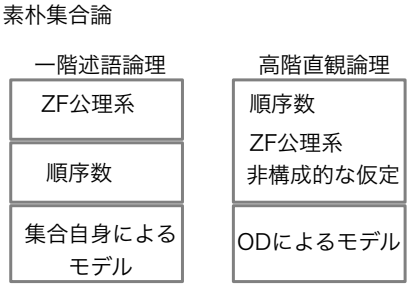

Figure 1: Set Theory and Agda

1　集合論と Agda

本論文では ZF 集合論の公理を仮定するのではなく、型付き関数型言語と (おなじことだが) 直観主義論理に基づく定理証明系である Agda[4] にいくつかの公理を導入し、ZF 集合論のモデルを構築する。

ZF 集合論の公理は Agda により、直観主義論理で記述される。まず自然数を無限集合に拡張した順序数 (Ordinal) の公理を record で定義し、その順序数上の方程式として Ordinal Definable (OD)[1] を Agda の record (直積に相当するデータ構造) として定義する。Agda で採用する最初の公理は、順序数と OD との一対一対応 (isomorphism) である。これにより、順序数方程式は対応する OD を解として持つことになり、OD は OD を含む集合になる。つまり「含む (∋)」が OD に対して定義される。次に、順序数と OD の対応で、OD の ∋ が定義する順序が順序数の順序に対応することを要求する。この時に逆は要求しない。つまり順序数の順序から OD の ∋ を導くことはできない。

順序数は公理的に定義されるが、具体的なデータ型として、その値を定義することも可能である。この場合、順序数は可算集合 (構成論理的に構築されるデータ構造) になる。集合 (OD) は順序数に順序同型なので、順序数のモデルにより集合のモデルが変わる (Fig. 1)。

歴史的には 90 年代に直観主義論理に基づく集合論としては構成的集合論 (Constructive Set Theory) が 80 年代に研究されている。それらは従来の集合論ではなく構成的な集合に対する理論となっている。直観主義論理は圏論的には Topos と呼ばれる。Topos は積と関数適用を持つ圏に Subobject classifier と呼ばれる真理値関数を導入したものである。この上で集合論を構築したのが圏論的集合論 (Categorical Set Theory)[5] である。推移律が成立する推移集合とその冪集合を生成する関手から構成される。Osius の論文ではモデル的な構成だが証明論的な構成も可能であることが言及されており、この構成は Topos を明示的には用いない証明論的な構成になっている。Topos は直観主義論理に対応するので、Topos に基づいた構成だと言っても良い。Coq で別な数学の基礎からの構成であるブルバキを証明する計画がフランスで実行されている。この論文 [2] では ZF の公理と Agda を用いて、同等のことを示したことになる。

一階述語論理では任意の関数や述語を含むメタな公理を記述できないので、公理図式や、ゲーデルベルナイによるエンコードなどが必要となる。あるいは $\psi(x_0, x_1, x_2, ...x_n)$ のような記述である。Agda による記述は具体的な高階関数による有限な記述になっている。また、モデルの構築は Agda によって行われるので、数学書に良くある省略は一切ない。証明に必要な詳細はすべて含まれている。

Agda の正しさと導入した公理、さらに非構成的な仮定に整合性があるかどうかは、ゲーデルの第二不完全性定理があるので、それを期待することはできない。この構築は ZF 集合論の別構成法を示しただけで、それ自体の無矛盾性を示してはいない。

2　さまざまな集合

ZF にはもちろんさまざまな集合が出てくるのだが、集合は階層的であり、同じ数学的構造物の集まりでも単純に集合と集合でないクラスにわけて考える必要がある。これは「自分自身を含まない集合」のような矛盾を避けるために導入されている。

Agda にも Set があり、階層化されているが、これは ZF の集合とは異なり、記号を値と型にわけた時の型に付いている記号を Set と呼んでいるだけである。ある型を値として使いたい場合は、その型は一つ上の Set となる。Agda の Set の階層は Level と呼ばれる自然数で識別される。

自然数は基本的な集合であり、ある数よりも小さい集合は有限集合となる。自然数全体は最初の無限集合となる。この無限集合を一つの集合と考えると、それの集まりをさらに考えることができる。これは人が持つ無限集合に対する直観であり、その直観を公理として記述することができる。これは自然の拡張の一つであり、順序数 (Ordinal) と呼ばれる。

順序数定義可能集合 (OD) は、順序数上の方程式であり、階層構造を持たない平坦な構造を持っている。すべての順序数が満たす方程式は λ **x → T** として表すことができる。これは順序数全体の (一つ上のレベルの) 集合に対応する。

対とは二つの集合から作る新しい集合である。空集合 ϕ から始めて、空集合を一つ含む集合を考えることができる{{ ϕ }}。その二つの対 verb+ ϕ , ϕ + 作る。これを繰り返して新しい集合をどんどん作れる。これを無限回繰り返した全部を含む集合 (Union) を一つの集合と考える。そして、また同じ操作を繰り返す。この操作により階層的な無限集合が作られる。これを V 呼ぶ.

自然数の部分集合全体、あるいは自然数から自然数への写像の集合を考えると、自然数よりも大きな無限集合を作ることができる。このように作り方を述語で指定する方法で集合を作ることもできる。空集合から始めて、空集合上のすべての述語から作られる集合を考える。その集合全体の Union を考える。その集合上で述語により構成できる集合を作る。それをすべて集める。この操作を繰り返して得られる階層的な集合を L という。(Fig.2)。

3　証明支援系 Agda

Agda はカーリーハワード対応、つまり、関数の型が論理式に対応し、型に対応する λ 項が証明に対応するという原理にもとづく証明支援系である。実際に Agda は単なる型付き関数型言語に強力な型推論、つまり、型の単一化を持つものに過ぎない。Agda は Haskell 上に実装されており、Haskell とほぼ同じ構文を持つ。

Agda は名前に:(colon) で型を対応させ、その型に対応した λ 項を=で対応させる。Haskell とのもっとも大きな構文的な差は、単語と空白の扱いである。Agda は Unicode を含む任意の文字の連続を単語として扱い、その区切り文字は空白と (){}のみである。つまり、=や:の前後には必ず空白が必要となる。Unicode を積極的に使っているので、λ や → を用いる。Agda は Polymorphism を持っていない。つまり、名前は型と独立に unique である必要がある。名前を再利用するには module を別にする必要がある。最後に、{}で定義された変数は省略可能である。省略された変数は型推論により自動的に呼び出した環境内の変数に結合される。

λ 項が証明になるには、型の整合性の他に停止性と充足性を満たす必要がある。停止性は再帰呼び出しがある場合に、それが再帰的な型をより小さい型に分解 (縮約) していることを Agda が理解する必要がある。充足性は、λ 項に表れる変数はすべて入力によって値を代入されている必要がある。これは λ 項に存在量化記号がないことに対応する。これらの問題は Agda の型推論機構が指摘し、すべてが整合すれば Agda 上での推論は正しいことになる。Agda の基本型はレベルの付いた Set である。レベルは自然数である。

```
function-from-A-to-B : {n m : Level} (A  : Set n) ( B : Set m )
   → Set (n ⊔ m)
function-from-A-to-B A B = A → B
```

Ordinal	順序数の公理を満たすもの。自然数の延長
V	一つ一つ増やす。無限回繰り返して、それを全部。さらにそれを繰り返して得られるもの
L	述語で定義されるものを集めるのを繰り返して作られるもの
OD	順序数上の方程式を満たす順序数の集合 階層化されていない
Agda の Set	型。値の種類を区別する記号。自身が値になると、一つ Level の高い型を持つ

Figure 2: さまざまな集合

function-from-A-to-B は単一の単語であり、型と値を持つ。二つのレベル **n m** の最大値が **n ⊔ m** である。Agda は Haskell と同じく indent が構文の一部なので改行する時に一段下げると同じレベルの構文ブロックとなる。これらは **Level** module で定義されている。**A → B** は Agda の基本的な型であり、レベルにしたがった **Set** になる。つまり、この定義は型を λ 項として定義している。型そのものを λ 項とすることができる、つまり、型を first order できる言語を依存型 (dependent type) と呼ぶことがある。

A → B は、また、implication でもあり論理式として見ることができる。つまり、**Set** は命題を表している。直観主義論理では命題は真偽を持つのではなく、証明を持つ命題を真とする。これは Hayting 代数とも呼ばれ、命題が真偽を必ず持つ一階述語論理と異なる点になっている。Agda のデータ型は、論理に対応するように導入されている。つまり、そのように型システムを定義するとカーリーハワード対応を実現できる。Agda では Haskell とは異なり、直積と排他和を別な構文で定義する。

直積は **record** で定義される。これは「かつ」conjunction に相当する。

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

_∧_ は Agda での二項演算子の定義である。_が引数の対応する場所を表す。ここでは空白は許されない。_∧_ で一つの単語になる。実際に用いる時には **A ∧ B** のように表れる。**record** は **where** 以降で定義された複数の field を持ち、それぞれの filed を名前と対応する型を持っている。この型はもちろん論理式でもある。つまり、**A ∧ B** は二つの論理式 **A, B** を結合した積である。**A ∧ B** は **A** と **B** のレベルの最大値を持つ。

Curry Howard 対応を持つ論理では、**record** のような論理式を構成する要素には必ず、導入 (intro) と除去 (elimination) の推論が対応する。これはプログラミング的には constructor と accessor に相当する。

∧ 除去は以下のようになる。(このように使うには **reocrd _∧_** を **open** する必要がある。

```
lemma1 : A ∧ B → A
lemma1 p = proj1 p

lemma2 : A ∧ B → B
lemma2 p = proj2 p
```

関数言語としては **proj1** は直積から最初の要素を取り出す関数であり、論理式としては、二つの証明の組から最初の証明を取り出す推論 **A ∧ B → A** になる。

∧ 導入は以下の構文を用いる。**record** の constructor を構文的に定義することもできる。**{}** で表された入力は値の定義の部分では省略されているのがわかる。

```
lemma3 : {n m : Level} {A  : Set n} { B : Set m } → A → B → A ∧ B
lemma3 a b = record { proj1 = a ; proj2 = b }
```

排他和は **data** で表されるデータ構造である。Haskell では、この構文を使って record も表している。

_∨_ は証明にいくつかの場合分けがあることを示している。つまり、**case1** と **case2** である。場合分けはどちからしか起きず、同時に起きることはないとする。

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

ここで、**case1** と **case2** は constructor であり、型 **A ∨ B** を返す関数の型を持つ必要がある。あるいは、場合分けされた証明に対応する型である。必要な入力を与えると証明を構成できる。この constructor は、もちろん、∨ 導入に対応する。

除去の場合は **A ∨ B** は入力に表れる。関数の入力のパターンとして constructor を記述する。ここでの**?**は

未決定の部分を表す。証明を完成させるには、すべての場合をつくす必要がある。

Agda では値を与えずに論理式を仮定 (postulate) することができる。Agda はインデントで構文的なブロックを表す。

```
postulate
  n : Level
  A : Set n
  B : Set n
  C : Set n
  lemma4 :  A ∨ B
  casea : A → C
  caseb : B → C
```

これらの仮定を用いた証明は以下のようになる。**lemma4** は二つの場合を持ち、それぞれから **C** を推論している。入力のパターンマッチが ∨ の除去規則になる。値として **case1 a** を使うと導入規則になる。

```
lemma5 :  C
lemma5 with lemma4
... | case1 a = casea a
... | case2 b = caseb b
```

仮定は module に対する入力と考えることもできる。module 内のすべての証明に隠れた入力があると見ることもできる。実際に module の入力として書いても良い。

矛盾 ⊥ は constructor のない **data** として表現することができる。

```
data  ⊥  {n  : Level} : Set n

⊥-elim : ∀ {w} {Whatever : Set w} → ⊥ → Whatever
⊥-elim ()
```

() は、それ自体で充足できない入力パターンを表す。それには **body**、つまり証明あるいは値は必要ない。これが ⊥ の除去規則であり、矛盾からはなんでも導出できるということに対応している。⊥ にはもちろん導入はない。

これらを用いて、対偶や二重否定の導入を証明できる。

```
¬_ : ∀ {ℓ} → Set ℓ → Set ℓ
¬ P = P → ⊥

contra-position : {n m : Level } {A : Set n} {B : Set m}
  → (A → B) → ¬ B → ¬ A
contra-position {n} {m} {A} {B}  f ¬b a = ¬b ( f a )

double-neg : {n  : Level } {A : Set n} → A → ¬ ¬ A
double-neg A notnot = notnot A
```

Agda では **¬ ¬ A → A** に対応する証明はない。これが直観主義論理あるいは構成的論理の特徴である。

さらに **⊥-elim** の使い方を示してく。矛盾からはなんでも導出できる、あるいは、可能な入力組合せが存在しないことを表す。

```
dont-or :   {n m : Level} {A  : Set n} { B : Set m }
  →  A ∨ B → ¬ A → B
dont-or {A} {B} (case1 a) ¬A = ⊥-elim ( ¬A a )
dont-or {A} {B} (case2 b) ¬A = b
```

4　ZF axiom in Agda

集合論の公理は一階述語論理で記述されている。これを Agda に合わせるには構文の他に存在記号 ∃ を除去する必要がある。一つの方法は、存在記号で示されたものを生成する関数を公理として要求することである。しかし、存在記号の場所によっては公理の意味を変えてしまうことがある。もう一つはドゥモルガン則の一種として否定の中の汎化記号 ∀ と置き換えるという方法である。

Agda では数学的構造は二つの record を使うのが便利である。record は直積なので公理の集合を記述することができる。公理は数学的構造で存在するとされているものの関係を論理式で記述したものである。存在物の consructor の集合を記述する record と。その間の関係を表す record の二つにわけることで、入力となる要素、必要な constructor、論理式を分離することができる。

```
record ZF {n m : Level } : Set (suc (n ⊔ m)) where
 field
   ZFSet : Set n
   _∋_ : ( A x : ZFSet  ) → Set m
   _≈_ : ( A B : ZFSet  ) → Set m
   ∅  : ZFSet
   _,_ : ( A B : ZFSet  ) → ZFSet
   Union : ( A : ZFSet  ) → ZFSet
   Power : ( A : ZFSet  ) → ZFSet
   Select :  (X : ZFSet  ) → ( ϕ : (x : ZFSet ) → Set m ) → ZFSet
   Replace : ZFSet → ( ZFSet → ZFSet ) → ZFSet
   infinite : ZFSet
   isZF : IsZF ZFSet _∋_ _≈_ ∅ _,_
     Union Power Select Replace infinite
```

ZF 集合論に必要なレベルは集合と関係の二つだけであり、ZF 集合論の集合の階層は Agda のレベルとしては出てこない。集合を定義する任意の述語や関数は既に含まれている。ZF 集合論の公理は [6] にあるものを使用する。

5　Agda での ZF 公理系の記述

record IsZF は、**record ZF** の前で定義され、ZF に登場する存在物の間に成立する論理式の直積、つまり、公理の集合になる。引数として **ZF** で定義されるものの型をすべて取る。**IsEquivalence** に実際の公理が記述されることになる。

```
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  ) → ( ϕ : (x : ZFSet ) → Set m ) → ZFSet )
    (Replace : ZFSet → ( ZFSet → ZFSet ) → ZFSet )
    (infinite : ZFSet)
      : Set (suc (n ⊔ m)) where
  field
    isEquivalence : IsEquivalence {n} {m} {ZFSet} _≈_
```

対の公理 **∀ x ∀ y ∃ z ∀ t (z ∋ t ⇔ t ≈ x ∨ t ≈ y)** は要素を二つ含む集合の存在を表している。**z** の要素は **x** か **y** に等しく、また、そのようなある集合がある (作れる) ことを意味している。この **_⇔_** は二方向の推論を含んでいる。これは **∧** を使って以下のように定義される。

```
_⇔_ : {n m : Level } → ( A : Set n ) ( B : Set m )  → Set (n ⊔ m)
_⇔_ A B =  ( A → B ) ∧ ( B → A )
```

これは ⇔ を二つの論理式に分解する。ある、作れるなどの存在記号は **_,_ : (A B : ZFSet) → ZFSet** という constructor で実現する。二つの論理式は導入と除去に相当する。これは Agda の直積とは別物である。Agda の直積は **A ∧ B** だった。

```
pair→ : { x y t : ZFSet  } →  (x , y)  ∋ t  → ( t ≈ x ) ∨ ( t ≈ y )
pair← : { x y t : ZFSet  } →  ( t ≈ x ) ∨ ( t ≈ y )  →  (x , y)  ∋ t
```

Union は **∀ x ∃ y ∀ z (z ∈ y ⇔ ∃ u ∈ x ∧ (z ∈ u))** である。式の途中に **x** の要素である集合 **u** に存在記号が付いている。**Union x** の要素は **x** に含まれる集合の要素である必要があるので、除去の場合に、そのような集合 **u** の存在を取ってくる必要がある。これは **union→** の場合は入力で与えられるので 汎化記号で良いが、**union←** では **(∃ u → ((X ∋ u) ∧ (u ∋ z)))** を **¬ ((u : ZFSet) → ¬ ((X ∋ u) ∧ (u ∋ z)))** で置き換える。つまり、**u** の様な集合がないことはないと主張する。排中律があれば、あとで示すように選択公理が成立するので、選択公理から実際に **u** を得ることができる。

```
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 )))
```

⊆, ∩, ∪, ｛ x ｝ などは既にあるものから構成されるので、そのままの定義が使える。存在記号のないものは、そのまま Agda の定義になる。Agda の **∀** はここでは飾りであり特に意味はない。｛ **x** ｝ は Agda のとは別ものなので注意が必要である。

```
field
  empty :  ∀( x : ZFSet  ) → ¬ ( ∅ ∋ x )
  extensionality :  { A B w : ZFSet  } → ( (z : ZFSet)
    → ( A ∋ z ) ⇔ (B ∋ z)  ) → ( A ∈ w ⇔ B ∈ w )
  infinity∅ :  ∅ ∈ infinite
  infinity :  ∀( x : ZFSet  ) → x ∈ infinite →  ( x ∪ ｛ x ｝ ) ∈ infinite
  selection : { ϕ : ZFSet → Set m } → ∀ { X y : ZFSet  }
   →  ( ( y ∈ X ) ∧ ϕ y ) ⇔ (y ∈  Select X ϕ )
```

置換公理

```
∀ x ∀ y ∀ z ( ( ϕ ( x , y ) ∧ ϕ ( x , z ) ) → y = z )
  → ∀ X ∃ A ∀ y ( y ∈ A ↔ ∃ x ∈ X ϕ ( x , y ) )
```

には Union と同様に対偶形式を用いることで対応できる。

```
replacement← : { ϕ : ZFSet → ZFSet} → ∀ ( X x : ZFSet  )
   → x ∈ X →  ϕ x ∈  Replace X ϕ
replacement→ : { ϕ : ZFSet → ZFSet} → ∀ ( X x : ZFSet  )
   →  ( lt : x ∈  Replace X ϕ )
   → ¬ ( ∀ (y : ZFSet)  →  ¬ ( x ≈ ϕ y ) )
```

冪集合の公理では二重否定を使う。これは、冪集合の定義に置換公理を用いる都合である。**_⊆_** は暗黙の変数を持っているが、これは明示的に受ける必要がある。選択公理があるなら後で示すように排中律を証明できるので、この二重否定を元に戻すことができる。

```
power→ : ∀( A t : ZFSet  ) → Power A ∋ t → ∀ {x}  →  t ∋ x → ¬ ¬ ( A ∋ x )
power← : ∀( A t : ZFSet  ) → ( ∀ {x}  →  _⊆_ t A {x})  → Power A ∋ t
```

選択公理 **∀ X [∅ ∉ X → (∃ f : X → ⋃ X) → ∀ A ∈ X (f (A) ∈ A)** の場合は関数に存在記号か付くので、明示的な関数が必要になる。この形式ではなく、record を用いる方法を後で使う。その方が証明しやすい。

```
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
```

正則公理 **∀ x (x ≠ ∅ → ∃ y ∈ x (y ∩ x = ∅))** の存在記号を関数で明示すると、形式的に選択公理と同じになってしまう。これは存在記号の範囲が広がってしまうことによる。

```
minimal : (x : ZFSet ) → ¬ (x ≈ ∅) → ZFSet
regularity : ∀( x : ZFSet  ) → (not : ¬ (x ≈ ∅))
   → (  minimal x not  ∈ x ∧ (  minimal x not  ∩ x  ≈ ∅ ) )
```

正則公理には ε -induction という別形式がある。これは超限帰納法の一種で任意の集合の要素に対して証明できれば集合全体に対して成立するというものである。一階述語論理では正則公理に変形可能だが直観主義論理では区別される。

```
ε -induction : { ϕ : ZFSet → Set m}
    → ( {x : ZFSet } → ({ y : ZFSet } →  x ∋ y → ϕ y ) → ϕ x )
    → (x : ZFSet ) → ϕ x
```

6　Agda での順序数の公理

順序数は自然数を無限の階層になるように拡張したものになるように定義する。全順序 **_o<_** が存在し推移律が成立する。つまり、二つの順序数に対して **Trichotomous** という **_≡_, _o<_** の三つの可能性を返す判定関数がある。

o∅ は最小の順序数であり、それより小さい順序数を持たない。**osuc** という次の順序数があり、これに割り込む順序数はない。つまり、**x o< osuc a** ならば、**x** は **a** に等しいか **a** より小さい。さらに順序数に関する超限帰納法が成立するとする。つまり、　**x** よりも小さい順序数 **y** について ϕ **y** が成立してる時に、ϕ **x** が成立することを示せば、すべての順序数について ϕ **x** が成立する。**y ≡ o∅** の場合は仮定は無条件に成立するので ϕ **o∅** の成立を示せば超限帰納法の前提になる。

```
record IsOrdinals {n : Level} (ord : Set n)  (o∅ : ord )
 (osuc : ord → ord )  (_o<_ : ord → ord → Set n) : Set (suc (suc n)) where
  field
    Otrans :  {x y z : ord }  → x o< y → y o< z → x o< z
    OTri : 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)
    TransFinite : {  ϕ  : ord  → Set (suc n) }
        → ( (x : ord)  → ( (y : ord  ) → y o< x →  ϕ  y ) →  ϕ  x )
        →  ∀ (x : ord)  →  ϕ  x

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

順序数の公理は **osuc**(直後順序数) を持っているので自然数を含んでいる。自然数と違って直後順序数以外の方法でも順序数を作ることができる可能性がある。そのような最初の順序数が無限であり、極限順序数と呼ばれる。自然数全体の集合はそのようなものだと考えられる。

7　Ordinal Definable Set

Ordinal Definable Set (順序数定義可能集合) はゲーデルによって導入された概念であり、順序数上の述語によって定義される集合である。これを Agda では以下のように記述する。

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

Agda は高階論理なのでこのような簡単な記述が可能となる。一階述語論理では $\psi(x_0, x_1, ..., x_n)$ などという記述が必要いになるが、Agda ではこの OD は **record { def = λ x → ... }** のように作られる。**...** のところにはその関数呼び出しの環境にある任意の変数や関数を記述できる。

もし OD が同じ順序数を解として持つならば、それは同値 **data _==_** であるとする。反射律、対称律、推移律は容易に証明できる。**def a x** は、OD である **a** が順序数 **x** を解として含む、つまり **x** を含む述語の証明である。同値性を証明するには入力として証明を受け取り、証明を返せば良い。

```
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
```

ここで推移集合を定義しておく。ある順序数以下のすべてを解に持つ OD は順序数の推移性により推移性を持つ。

```
Ord : ( a : Ordinal  ) → OD
Ord  a = record { def =  λ  y → y o< a }
```

Ord は順序数に一対一対応するので OD は順序数を含む。OD は集合のように見えるが、含んでいるのでは順序数だけなので一般的な集合ではない。OD を集合にするには、さらに仮定が必要である。

OD の def に書けるものは実際には Agda のレベルで制限される。定義により、それは OD のレベル **n** に等しいか小さい必要がある。**≡, o<**、さらに **∧, ∨, ¬** を使うことができる。返す値のレベルが適切ならば、呼び出される関数はより大きなレベルの引数を持っても良い。実際に何が書けるかは Agda の推論規則による。

8　OD と順序数の関係

最初の仮定は OD と順序数の一対一対応である。OD は順序数を含むのでおかしな気もするが、順序数は無限なので自分自身の部分集合と一対一対応が存在するので問題ない。あるいは Agda 内部で OD はメモリ上にアドレスを持つので順序数と対応すると考えても良い。つまり、OD のすべての値は対応するゲーデル番号を持つということでもある。一対一対応は二つの写像と二つの等式からなる。

```
postulate
  od→ord : OD  → Ordinal
  ord→od : Ordinal  → OD
  oiso  : {x : OD }     → ord→od ( od→ord x ) ≡ x
  diso  : {x : Ordinal } → od→ord ( ord→od x ) ≡ x
```

従来の集合論上でこれは証明可能であるがメタな議論が必要となる。我々はこの段階では ZF 集合論を持ってないので、深く追求せずにこれを仮定として採用する。

postulate Agda の命令であり仮定を表す。record を用いても良い。この仮定は OD に対する制限となる。例えば、**even : x : ℕ → x mod 2 ≡ 0** を仮定すると自然数は偶数のみが存在することになる。この仮定の元に関数の入力と出力は偶数である。

これで OD が含む順序数に OD が対応するようになったので、**∋** を定義できる。

```
_∋_ : ( a x : OD  ) → Set n
_∋_  a x  = def a ( od→ord x )
```

∋ は OD の半順序と考えられる。OD から順序数の写像でこの順序が保存されると仮定する。

```
c<→o< : {x y : OD  }
  → def y ( od→ord x ) → od→ord x o< od→ord y
```

もし、この逆 **o<→c<** も仮定するとすべての OD は順序数に限定される。つまり、**Ord x** のみの世界となる。つまり、順序数としては順序を持つが **∋** で繋がらない OD が存在する。

この仮定の元では **record { def = λ x → true }**は OD ではなくなる。この OD は他のすべての OD よりも大きいので対応する順序数を持たない。つまり、この仮定により OD の部分集合を取り扱うことになる。これは遺伝的順序定義可能集合 (HOD) とも呼ばれる。従来の集合論では **TC x** を x の推移閉包として、

HOD = { x | TC x ⊆ OD }

で定義されるものである。つまり、OD の要素はすべて OD であるような OD である。これは ZF を仮定した議論なので集合や推移閉包をを使うことができている。

OD は順序数の部分集合を指定する方程式であり、ある順序数 **a** があれば **λ x → x o< a** という形で OD を対応させることできる。つまり、任意の順序数に対して OD が存在する。これを **Ord a** と書く。これは OD の世界での順序数になっている。

仮定された写像 od→ord x は OD が順序数で表された名前を持っているということである。つまり、OD と順序数は一対一で対応している (Fig.??。しかし、この対応は上への写像ではない。OD は順序数全体に対応する方程式を持っているが、それは順序数の中にはない集合である。

OD から順序数への写像には自由度があるが、OD 側の max(順序数全体) は、それより小さい順序数と切り離されている必要がある。つまり、max とそれ以外は別な順序数の階層に属している。max を除く OD 全体を順序数の最初の階層、つまり、可算順序数に写像することも可能である。

OD 側から順序数への順序は写像 **od→ord** で保存される必要がある。これが **c<→o<**である (Fig.4)。逆 **ord→od** は順序は保存しない。もし、保存することを仮定すると、順序数と OD が正確に対応することになる。これは OD が表す集合は順序数しかないことに対応する。

9　非構成的な仮定

次の仮定は上界の存在 **sup**(Supreme Upper Bound) である。順序数の任意の関数に上界となる順序数があると仮定する。

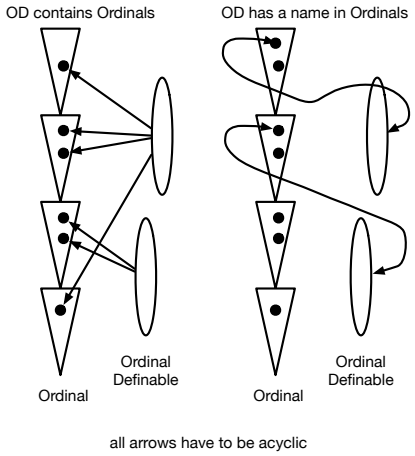

Figure 3: OD and Ordinals

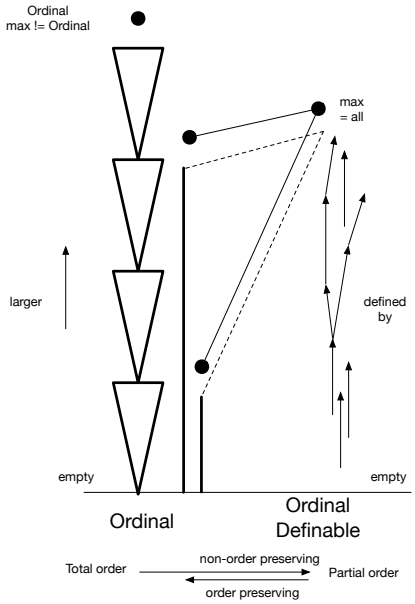

Figure 4: Mapping between OD and Ordinals

```
sup-o : ( Ordinal → Ordinal ) → Ordinal
sup-o< : { φ : Ordinal → Ordinal }
  → ∀ {x : Ordinal } → φ x o< sup-o φ
```

ここでは上界の最小性は仮定していない。自然数では ϕ が **suc** ならば上界は存在しない。順序数でも状況は変わらないので、Ordinal の上限があると明示的しないと、この仮定が成立しない可能性があるが、ここでは単純に仮定として要求している。これは **reocrd Ordinals** で定義した方が良いかも知れない。

次の公理は OD の同値性が Agda の値としての同値性 ≡ を導くというものである。これは同じ解を持つ OD をグループとしてまとめることに相当する。外延公理を証明する時に必要となる。

```
==→o≡ : { x y : OD } → (x == y) → x ≡ y
```

構成主義論理では正則公理に対応するものを仮定すると、**minimal** の存在は選択公理と同じ記述になってしまう。これは非構成的な存在記号を使用することができないためである。

```
minimal : (x : OD ) → ¬ (x == od∅ )→ OD
x∋minimal : (x : OD ) → ( ne : ¬ (x == od∅ ) )
  → def x ( od→ord ( minimal x ne ) )
minimal-1 : (x : OD ) → ( ne : ¬ (x == od∅ ) )
  → (y : OD ) → ¬ ( def (minimal x ne) (od→ord y)) ∧ (def x (od→ord y))
```

これらの仮定は Agda 上で ZF 集合論を構築する時の公理だと思って良い。この公理は順序数を仮定しているが、順序数も公理で規定されていて、具体的な順序数の実装 (record の値) は複数ありえる。それによって異なる ZF 集合論が構築される。

10　抽象的順序数による ZF 公理の恒真性の証明

証明の詳細なコードは [3] にあるので省略する。ZF 集合論の公理のいくつかは OD の視点からは単なる論理式となる。これらは順序数の性質を利用せずに証明可能である。必要なのは OD と順序数の対応だけである。OD の本質は順序数上の論理式ということである。対の公理はそういう場合となる。対は論理式として OD で定義することができる。ZF の公理はその論理式の恒真性により恒真となる。

```
_,_ : OD → OD → OD
x , y = record { def = λ t → (t ≡ od→ord x ) ∨ ( t ≡ od→ord y ) }

pair→ : ( x y t : ZFSet ) → (x , y) ∋ t → ( t == x ) ∨ ( t == y )
pair→ x y t (case1 t≡x ) = ?
```

```
pair← : ( x y t : ZFSet  ) → ( t == x ) ∨ ( t == y ) →  (x , y)  ∋ t
pair← x y t (case1 t==x) = ?
```

subst は二つの引数の型を同値関係で置き換える。**==** と **≡** の変換が使われている。**cong** は **x ≡ y** から **f x ≡ f y** を推論する合同性である。これらは Agda のライブラリで定義されている。この手の置き換えは煩雑だが、Agda では明示する必要がある。省略することにする。

通常の集合論では **pair** は二つの順序数の最大値として定義することもできるとされている。**Ord (omax (od→ord x) (od→ord y))** という形である。しかし、今回用いている方法では **(x , y) ∋ t** から、**od→ord t o< od→ord x** を導く写像 **c<→o<**が存在しない。従って、その定義を用いることはできない。

空集合は **od∅** の推移集合 **Ord** で定義され空集合の公理が成立する。

```
od∅ : OD
od∅  = Ord o∅

empty : (x : OD  ) → ¬  (od∅ ∋ x)
empty x = ¬x<0
```

¬x<0 は順序数の公理 **x : ord → ¬ (x o< o∅)** である。空集合は対応する順序数が一意に決まる。外延性の公理により、空集合は唯一つしかないこともわかる。また、空集合かどうかの判定する述語は順序数の順序を判定する述語となる。

Uinon も論理式であるが、否定を対偶の形で使用する。

```
Union : OD  → OD
Union U = record { def = λ  x →
  ¬ (∀ (u : Ordinal ) → ¬ ((def U u) ∧ (def (ord→od u) x)))  }
```

Union U は **x** を含む OD が **U** に含まれている時に **x** を含む。除去規則では、その存在を示す必要があるが構成主義論理では明示的でない存在記号は使えない。そこで「存在しないことはない」という二重否定を使う。

導入規則では **Union X ∋ z** を証明する必要があるが、これは **¬ (∀ (u : Ordinal) → ¬ ((def U u) ∧ (def (ord→od u) x)))** という否定の形をしている。否定は **A → ⊥** という形をしているので、否定が入力にある時はその入力になる値を作り、**⊥** を作り出して矛盾からは何でも導けると言う形で矛盾の除去を用いる。**u** と **(def U u) ∧ (def (ord→od u) x)** という二つの入力がある関数を否定の仮定として入力にもつ。これが **not** である。**not** に入力の **u** を順序数に変換したものと連言の条件を入力から構成して渡す。**subst** での変換が必要となる。

```
union→ :  (X z u : OD) → (X ∋ u) ∧ (u ∋ z) → Union X ∋ z
union→ X z u xx not = ⊥-elim ( not (od→ord u) ? )
union← :  (X z : OD) (X∋z : Union X ∋ z)
  →  ¬ ( (u : OD ) → ¬ ((X ∋  u) ∧ (u ∋ z )))
union← X z UX∋z =  FExists _ lemma UX∋z where
  lemma : {y : Ordinal} → def X y ∧ def (ord→od y) (od→ord z)
    → ¬ ((u : OD) → ¬ (X ∋ u) ∧ (u ∋ z))
  lemma {y} xx not = not (ord→od y) ?
```

除去では **FExists** という対偶形式の存在記号からの推論を行う。一般的に直観主義論理では一つ否定を挟むと古典論理と同じように推論することが可能になる。これは二重以上の否定を一重にすることは可能だからである。しかし、**∨** が入る場合には注意が必要である。

```
FExists : {m l : Level} → ( ϕ : Ordinal  → Set m )
   → {p : Set l} ( P : { y : Ordinal  } →  ϕ y → ¬ p )
   → (exists : ¬ (∀ y → ¬ ( ϕ y ) ))
   → ¬ p
FExists  {m} {l} ϕ {p} P =
   contra-position ( λ  p y  ϕ  y → P {y}  ϕ  y p )
```

これは **ϕ y → ¬ p** の時に **P : ¬ (∀ y → ¬ (ϕ y))** から **¬ p** を推論する。否定形の存在の仮定からの推論になる。**¬ p** の形でしか推論することはできない。実際に **p** を構築はできないからである。

contra-position の引数は以下のように三つある。最初の引数は **(A → B)** であり、 **λ p y ϕ y → P y ϕ y p** の **p** が **A** になる。残りの **λ y ϕ y → P y ϕ y p** が **B** である。つまり **contra-position** は最終的には **¬ p** を返す。しかし、引数が二つ足りない。

```
contra-position : {n m : Level } {A : Set n} {B : Set m}
  → (A → B) → ¬ B → ¬ A
contra-position {n} {m} {A} {B}  f ¬b a = ¬b ( f a )
```

良く見ると **FExists** の引数は **P** までであり、**exists** の分がない。つまり、これが **¬ B** となる。最後の足りない分は **¬ A** が **A → ⊥** なのでその **A** の分、つまり **p** である。これは、**FExists** の **¬ p** が **p → ⊥** なのでつじつまがあう。あとは第二引数つまり **exists** が **¬ B** の型になっているかを確認すればよい。**exists** は **∀ y → ¬ (ϕ y) → ⊥** なので、これが **(λ y ϕ y → P y ϕ y p) → ⊥** つまり **(λ y ϕ y → ϕ y → ⊥) → ⊥** となるので一致することがわかる。

Select と extensionality も純粋な論理式なので必要なのは順序数と OD の対応だけになる。外延性公理には **==→o≡** が必要である。ここでは入力に record のパターンを使う Agda の機能を用いている。これらの証明は省略する。

無限の公理を示すには Agda のデータ構造を用いるのが良い。これは Agda と ZF の論理式の組合せの例となっている。この場合 **Union (x , (x , x))** は、それほど重要ではなく、それが入れ子になったデータ構造 **infinite-d** が Agda で定義できて、それを OD の条件に用いることができる。**infinite-d** の **isuc** が無限に続くデータ構造を作る関数となる。

```
data infinite-d  : ( x : Ordinal  ) → Set n where
   i φ :  infinite-d o∅
   isuc : {x : Ordinal  } →   infinite-d  x  →
    infinite-d  (od→ord
     ( Union (ord→od x , (ord→od x , ord→od x ) ) ))

infinite : OD
infinite = record { def = λ x → infinite-d x }
```

ここでは順序数側で **TransFinite** を仮定しているので、これから ε **-induction** を証明することできる。超限帰納法はすべての順序数について走る。対応するすべての集合について ε **-induction** の仮定 (OD の ∋ 関係) から順序数の順序が導出され、超限帰納法の帰納の仮定が成立する。

11　置換公理と冪集合の公理

置換公理は指定された集合の要素を関数で置き換えて新しい集合を作る操作である。置換公理はより大きな集合を作る操作であり純粋に論理的な操作とは異なっている。関数の domain に相当する順序数があるのは **sup** の仮定が必要になる。置換公理で指定された関数は任意の大きさの順序数を返す可能性がある。しかし、Agda 的には順序数に過ぎずレベルは同じなままである。

置換公理にも導入と除去があり、除去規則では要素に対応する関数の入力を見つける必要がある。値から入力を見つけるために別な集合 **in-codomain** を導入する。**Replace** は上界よりも小さいという条件とこの集合の要素であるという二つの条件を持つ connstructor になる。

冪集合の公理は部分集合全体という集合があるという公理である。部分集合自体は簡単である。部分集合をベン図のように二つの集合から生成する。

```
ZFSubset : (A x : OD  ) → OD
ZFSubset A x =  record { def = λ y → def A y ∧  def x y }
```

集合論では、構成可能集合を使うことが多いが、ここでは、冪集合を含む推移集合 **Ord** を上界を使う。この方法は Osius の Power Set Functor[5] と同じ方法である。

```
Def :  (A :  OD ) → OD
Def  A = Ord ( sup-o
  ( λ  x → od→ord ( ZFSubset A (ord→od x )) ) )
```

sup-o は大小関係しか与えないが、**Ord** は大小関係がそのまま ∈ に対応するので、**Def A** は **A** を任意の集合で切り取られた部分集合全体を含む推移集合になっている。推移集合なので、**Def A** の要素である部分集合は空集合を必ず要素として含む。ここでの **sup-o** の定義域の制限は自明には存在しないが、構成する ZF のモデル全体としてはなんらかの値で押さえられていると考えられる。Agda の証明的には必要ない。

これに対して **A** が **Ord a** の形時の導入規則が証明できる。除去規則を証明することは **t** も **Ord** の時にのみ可能である。これは排中律が存在しないことに関係していると思われる。上界の性質と **t ∋ x → (Ord a) ∋ x** から証明できる **ZFSubset (Ord a) t == t** を用いる。

```
ord-power← : (a : Ordinal ) (t : OD) → ({x : OD}
     → (t ∋ x → (Ord a) ∋ x)) → Def (Ord a) ∋ t
ord-power← a t t→A  = ? where
   lemma :  od→ord (ZFSubset (Ord a) (ord→od (od→ord t)) )
     o< sup-o ( λ  x → od→ord (ZFSubset (Ord a) (ord→od x)))
   lemma = sup-o<
```

通常の集合に対する冪集合は推移集合と通常の集合との共通集合を使う。通常の集合 **A** は、**Ord (od→ord A)** に含まれ、**A ∩ Ord (od→ord A)** は **A** に等しい。一旦、推移集合による部分集合全体の集合 **Def (Ord (od→ord A))** を作ったのちに、置換公理により、その要素を **A** との共通部分に置き換えることにより、空集合を含まない集合に対する冪集合を定義できる。

12　選択公理と排中律

ここでは直観主義論理における選択公理と排中律 (he Low of exclude middle (LEM)) に関する証明を行う.

選択公理を仮定すると、OD が実質的に述語であることから LEM を導出することができる。

```
ppp :  { p : Set n } { a : OD  } → record { def = λ  x → p } ∋ a → p
ppp  {p} {a} d = d
```

ppp は要素がある時に **p** が真となる集合である。これが空かどうかを順序数を用いて判定することができるが、選択公理があれば、空でなければ実際に要素を

取ってくることができる。それは **p** の証明を実際に取ってこれることを意味する。**minimual** は選択公理と同等の正則公理の集合を取ってくる関数である。

```
p∨¬p : ( p : Set n ) → p ∨ ( ¬ p )  -- assuming axiom of choice
p∨¬p  p with is-oℴ ( od→ord ( record { def = λ x → p } ))
p∨¬p  p | yes eq = case2 ?
p∨¬p  p | no ¬p =
    case1 (ppp  {p} {minimal ps ( λ eq →  ¬p ?)} ? ) where
  ps = record { def = λ x → p }
```

逆に LEM があれば、∋ に関する決定関数 (**∋-p : (A x : OD) → Dec (A ∋ x)** を持つので、超限帰納法により空でない集合から要素を取り出すことができる。これは、集合の整列性から選択関数を作ることに相当する。OD の整列性は既に仮定されているが、その整列性は構成された ZF のモデルの中か見えるとは限らない。

この構成では選択公理は LEM と同等なので、直観主義論理上では選択公理は証明できない。ここでは選択公理を恒真にしたり否定したりするモデルを構築することはしてない。

13　Agda から見た ZF の公理の性質

ZF の公理のうちのいくつかは、順序数上の論理的な関係に対応する。例えば、pair は二つの順序数の最大値に対応する。これらは純論理的な公理となっている。

今回の ZF のモデルの構築で使われている非構成的な仮定はいくつかあり、冪集合の公理や置換公理で使われている。任意の関数の上界の存在が非構成的な仮定になっている。選択公理や、集合の同一性を表す外延性の公理なども非構成的な仮定、つまり、Agda で明示的に仮定することが必要な公理である (Fig.5)。

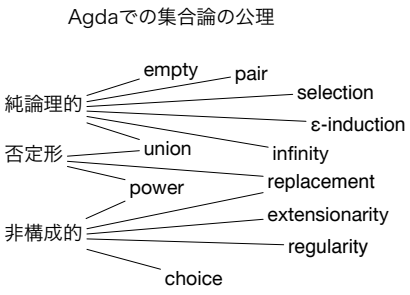

Figure 5: Various kind of ZF Axiom

前の説で述べたように、正則公理をそのまま採用すると、構成主義論理できは選択公理と同じ物になってしまう。それは minimum な要素を取ってこれるという公理であり、任意の集合から構成的に特定の要素を取ってくることが可能になる。一階述語論理では、それを非構成的な存在として記述することが存在記号によって可能である。

選択公理と排中律は、今回のモデルの中では同じ命題になっている。これは、OD が順序数への写像を持ち既に整列されているからである。しかし、この整列性は、構成されたモデルの中からは観測されない。二つの OD が与えられた時に、その順序を決定することはできない。つまり、モデル内部での集合の 整列定理には自由度がある。今回は選択公理、つまり排中律を仮定してから内部の整列定理を証明することは行っていない (Fig.6)。

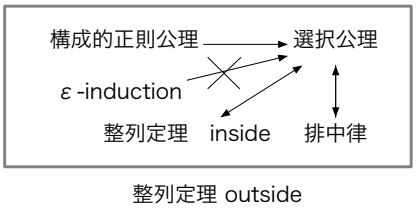

Figure 6: Axiom dependency

14　結論

80 年代に研究された構成的集合論は構成論理に合わせた集合論であり、古典的な集合論とは異なる。この構成では従来の ZF の公理の恒真性を持つモデルを Agda を元に構成しているので、古典的な集合論を実装したものになっている。特に選択公理を仮定すれば排中律も導出されるので古典的な集合論を一階述語論理上で展開したものと同一になっている。

圏論的に定義された Categorical Set Theory は、積と関数適用を持つ圏 (Caritesian Close Category) に真理値を表す Subobject classifier を加えた Topos 上に作られるものである。CST では推移性を持つ集合を基本とするので集合が圏を構成する。この上で冪集合を得る関手を定義する。そして通常の集合を推移集合と通常の集合の共通部分集合として定義する。この論文ではモデル論的な議論がされているが、証明論的な議論も可

能だとされており、その証明論的な部分を遂行したものとなっている。

ここでは Agda 上でいくつかの公理を仮定して、ZF 集合論のモデルを構築した。構築は抽象的な順序数を用いるものと、具体的な実装を持つ順序数を用いるものとがある。使用した公理は順序数上の方程式である OD と順序数の対応と、OD の包含関係と順序数の順序がその対応で一方向に保存されること、そして、順序数上での上界の存在、そして、選択公理である。

この順序数の実装として、フィルタを使うことができるはずである。この集合論のモデルの構築では濃度に付いては触れていない。それには集合内部での集合同士の一対一対応を定義する関数を定義する必要がある。それが ZF 集合論内部から見た集合の濃度になる。

この構成主義論理上の ZF 公理系上に従来の集合論や位相空間論をそのまま構築することも可能であると思われる。その際に、排中律を仮定して問題ない。排中律を仮定すると自動的に選択公理も仮定されることになる。逆に排中律を仮定せずに選択公理を仮定することはできない。

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, pp. 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, Vol. 3, No. 1, pp. 79–126, 2010.

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

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

[5] Gerhard Osius. Categorical set theory: A characterization of the category of sets. iJournal of Pure and Applied Algebra, No. 4, pp. 79–119, 1974.

[6] 田中尚夫. 公理的集合論. 培風館, 1982.