Agda による Automaton の記述

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

Abstract

Automaton[9] はコンピュータの能力の限界や階層、計算量を理解するための重要な理論である。Automaton の理論に関する証明を Agda[7] で行う [5]。従来の教科書は一階述語論理と素朴集合論程度で書かれていて、数学的な構造や定理、証明が形式化されていることはほとんどない。Agda はプログラミング言語であると同時に定理証明系なので、Automaton の実装と、その性質の証明を同じ言語で行うことができる。Automaton の基本的な部分の証明を理解することがプログラムの信頼性を理解することにつながる。Agda は古典的な一階述語論理と異なる証明により真理値が決まる直観主義論理であり、それを理解することは数学の証明が何かを理解することになる。特に Automaton と非決定性 Automaton の意味 (language) としての同等性を簡単に示すことができる。Automaton の状態の有限性に関しても考察を行う。

Automaton[9] is a important theoty of understanding limit and hierarchy, complexity of computation. We describe proofs of Automaton in Agda[5]. In traditional Automaton text, their description may not formalized enough, in which mathemtical structure and propsition, proofs are written in old first order logic and naive set theory. Agda[7] is both a programming language and proof system, so Automaton itself and its theorem and proof are written in the same Agda. Understanding basics of Automaton makes it possible to understanding reliablity of programs. Agda is an intutionistic logic which defines truth values as proofs. The diffrent from classical first order logic leads to the understanding of mathematical proof itself. In this method, equivalence of Automaton and non deterministic Automaton is easily proved. We also discuss effect of the finiteness of Automaton states.

1 Automaton と Logic を学ぶ理由

Automaton[9] は計算機科学の基礎的な部分だが、いくつかの部分で学ぶ理由がある。一つは、今、自分たちが使える計算機は量子計算機を含めても、Finite Automaton でしかないからだから。巨大なニューラルネットワークでも、それは Finite Automaton に過ぎない。

もう一つは、それが人間が理解できる計算機の限界でもあるから。もちろん、状態や状態遷移が複雑になればそれは人の限界を越えるわけだが、少なくとも理解できる範囲ではある。

さらに、これは仕様記述の手法の一つになっている。入出力の正しさは一階あるいは高階の論理で記述されるが、時間的な仕様記述、例えば、「公平に実行される」「いつか実行される」などの性質は、時相論理で記述する。それは ω-Automaton に相当する。

そして計算量の P と NP は決定的あるいは非決定的な Turing Machine (TM) として定義されて、それは決定性 Automaton (FA) と非決定性 Automaton (NFA) で定義されている。もちろん、TM の停止性を判定する TM が存在しないことの証明はプログラムに関わる人は理解しているのが望ましい。

この時に、もちろん、これらの証明を記述する方法が必要になる。それも、また Automaton を学ぶ理由の一つになる。もちろん、既に素晴らしい教科書が既にたくさんあるのだが、Agda[7] を使うことにより、命題と証明を形式化することができる [5]。

しかし、古典的な一階述語論理と集合を用いた記述と、直観主義論理/Agda を用いた記述はもちろん異なる。これは、それぞれに異なる得意な領域があるからである。ここでは、その一部を具体的に考察する。

2　Agda を使った Automaton の学習

Agda は定理証明支援系だが、Haskell[8] に近い構文を持つ単なる純関数型言語でもある。関数型言語の型とそれを実装する関数は、命題と証明とに読み替えることができる [4]。

Agda は Sum type を持っているので、List を **[]** と cons **::** を用いて、二種類の constructor を場合分けとして持つデータ構造として記述できる。

Agda の **Set** は型、あるいは命題を表す一般的な型だが、実際には矛盾を避けるために Level がついている。ここでは、それをできるだけ省略することにする。

```
data  List  (A : Set ) : Set  where
    [] : List A
    _::_ : A → List A → List A
```

これにより append は以下のように実装される。

```
  _++_ :  {A : Set } → List A → List A → List A
  []      ++ ys = ys
  (x :: xs) ++ ys = x :: (xs ++ ys)
```

これは普通の再帰的なプログラムになっている。ℕ は自然数であり、これにより長さを求めることができる。

```
data  ℕ : Set  where
  zero : ℕ
  suc : ℕ → ℕ

length : List A → ℕ
length [] = zero
length (h :: t) = suc (length t)
```

Agda では、さらに等式を **data** として定義できる。

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

これは「**x ≡ x**」という型、つまり、命題「**x ≡ x**」の証明が constructor **refl** であると定義している。これが Curry Howard 対応で、純関数型言語の型と直観主義論理の命題が対応し、純関数型言語の λ 項と直観主義論理の推論が対応することになっている。

この時の等しさは、**x ≡ y** の両辺を評価して既約形にした時に λ 項取して同じだということを意味している。これは、x と y の単一化を実行したのとだいたい同じになっている。

命題 **A → B** の証明は、**A → B** の型を持つ λ 項であり、もちろん、それは、その型を持つ関数のことになっている。証明を表す λ 項は普通のプログラムの関数としての意味を持たないことも多い。

Agda は Haskell の構文とほぼ同じだが、**{}**の部分が異なる。これは省略可能な変数となっている。また、data と record が分離されているのが異なる。直接的には Set で型自身を値として返す関数が定義できるのが大きな違いである。

3　等式変換と再帰による証明

```
length-lemma : (xs ys : List A) → length (xs ++ ys) ≡ length xs + length ys
length-lemma length-lemma [] ys = refl
length-lemma (x :: xs) ys = begin
   length ((x :: xs) ++ ys ) ≡⟨ cong suc (length-lemma xs ys) ⟩
   length (x :: xs ) + length ys   where open ≡-Reasoning
```

この構文は、≡ を連続して変形していく構文になっている。これ自体も Agda で記述されている。ここで、

```
cong : {N : Set } →  (f : N → N ) → { x y : N } → x ≡ y → f x ≡ f y
cong f refl = refl
```

で、f を両辺に適用しても等しい合同性 (congluent) になっている。

length-lemma xs ys が再帰的に呼び出されることにより Induction が実行されている。

ここで、この Induction は List の data を分解していく形の Induction であり、その分解が正しく行われていることを Agda がチェックする。つまり、この形で、length の停止を Agda が確認する必要がある。

4　Agda の safe な推論

一般的には再帰的なプログラムは停止するとは限らない。その場合は**{-# TERMINATING #-}** という annotation をつける必要がある。この場合は、Agda の推論は unsafe と言われる。unsafe な推論はいくつかあり、**{-# OPTIONS --cubical-compatible --safe #-}** を付けることにより、それを確認できる。Cubical[6] は Agda の最近の型理論の一つである。

Agda には Cubical type があり、それと互換性があることもチェックできる。

5　∧ と ∨ と ¬

基本的な論理演算子として、∧ と ∨ と ¬ を

```
record _∧_ (A : Set ) ( B : Set ) : Set where
 constructor _,_
 field
  proj1 : A
  proj2 : B

data _∨_ (A : Set ) ( B : Set ) : Set where
 case1 : A → A ∨ B
 case2 : B → A ∨ B

data ⊥ : Set where

¬_ : Set → Set
¬ A = A → ⊥
```

と定義できる。record が直積を表し、data が場合分けを表す。

⊥ は constructor のない型/命題であり、その場合がないこと、あるいは矛盾を表している。

6 Bool

命題論理は true / false をつかって表すこともできる。

```
data Bool : Set where
  true : Bool
  false : Bool

_∧_ : Bool → Bool → Bool
true ∧ true = true
_ ∧ _ = false

_∨_ : Bool → Bool → Bool
false ∨ false = false
_ ∨ _ = true

not : Bool → Bool
not true = false
not false = true
```

つまり、Agda には真理値を表すのに、Set を使う場合と Bool を使う場合の二種類がある。

この使い分けは古典論理では出てこない。

7 関数型言語 Agda による Automaton の記述

これで準備ができたので、いよいよ Automaton を記述する。数学的な構造は型の間の関数と論理的関係であり record をつかって記述する。

```
record Automaton ( Q : Set ) ( Σ : Set )
 : Set where
 field
  δ : Q → Σ → Q
  aend : Q → Bool
```

これは Automaton の要件であり、その意味は以下の関数で定義する。

```
accept : { Q : Set } { Σ : Set }
  → Automaton Q Σ
  → Q
  → List Σ → Bool
accept M q [] = aend M q
accept M q ( H :: T ) = accept M ( δ M q H ) T
```

ここで、Q は有限という条件は付いてない。また、Set ではなく Bool で定義されている。

8 NFA の記述

非決定性 Automaton (NFA) は以下のように定義する。

```
record NAutomaton ( Q : Set ) ( Σ : Set )
 : Set where
 field
  Nd : Q → Σ → Q → Bool
  Nend : Q → Bool
```

Automaton では Q を返す関数 δ があったが、ここでは Q の部分集合 **Q → Bool** を使う。つまり、Q の要素 (値) に対して、true / false を返す関数なので、条件の重複を許すことになる。

この意味は、以下のように定義する。

```
Naccept : { Q : Set } { Σ : Set }
  → NAutomaton Q Σ
  → (exists : ( P : Q → Bool ) → Bool)
  → (Nstart : Q → Bool) → List Σ → Bool
Naccept M exists sb [] = exists ( λ q → sb q ∧ Nend M q )
Naccept M exists sb (i :: t ) = Naccept M exists
 ( λ q → exists ( λ qn → (sb qn ∧ ( Nd M qn i q ) ))) t
```

ここで、**(exists : (P : Q → Bool) → Bool)** は、Q の部分集合 **(P : Q → Bool)** が空かどうかを判定する関数のつもりである。この定義を含めて、NFA の定義になるが、これを **NAutomaton** に入れない方が良い。

この NFA の記述は普通の教科書の記述とは異なる。実際、Automaton でも、自然数の添字を使った表現が多い。教科書にまったく沿った記述も可能だが、直観主義論理には向かない。また、**(P : Q → Bool)** を使うことにより、複雑な集合論を回避している。

9 NFA が Automaton であることの証明

直観主義論理では、この証明はまったく単純である。以下のように実際に Automaton を構成する。record を実際に構築すれば良い。

```
subset-construction : { Q : Set } { Σ : Set  } →
  ( ( Q → Bool ) → Bool ) →
  (NAutomaton Q  Σ ) → (Automaton (Q → Bool)  Σ )
subset-construction {Q} { Σ }  exists NFA = record {
      δ = λ f i q → exists ( λ r → f r ∧ Nd   NFA r i q )
   ; aend = λ f  → exists ( λ q → f q ∧ Nend NFA q )
  }
```

これは subset-construction でもなんでもない。単に部分集合 verb+(P : Q → Bool)+をを Automaton の状態 Q としているだけである。

```
subset-construction-lemma→ : { Q : Set } {
Σ : Set  } → (exists : ( Q → Bool ) → Bool ) →
  (NFA : NAutomaton Q  Σ ) → (astart : Q → Bool )
  → (x : List Σ )
  → Naccept NFA exists astart  x ≡ true
  → accept (  subset-construction exists NFA ) astart  x ≡ true
subset-construction-lemma→ {Q} { Σ }  exists NFA astart x naccept
    = lemma1 x astart naccept where
  lemma1 :  (x : List Σ ) → ( states : Q → Bool )
    → Naccept NFA exists states x ≡ true
    → accept (  subset-construction exists NFA ) states x ≡ true
  lemma1 [] states naccept = naccept
  lemma1 (h :: t ) states naccept
    = lemma1 t ( δ  conv exists (Nd NFA) states h) naccept

subset-construction-lemma← : { Q : Set } {
Σ : Set  }  → (exists : ( Q → Bool ) → Bool ) →
  (NFA : NAutomaton Q  Σ ) → (astart : Q → Bool )
  → (x : List Σ )
  → accept (  subset-construction exists NFA ) astart x ≡ true
  → Naccept NFA exists astart x ≡ true
subset-construction-lemma← {Q} { Σ }  exists NFA astart x saccept
   = lemma2 x astart saccept where
  lemma2 :  (x : List Σ ) → ( states : Q → Bool )
    → accept (  subset-construction exists NFA ) states x ≡ true
    → Naccept NFA exists states x ≡ true
  lemma2 [] states saccept = saccept
  lemma2 (h :: t ) states saccept
    = lemma2 t ( δ  conv exists (Nd NFA) states h) saccept
```

このように簡単になるのは、実行の定義が List の分解に沿って行われるからであることがわかる。

Automaton と NAutomaton の record は状態間の関係を表しているだけで、実際の実行は別に指定されている。これは、前者はプログラムの記述だが、実行とは独立であることを意味してる。同じプログラムに対して、ことなる実行を定義することもできる。

10 古典論理と直観主義論理の違いと言語の記述

言語は以下のように定義される。

```
language : { Σ : Set } → Set
language { Σ } = List Σ → Bool
```

これは List Σの部分集合のことである。A が language なら **A x ≡ ture** は x が言語 A に属することを意味する。

例えば、Union を以下のように簡単に定義できる。

```
Union : { Σ : Set} → ( A B : language { Σ } ) → language { Σ }
Union { Σ } A B x = A x ∨ B x
```

しかし Concatenetation の定義は難しい。二つの List の結合が以下の record で表される。

```
record Split { Σ : Set} (A B : List Σ → Bool )
  (x :  List Σ ) : Set where
field
   sp0 sp1 : List Σ
   sp-concat : sp0 ++ sp1 ≡ x
   prop0 : A sp0 ≡ true
   prop1 : B sp1 ≡ true
```

部分、sp0 と sp1 があり、sp0 は言語 A であり、sp1 は言語 B に属する。これは、まったく複雑だが、これを使って、

```
concat' : { Σ : Set } → (A B : List Σ → Bool ) →  Set
concat' { Σ } = List Σ → Split A B
```

とできる。しかし、これは、

```
concat : { Σ : Set } → (A B : List Σ → Bool ) → List Σ →  Bool
```

とは違うものである。これを以下のように定義することができる。

```
split : { Σ : Set} → (x y : language { Σ } ) → language { Σ }
split x y  [] = x [] ∧ y []
split x y (h  :: t) = (x [] ∧ y (h  :: t)) ∨
 split ( λ t1 → x (  h :: t1 ))  ( λ t2 → y t2 ) t

concat : { Σ : Set } → (A B : List Σ → Bool ) →  Bool
concat A B = split A B
```

実際、

```
test-AB→split : { Σ : Set} → {A B : List In2 → Bool}
  → split A B ( i0 :: i1 :: i0 :: [] ) ≡ (
    ( A [] ∧ B ( i0 :: i1 :: i0 :: [] ) ) ∨
    ( A ( i0 :: [] ) ∧ B ( i1 :: i0 :: [] ) ) ∨
    ( A ( i0 :: i1 :: [] ) ∧ B ( i0 :: [] ) ) ∨
    ( A ( i0 :: i1 :: i0 :: [] ) ∧ B  [] )
```

が示せる。

```
split→AB :  { Σ : Set} → (A B : List Σ → Bool ) → ( x : List Σ )
  → split A B x ≡ true → Split A B x

split→AB1 :  { Σ : Set} → (A B : List Σ → Bool ) → ( x : List Σ )
  → Split A B x →  split A B x ≡ true
```

を実際に証明することができる。これは、

```
lemma-concat : { Σ : Set} → ( A B : language { Σ } ) → (x : List Σ)
  → Split A B x ∨ ( ¬ Split A B x )
lemma-concat { Σ } A B x with split A B x in eq
... | true = case1 (split→AB A B x eq )
... | false = case2 ( λ  p →  ¬-bool eq (split→AB1 A B x p )) where
  ¬-bool : {a : Bool} →  a ≡ false  → a ≡ true → ⊥
  ¬-bool refl ()
```

つまり Split A B x に対して排中律が成立すること意味している。ここで ... を使った場合分けを使った。

つまり、**List Σ → Bool** と **List Σ → Set** とは、Set にはいる部分に排中律が成立すれば同じことになる。逆に、排中律が成立しないなら、Bool と Set とは使い分けが必要になる。

```
langauge' : { Σ : Set } → List Σ  → Set
```

は、排中律が成立しない record を指定することもできる。

同様に Star を定義することもできる。

```
Star : { Σ : Set} → ( A : language { Σ } ) → language { Σ }
Star { Σ } A [] = true
Star { Σ } A (h :: t) = split A ( Star { Σ } A ) (h :: t)
```

としたくなるが、これは停止性が agda に自明にならない。つまり、safe にならない。これは無限列を生成するが遅延評価があるので **A []** が成功しなければ動作はする。少し工夫することにより、これを safe にすることは可能で

```
repeat : { Σ : Set} → (P : List Σ → Bool) → (pre y : List Σ ) → Bool
repeat P [] [] = true
repeat P (h :: t) [] = P (h :: t)
repeat P pre (h :: []) =      P (pre ++ (h :: []))
repeat P pre (h :: y@(_ :: _)) =
    ((P (pre ++ (h :: []))) ∧ repeat P [] y )
    ∨ repeat P (pre ++ (h :: [])) y
```

ここで、split も repeat も停止性が Agda に対して明示されているので、safe な推論になっている。

```
record StarProp { Σ : Set} (A : List Σ → Bool ) (x :  List Σ ) : Set where
 field
  spn : List ( List Σ )
  spn-concat : foldr ( λ k → k ++_ ) [] spn ≡ x
  propn : foldr ( λ k → λ j → A k ∧ j ) true spn ≡ true
```

に対して、repeat が同等に動作することも証明可能である。

11　有限性

ここまで、状態の有限性は問題にしていない。問題になるのは、**(exists : (Q → Bool) → Bool)** の実装で、Q が有限ならばこれは決定的な関数になる。しかし、Q が有限でなくても、これがあれば問題なく NFA が定義される。

正規言語は「有限 Automaton が受理する言語」なので有限性は必要になる。これを外すと、例えばかっこの対応などを、この Automaton の定義では実現できてしまう。

有限性は、有限な自然数 **Fin n** に一対一対応があるといおうことで、これを以下の record で定義できる。

```
record FiniteSet ( Q : Set ) : Set  where
 field
  finite : ℕ
  Q←F : Fin finite → Q
  F←Q : Q → Fin finite
  finiso→ : (q : Q) → Q←F ( F←Q q ) ≡ q
  finiso← : (f : Fin finite ) → F←Q ( Q←F f ) ≡ f
```

さまざまなデータ構造が有限であり、例えば、

```
fin2List : {n : ℕ } → FiniteSet (Vec Bool n)
```

などを示せる。

```
fin→ : {A : Set} → FiniteSet A → FiniteSet (A → Bool )
```

も示せるが、これには少し問題がある。FiniteSet だと、これを示すには関数 **(A → Bool)** の ≡ が必要になる。これは、すべての入力に対して、出力が同じなら関数も同じという関数外延性が必要でこれは safe とはされていない。実際、同じ関数は異なるλ式で定義されるので一般的には正しくない。

```
Extensionality : (a b : Level) → Set _
Extensionality a b =
  {A : Set a} {B : A → Set b} {f g : (x : A) → B x} →
  (∀ x → f x ≡ g x) → f ≡ g
```

これを postulate つまり仮定することができる。これは入項の関数を入力と値の同一性で割った商集合の同値性だと考えることもできる。

これを避けるには、値ごとに等しいという関係にしてやれば良い。そうすると、FiniteSet を直接には使えなくなる。Turing Machine の停止性の部分で、これに代わるものを使う。

12 正規言語

正規言語は有限 Automaton で受理される言語だが、既に NFA との同値性を示したので、NFA で受理される言語で良い。

NFA の状態が有限である必要があるが、Q が有限なら **Q → Bool** は (関数外延性の仮定の元に) 全体の状態が有限であることはわかる。なので、NFA の直接の状態 Q の有限性のみを要求すると簡単になる。

実際、**Q → Bool** を有限な数値に対応させれば、そのまま有限 Automaton になる。この変換が簡単というわけではないので注意が必要である。例えば BDD などを使うことができると思われる。

```
record NRegularLanguage ( Σ : Set ) : Set (Suc Zero) where
 field
  states : Set
  astart : states  → Bool
  aexists : (states  → Bool) → Bool
  afin : FiniteSet states
  nfa : NAutomaton states Σ
 ncontain : List Σ → Bool
 ncontain x = Naccept nfa aexists astart x
```

Automaton は NFA で簡単に記述できるし、一階述語論理では NFA の記述が複雑なので Automaton の結合を NFA で表すとか、ε 遷移とかを工夫することになるが、NFA の記述が充分に簡単なのですべてを NFA で処理する方が楽である。

13 Concat が正規言語について閉じる証明

NFA の結合は、以下のように比較的簡単に定義することができる。

```
NConcat-NFA :  { Σ : Set} → (A B : NRegularLanguage Σ ) → NAutomaton (states A ∨ states B )
Σ
NConcat-NFA { Σ } A B = record { Nd = δ  nfa ; Nend = nend }
  module Concat-NFA where
    δ  nfa : states A ∨ states B → Σ → states A ∨ states B → Bool
    δ  nfa (case1 q) i (case1 q ) = Nd (nfa A) q i q
    δ  nfa (case1 qa) i (case2 qb) = (Nend (nfa A) qa ) ∧ aexists B ( λ  qn → astart B qn ∧ Nd (nfa B) qn i q
    δ  nfa (case2 q) i (case2 q ) = Nd (nfa B) q i q
    δ  nfa _ i _ = false
    nend : states A ∨ states B → Bool
    nend (case2 q) = Nend (nfa B) q
    nend (case1 q) = Nend (nfa A) q ∧ aexists B ( λ  qn → Nend (nfa B) qn ∧ astart B qn ) -- empty B case
    ncfin : FiniteSet (states A ∨ states B )
    ncfin = fin-∨ (afin A) (afin B)
    nexists : (states A ∨ states B  → Bool) → Bool
    nexists f = exists (ncfin) f
    nstart : states A ∨ states B  → Bool
    nstart (case1 x) = astart A x
    nstart (case2 x) = false

closed-in-concat→  : { Σ : Set} → (A B : NRegularLanguage Σ ) → ( x : List
Σ )
   → Naccept (NConcat-NFA A B) (Concat-NFA.nexists A B)  (Concat-
NFA.nstart A B)  x ≡ true
   → split (ncontain A ) (ncontain B) x ≡ true
closed-in-concat→  { Σ } A B x na = ?

closed-in-concat←  : { Σ : Set} → (A B : NRegularLanguage Σ ) → ( x : List
Σ )
   → split (ncontain A ) (ncontain B) x ≡ true
   → Naccept (NConcat-NFA A B) (Concat-NFA.nexists A B)  (Concat-
NFA.nstart A B)  x ≡ true
closed-in-concat←  { Σ } A B x na = ?
```

を証明すれば良い。Star についても同様に証明することになる。

14 その他の記述手法

Automaton の記述方法は他にもたくさんある。Agda のライブラリで最近良く使われているのは Codata と呼ばれる coinduction[3] を使ったものである。以下の例では field を関数形式で定義する構文を使っている。**record {}** 使う構文は coniductive な record には使えない。

```
data Colist (i : Size) (A : Set) : Set where
  [] : Colist i A
  _∷_ : {j : Size< i} (x : A) (xs : Colist j A) → Colist i A

record  Lang (i : Size)  : Set  where
  coinductive
  field
    ν  : Bool
    δ  : ∀{j : Size< i} → A → Lang j

_∋_ : ∀{i} → Lang i → Colist i A → Bool
l ∋ [] =  ν  l
l ∋ ( a ∷ as ) =  δ  l a ∋ as

trie : ∀{i}  (f : Colist i A → Bool) → Lang i
 ν  (trie f) = f []
```

```
δ (trie f) a = trie (λ as → f (a :: as))
```

List は、それが入力で与えられれば長さが決まるが、無限の長さの List を作ることができる。長さを最初に指定するとそれを制限することができる。Size は agda の builtin な型になっている。ℕ と同じだが、Agda で書かれた定義はない。Clist でなく List を用いると、Agda は Cannot solve size constraints というエラーを出す。

```
accept : ∀{i} {S} (da : Automaton S A) (s : S) → Lang i
Lang. ν (accept da s) = Automaton.aend da s
Lang. δ (accept da s) a = accept da (Automaton. δ da s a)

nlang : ∀{i} {S} (nfa : NAutomaton S A ) (s : S → Bool ) → Lang i
Lang. ν (nlang nfa s) = exists ( λ x → (s x ∧ NAutomaton.Nend nfa x ))
Lang. δ (nlang nfa s) a = nlang nfa (λ x → s x ∧ (NAutomaton.Nd nfa x a) x)
```

という形で使うことができる。

```
_∪_ : ∀{i} (k l : Lang i) → Lang i
ν (k ∪ l) = ν k ∨ ν l
δ (k ∪ l) x = δ k x ∪ δ l x

_・_ : ∀{i} (k l : Lang i) → Lang i
ν (k ・ l) = ν k ∧ ν l
δ (k ・ l) x = let k' = δ k x ・ l in if ν k
  then k' ∪ δ l x else k' 
```

と concat が定義されているが、これは split による concat の定義と一致すれば良い。

```
LtoSplit : (x y : Lang ∞ ) → (z : Colist ∞ A)
  → ((x ・ y ) ∋ z) ≡ true
  →  split (λ w → x ∋ w) (λ w → y ∋ w) z ≡ true

SplitoL : (x y : Lang ∞ ) → (z : Colist ∞ A)
  → ((x ・ y ) ∋ z) ≡ false
  →  split (λ w → x ∋ w) (λ w → y ∋ w) z ≡ false
```

が示せれば良い。

15 GearsAgda

GearsAgda[11] は、不定の型を常に返すことにより、継続形式でのプログラミングを強制する。これにより、Hoare Logic [1] による Invariant をプログラムの記述と平行して持ち歩くことができる。

```
fa-driver : { Q t : Set } { Σ : Set } → (x : List Σ )
    → Q → (next : Q → Σ → (Q → t) → t) → (exit : Q → t) → t
fa-driver {Q} {t} { Σ } [] q next exit = exit q
fa-driver {Q} {t} { Σ } (h :: x) q next  exit
   = next q h (λ q → fa-driver x q next exit)

au-driver : { Q : Set } { Σ : Set } → (x : List Σ )
  → Q → Automaton Q Σ → Bool
au-driver {Q} { Σ } x q M = fa-driver x q
  (λ q i k → k (δ M q i)) (λ q → aend M q)
```

fa-driver は Automaton の記述を実行する accept に相当する。実際、au-driver は accept と同値になる。

16 証明できてることとできてないこと

できていること

- Automaton / NFA の実装
- Regex の意味の定義
- FiniteSet と Bijection のいくつか
- CFG の実装
- PDA の実装
- Turing Machine の実装
- UTM の実装
- ω Automaton の定義
- Muller Automaton と Buchi Automaton の対応
- NFA が Automaton であること
- Automaton の結合が Automaton であること
- Turing Machine の停止性を判定する述語が存在しないこと
- Pumpling lemma
- 正規言語でない言語の存在
- 微分法の実装
- 素数が無限にある
- 素数の平方根が無理数であること

できてないこと。これらは、問題の記述ができてるものと、そうでないものがある。

- Automaton が Star について閉じていること
- CFG が PDA で受理されること
- 微分法 [10] の Soundness
- 微分法が生成する状態の有限性
- 時相論理 [2]
- モデル検査

17 Extended Automaton in Agda

Automaton の定義は

```
record Automaton ( Q : Set ) ( Σ : Set ) : Set where
 field
  δ : Q → Σ → Q
  aend : Q → Bool
```

だが、Agda では Q は高階の型変数で有限性の制限もない。そこで、ここに色々なものをいれることができる。

例えば、Q を List Q とすると、

```
δ : List Q → Σ → List Q
```

には、とくに何も制限がないので、ランダムアクセスしても良い。つまり、これは RAM だと考えても良い。

```
Turing : ( Q : Set ) ( Σ : Set ) → Set
Turing Q Σ = Automaton ( List Q ) Σ
```

これを非決定的にするには、**List Q → Bool** とする。

```
NDTM : ( Q : Set ) ( Σ : Set ) → Set
NDTM Q Σ = Automaton ( List Q → Bool ) Σ
```

これに対して、

```
exists : ( List Q → Bool ) → Bool
```

があれば、subset-construction で Turing Machine に落ちる。subset-construction には状態の有限性は要求されないから。

exists は恣意的に決めてよくて、Agda の関数あるいは、外から与えられたすごい性質を持つ何かでもよい。

18 まとめ

Automaton 理論に関して、Agda での定義と証明をおこなった。定義は割と簡潔だが、証明は複雑で長く技巧的になる。学生とって、証明を読む意味はあまりない。しかし、自分で証明を再構成すると、理解が進む。

証明できてないことは、単純に複雑で長いというのもあるが、同じことだが時間がかかるせいでもある。

ただし、証明を思い出す時には、確実な助けになる。特に自分で書いたものならば。

既に集合論も使用可能なので、教科書上の証明はすべて Agda で書けるはずだが、それが Agda 的に良いかどうかは一階述語論理と高階直観論理の差があるので、まだ、開拓の余地がある。

Agda の safeness は、一つのコーディング規則みたいなものだが、その有用性はまだわかってない。

Codata は、safe でないという問題があるが、それは改善されると予想される。それを使った Automaton の記述が良いといわれているが、まだ確認してない。ただ、Agda の今までの推論とは少しずれがある。これらは、大きさを ℕ などで明示的に持ち歩いて、GearsAgda の loop connector で接続すると、Hoare Logic に帰着できる。

References

[1] Hoare logic in agda2. Accessed: 2018/12/17(Mon).

[2] Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. J. ACM, 37(2):279 – 317, apr 1990.

[3] Ana Bove, Peter Dybjer, and Andrés Sicard-Ramírez. Embedding a logical theory of constructions in agda. In Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification, PLPV '09, page 59 – 66, New York, NY, USA, 2009. Association for Computing Machinery.

[4] Jean-Yves Girard, Paul Taylor, and Yves Lafont. Proofs and Types. Cambridge University Press, New York, NY, USA, 1989.

[5] Shinji Kono. https://github.com/shinji-kono/automaton-in-agda, 2023.

[6] Thomas Lamiaux, Axel Ljungström, and Anders Mörtberg. Computing cohomology rings in cubical agda. In Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, page 239 – 252, New York, NY, USA, 2023. Association for Computing Machinery.

[7] 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.

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

[9] Michael Sipser. Introduction to the theory of computation. SIGACT News, 27(1):27 – 29, mar 1996.

[10] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. Symbolic boolean derivatives for efficiently solving extended regular expression constraints. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, page 620 – 635, New York, NY, USA, 2021. Association for Computing Machinery.

[11] 外間 政尊 and 河野 真治. Gears os の hoare logic をベースにした検証手法. In **電子情報通信学会ソフトウェアサイエンス研究会**, Jan 2019.