Agda による Galois 理論のプログラミング

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

Abstract

1 All connected

数学の証明に計算機を使うのは既に古い歴史がある。Agda [5] は定理証明支援系だが、普通の関数型プログラミング言語でもあり、群の要素の数え上げなどを直接に実行できる。この時に「正しく数え上げのか」ということが問題になる。4 色問題が計算機を使って解かれた時にも検算 [4] が問題になった。

Agda[1] は Curry Howard 対応に基づく定理証明を行う。命題が型であり証明がλ項に対応する。つまり、Agda は証明を変数の値として持ち歩くことができる。つまり、今計算している値だけでなく、その値の性質 (5 より小さいとか)、そして、その由来 (何から作られたのか) を値として持ち歩くことができる。計算結果が、ちゃんと要求された証明につなかっていることを示すことが可能である。つまり計算だけなくて、証明としてちゃんとつながっていることになる。

ここでは、例題としてガロア理論の一部である可解群を使い、証明を含むデータ構造として、

Data.Fin	有限な自然数
FL	順列に対応する減少列
Data.List.Fresh	**Sorted List**

を使う。一部は、Agda の Library でる。

証明付きのデータ構造は assert と異なり、それを作成する段階で証明を構成する必要がある。証明は型の整合性の検査なので、実行時ではなく、プログラムコードの入力時に決まる。しかし、Data の構成は実行に行われるので、群の要素の数え上げも型検査の時に実行される。つまり、定理証明の時間が問題になる。

証明付きデータを使う方が 10 倍速いことがあり、また、証明自体も手で書く部分が少なくなるので楽になる。ただし、List への挿入などは、証明を一緒に書くことになるのでやさしいとは言えない。

2 ガロア理論

x の多項式 f(x) が解を持てば因数定理により

$$(x - \alpha)(x - \beta)(x - \gamma)$$

の形に因数分解可能である。$\alpha\ \beta\ \gamma$ は定数だが、この三つを置換しても式は同じになる。つまり展開すれば同じ式になる。つまり、順列の群である対称群が対応することになある。f(x) 因数定理による分解

$$f(x) \Rightarrow (x - \alpha)(g\ x)$$

は、f(x) の対称群から g(x) の対称群への変換になる。具体的には三次の対称群から二次の対称群の変換がなければならない。ここで二次式に帰着できれば変換があり、変換があれば三次式を二次式に変換できる。

三次の対称群、つまり三つの数の入れ替えは、$\alpha\ \beta\ \gamma$ は γ .. と . γ . と .. γ からなるが、三次式を二次式に変換できるなら、$\alpha\ \beta\ \gamma$ と $\beta\ \alpha\ \gamma$ の二つの形にならないとおかしい。つまり、

$$\alpha\ \gamma\ \beta\ =\ \alpha\ \beta\ \gamma$$

と γ を移動できる必要がある。ということは、

$$\gamma\ \beta = \beta\ \gamma$$

である必要がある。逆元をかけて

$$e = \beta^{-1}\ \gamma^{-1}\ \beta\ \gamma$$

になれば良い (これは正確な話ではない)。左辺を Commutator(交換子) と呼び $[\beta, \gamma]$ と書く。

ある群の二つの要素を取って交換子に変換したものを交換子群と呼ぶ。これは群から群への関手になる。交換子群を繰り返し作り、単元だけの群になる場合を

Solvable 可解と呼ぶ。多項式に対応する対称群が可解であることがべき根を使って解けるための条件になるとされている。今回は対称群が可解であることがが方程式がべき根で解ける条件であることには関与せず、2-5 次の対称群が可解かどうかを計算で示す。

3 Agda による Group の表現

群は階層的な構造をもっており、演算と同値関係を持つ Magma, 合同則を持つ Semigroup, 結合則がある Monoid, そして逆元をもつる Group という順序で作成する。ここで作るのは対称群なので順列を要素にする。

Agda は Haskell と異なり、直積である record と Sum 型である data を区別する。record は field を持つオブジェクトに相当する。IsGroup は別に定義されていて、そこに群の公理が記述されている。**record** が record の構成子になる。

Agda の型には Level があり、基本型の後に自然数で指定する。⊔ は Level の最大値演算子である。: の前が名前で、後ろが型である。Carrier は要素の型であり、関係と演算子が **Rel Carrier ℓ, Op Carrier** と定義されているが、これらは **Carrier → Carrier → Set ℓ** と **Carrier → Carrier → Carrier** のことである。Agda は Haskell と同じく indent で構文のブロックを表している。

```
record Group c ℓ : Set (suc (c ⊔ ℓ)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier ℓ
    _•_     : Op Carrier
    ε       : Carrier
    _⁻¹     : Op Carrier
    isGroup : IsGroup _≈_ _•_ ε _⁻¹
  open IsGroup isGroup public
  monoid : Monoid _ _
  monoid = record { isMonoid = isMonoid }
  open Monoid monoid public
    using (rawMagma; magma; semigroup; rawMonoid)
```

群の構成要素は field で定義されているが、Monoid の公理 isMonoid などは module で定義されている。群を定義するにはこれらの field を適切に埋めれば良い。

順列は Agda に

```
Data.Fin.Permutation
```

というのがあるので、それを使う。これは Data.Fin の間の Bijection として 4 抽象的に定義されている。中括弧は省略可能な引数である。置換 p には二つの写像が中置演算子 **⟨$⟩,⟨$⟩ʳ** として定義されている。record の中には Bijection の性質が入っている。

```
-- Data.Fin.Permutation.id
pid : {p : ℕ } → Permutation p p
pid = permutation ( λ x → x)( λ x → x) record
  { left-inverse-of = λ x → refl
  ; right-inverse-of = λ x → refl }

-- Data.Fin.Permutation.flip
pinv : {p : ℕ } → Permutation p p → Permutation p p
pinv {p} P = permutation (_⟨$⟩ˡ_ P) (_⟨$⟩ʳ_ P ) record
  { left-inverse-of = λ x → inverseʳ P
  ; right-inverse-of = λ x → inverseˡ P }
```

という感じで簡単に逆元と単位元が求まる。refl は data である **x ≡ x** を生成する constructor である。

置換の同値性は写像が等しければよい。これを record で定義する。

```
record _=p=_ {p : ℕ } ( x y : Permutation p p ) : Set where
  field
    peq : ( q : Fin p ) → x ⟨$⟩ʳ q ≡ y ⟨$⟩ʳ q
```

これらを使って置換の群を以下のように定義できる。(presp と passoc などはここでは記述していない)

```
Symmetric : ℕ → Group  Level.zero Level.zero
Symmetric p = record {
    Carrier       = Permutation p p
  ; _≈_           = _=p=_
  ; _•_           = _∘_
  ; ε             = pid
  ; _⁻¹           = pinv
  ; isGroup = record { isMonoid  = record {
    isSemigroup = record { isMagma = record {
     isEquivalence = record {refl = prefl
       ; trans = ptrans ; sym = psym }
    ; •-cong = presp }
    ; assoc = passoc }
    ; identity = ( ( λ q → record { peq = λ q → refl } )
       , ( λ q → record { peq = λ q → refl } )) }
    ; inverse   = ( ( λ x → record { peq = λ q → inverseʳ x} )
       , ( λ x → record { peq = λ q → inverseˡ x} ))
    ; ⁻¹-cong   = λ i=j → record { peq = λ q → p-inv i=j q }
    }
  }
```

例えば identity は (ε • x) =p= x だが、これは x $\langle qx \rangle^{r}$ q なので、refl で生成できる。

4 Data.Fin

Agda は単なる自然数ではなく、有限な自然数を Sum 型として data 表されている。これが証明を持つデータ構造の最初の例になっている。

```
data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} (i : Fin n) → Fin (suc n)
```

zero と suc は Nat と重なっているので Monomorphic な Agda では注意が必要である。このデータ構造は基本的には Nat と同じだが、**Fin zero** は作れない。**# 1** と書く記法が用意されているが、**n** が決まるように記述する必要がある。つまり、**# 1** でも n が異なると別物になる。**to**ℕ で自然数を取り出すことができる。**Fin n** を作るためには、それが n よりも小さい証明が必要である。**{n : ℕ}** 省略可能で型推論により自動的に決まる。決まらなければ Agda が黄色で instantiation が足りないと警告される。証明では黄色を全部消す必要がある。計算自体は値が決まってなくても計算可能なところまで進む。これは関数型言語の特徴である。

Fin は n よりも小さいという性質をデータ構造として持っているので、それを証明により取り出すことができる。

```
-- toℕ≤n
fin≤n : {n : ℕ} (f : Fin (suc n)) → toℕ f ≤ n
fin≤n {_} zero = z≤n
fin≤n {suc n} (suc f) = s≤s (fin≤n {n} f)
```

ここで、**s≤s** と **z≤n** は不等式ををを表す data である。つまり、この証明は Fin から ≤ へのデータ構造の変換になっている。

5 交換子

対称群が定義できたので交換子を定義する。

```
[_,_] :  Carrier  → Carrier  → Carrier
[ g , h ] = g ⁻¹ • h ⁻¹ • g • h
```

このように、Unicode ってほぼ数式として定義できるところが Agda の特徴である。Coq[2] と異なる部分でもある。

この形で生成されるものを繰り返し生成したい。そのための生成演算子を以下のように定義する。

```
data Commutator (P : Carrier → Set (Level.suc n ⊔ m))
     : (f : Carrier) → Set (Level.suc n ⊔ m) where
  comm  : {g h : Carrier} → P g → P h  → Commutator P [ g , h ]
  ccong : {f g : Carrier} → f ≈ g → Commutator P f → Commutator P g
```

: の前の引数 **(P : Carrier → Set (Level.suc n ⊔ m))** は前回生成した時の条件である。ここに前の **Commutator** を入れる。: の後の引数はこの data の出力であり、生成された要素になる。これは comm で作成した時に作らている。ccong がないと、Commutator の等式が処理できない。条件がない時には ⊤ を用いる。これは唯一つの生成演算子を持つ Data.Unit つまり Singleton である。

```
data ⊤ : Set where
  tt : ⊤
```

という定義を持つ。交換子は積に付いて閉じてないので、交換子群を作る場合には以下の生成子も必要になる。しかし、今回は単位元だけになるかどうかを知りたいだけなので、これは使わない。

```
gen  : {f g : Carrier} → Commutator P f → Commutator P g
     → Commutator P ( f •  g  )
```

交換子の生成を繰り返すのが次の述語 deriving である。。これも Set (Level.suc n ⊔ m) を返してる。つまり証明するべき命題を返すような関数になっている。証明はもちろん、Commutator で作られるλ項であり、⊤ の構成子 tt で終わる。

```
deriving : ( i : ℕ ) → Carrier → Set (Level.suc n ⊔ m)
deriving 0 x = Lift (Level.suc n ⊔ m) ⊤
deriving (suc i) x = Commutator (deriving i) x
```

これを dervied-length 回繰り返すと、単位元のみになるというのは以下の record で表される。これが可解の定義になる。

```
record solvable : Set (Level.suc n ⊔ m) where
 field
   dervied-length : ℕ
   end : (x : Carrier ) → deriving dervied-length x →  x ≈ ε
```

dervied-length 回繰り返すと、単位元のみになるという数学的構造である。この時に、**dervied-length :** ℕ は存在記号が付いていると思って良い。

record を定義する以下のような記法で Symmetric 2 が solvable であることを表す。

```
sym2solvable : solvable (Symmetric 2)
solvable.dervied-length sym2solvable = 1
solvable.end sym2solvable x d = ?
```

この ? を埋めるのが今回の仕事になる。5 次対称群は可解でないので、以下を証明する。

```
counter-example :  ¬ (abc 0<3 0<4  =p= pid )
counter-example = ?

 end5 : (x : Permutation 5 5) → deriving (dervied-length sol) x
    →  x =p= pid
 end5 = ?

¬sym5solvable : ¬ ( solvable (Symmetric 5) )
¬sym5solvable sol = counter-example (end5 (abc 0<3 0<4 ) ? )
```

ここで **¬ A = A → ⊥** であり、⊥ は生成子のない data である。sol には仮定した solvable が来る。**abc 0<3 0<4** は 5 つの Fin のうち、a を b に、b を c に、c を a に置換する置換である。これが何回交換子を作っても残ってしまうことはアルティンの邦訳には以下のような証明が載っている。基本的にこれを証明することになる。

abc 以外の二つの要素とかを Agda では具体的に指摘する必要があり、これが一般的に成立することも示す必要がある。Agda によれば

```
全部証明しろ
```

ということになる。

6　Permutation に対応するデータ構造

有限な自然数 Fin の一対一対応 Bijection が Permutation だが 関数なので見えない。これを数え上げる必要があるので、もっと具体的なものが望ましい。単なるリストでも良いのだが、長さが決まらなかったり、同じ数字が入ったり、数字が足りなかったりで都合が悪い。そこで Permutation を以下の data 構造に対応させる。

```
data  FL : (n : ℕ )→ Set where
  f0 :  FL 0
  _::_ : { n : ℕ } → Fin (suc n ) → FL n → FL (suc n)
```

Fin の減少列になる。最初の挿入の選択肢はゼロ、次は一つ、次は二つなどとなる。

```
   ((# 3) :: ((# 1) :: ((# 0 ) :: f0)))
```

この List に大小関係を入れる。

```
data _f<_ : {n : ℕ } (x : FL n ) (y : FL n)  → Set  where
  f<n : {m : ℕ } {xn yn : Fin (suc m) } {xt yt : FL m}
    → xn Data.Fin.< yn →   (xn :: xt) f< ( yn :: yt )
  f<t : {m : ℕ } {xn : Fin (suc m) } {xt yt : FL m}
    → xt f< yt →   (xn :: xt) f< ( xn :: yt )
```

これは順列と 1 対 1 対応するので都合が良い。

7　Permutation と FL n の対応

FL n が置換に対応することは証明する必要がある。ここでは順列の combinator を使う。

```
FL→perm  : {n : ℕ }  → FL n → Permutation n n
FL→perm f0 = pid
FL→perm (x :: fl) = pprep (FL→perm fl)  ∘ pins ( toℕ≤pred[n] x )

perm→FL  : {n : ℕ }  → Permutation n n → FL n
perm→FL {zero} perm = f0
perm→FL {suc n} perm = (perm ⟨$⟩ʳ (# 0)) :: perm→FL (
   shrink (perm ∘ flip (pins (toℕ≤pred[n] (perm ⟨$⟩ʳ (# 0))))) (p=0 perm) )
```

pprep は先頭に 0 を付加する。**0 :: 1 :: 2 :: []** が 0 :: 1 :: 2 :: 3 :: [] になる。pins は 0 を指定した位置に挿入する。**0 :: 1 :: 2 :: 3 :: []** が **1 :: 2 :: 0 :: 3 :: []** にするような順列に対する演算である。shirink は pins の逆演算で 0 の位置を指定する p=0 を使っている。

```
shrink : {n  : ℕ} → (perm : Permutation (suc n) (suc n) )
    → perm ⟨$⟩ˡ (# 0) ≡ # 0 → Permutation n n
```

つまり、その位置の置換先が 0 でないと shrink は呼び出せない。pins/shrink は Bijection を構成するのでやや複雑。

問題は、FL→perm と perm→FL が Bijection であることを示すことだが、

```
shrink-iso : { n : ℕ } → {perm : Permutation n n}
    → shrink (pprep perm)  refl =p=  perm
shrink-iso2 : { n : ℕ } → {perm : Permutation (suc n) (suc n)}
   → (p=0 : perm ⟨$⟩ˡ (# 0) ≡ # 0)  → pprep (shrink perm p=0) =p= perm
```

が比較的簡単に示せるのでそれを使って証明できる。

8　実際の証明

数え上げは FL n でおこなうのだが、

```
p0id :  FL→perm ((# 0) :: ((# 0) :: ((# 0 ) :: f0))) =p= pid
```

を示そうと思うとかなり面倒なことになる。しかし、ここで equalizer を使うと楽になる。置換は数字の列で表される。その列が等しければ、元の置換も等しいことが証明できる。

```
pleq  : {n  : ℕ} → (x y : Permutation n n ) → plist0 x ≡ plist0 y → x =p= y
```

これを使って

```
p0id :  FL→perm ((# 0) :: ((# 0) :: ((# 0 ) :: f0))) =p= pid
p0id = pleq _ _ refl
```

という風に簡単に証明できる。plist0 x は具体的なλ項どうしの比較だから refl で良い。pleq の証明は plist0 の algorythm を二つの項 x, y で同時にたどって、入力 q : Fin n に対して (x ⟨q⟩(y)$^{\mathrm{r}}$ q) を示していけばよい。

3 次では

```
p3 =  FL→perm ((# 1) :: ((# 1) :: ((# 0 ) :: f0)))
p4 =  FL→perm ((# 2) :: ((# 0) :: ((# 0 ) :: f0)))
st02 :  ( g h : Permutation 3 3) →
  ([ g , h ] =p= pid) ∨ ([ g , h ] =p= p3) ∨ ([ g , h ] =p= p4)
```

になるのだが、これを 3 次対称群の要素全部に付いて確認する必要がある。図 1 というように 50 行程度書く必要がある。この証明チェックには数分かかる。

9　5 次対称群

abc は pins などで定義できる。

```
--- 1 :: 2 :: 0 :: []     abc
3rot : Permutation 3 3
3rot = pid {3}∘ pins (n≤ 2)
```

5 次の中の 3 次なので二つ空き場所がある。それを不等式で指定する。不等式なのは Fin を指定するのに便利だからである。

```
abc : {i j : ℕ }  → (i ≤ 3 ) → ( j ≤ 4 ) → Permutation  5 5
```

これを交換子から生成してやればよい。**[dba , aec]** なのだが、場所を正確に指定する必要がある。

```
record Triple {i j : ℕ } (i<3 : i ≤ 3) (j<4 : j ≤ 4)
     (rot : Permutation 3 3) : Set where
  field
    dba0<3 : Fin 4
    dba1<4 : Fin 5
    aec0<3 : Fin 4
    aec1<4 : Fin 5
    abc= : ins2 rot i<3 j<4 =p=
      [ ins2 (rot ∘ rot)  (fin≤n {3} dba0<3) (fin≤n {4} dba1<4)
      , ins2 (rot ∘ rot) (fin≤n {3} aec0<3) (fin≤n {4} aec1<4) ]
```

これをすべての場所に付いて record を作成する。これがすべての組み合せについて記述される。図 1 の 10 行程度である中の値は自分で計算する必要がある。

これだけでは閉じてなくて、もう一種類必要になる。3 次の置換には右回転と左回転があり、右回転は左回転の交換子で作るためである。この対称性は置換のレベルでも FL n のレベルでも自明には見えない。なので、別々に手計算する必要がある。

```
dba-triple : {i j : ℕ }  → (i<3 : i ≤ 3 ) → (j<4 :  j ≤ 4 )
  → Triple i<3 j<4 (3rot ∘ 3rot )
```

5 次対称群の中で 3 次の置換は二種類にわかれて存在していて、それらが交互に交換子を生成するようになっている。

```
dervie-any-3rot0 : (n : ℕ ) → {i j : ℕ }  → (i<3 : i ≤ 3 ) → (j<4 : j ≤ 4 )
    → deriving n (abc i<3 j<4 )
dervie-any-3rot1 : (n : ℕ ) → {i j : ℕ }  → (i<3 : i ≤ 3 ) → (j<4 : j ≤ 4 )
    → deriving n (dba i<3 j<4 )
```

この二つが相互参照している構造になっている。本の証明ではそれを気にする必要はないが、Agda は出目を気にするので必要である。

このチェックにも時間はかかるが、3 次対称群ほどではない。

しかし、この方法では 4 次対称群の可解性を示すのは厳しい。そこで、Sorted List である Fresh List [3] を使う。

10　Fresh List

各要素の大小関係がすべて入っている sort された List。これを使うと重複要素を排除できる。

```
data List# : Set (a ⊔ r)
fresh : (a : A) (as : List#) → Set r

data List# where
  []  : List#
  cons : (a : A) (as : List#) → fresh a as → List#

infixr 5 _∷#_
pattern _∷#_ x xs = cons x xs _

fresh a []       = ⊤
fresh a (x ∷# xs) = R a x × fresh a xs
```

これを使うと

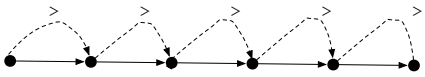
```
  FList : {n : ℕ } → Set
  FList {n} = List# (FL n) ⌊ _f<?_ ⌋

fr1 : FList
fr1 =
  ((# 0) :: ((# 0) :: ((# 0 ) :: f0))) ∷#
```

```
st02 :  ( g h : Permutation 3 3) →  ([ g , h ] =p= pid) ∨ ([ g , h ] =p= p3) ∨ ([ g , h ] =p= p4)
st02 g h with perm→FL g | perm→FL h | inspect perm→FL g | inspect perm→FL h
... | (zero :: (zero :: (zero :: f0))) | t | record { eq = ge } | te = case1 (ptrans (comm-resp {g} {h} {pid} (FL-inject ge ) prefl ) (idcomtl h) )
... | s | (zero :: (zero :: (zero :: f0))) | se |  record { eq = he } = case1 (ptrans (comm-resp {g} {h} {_} {pid} prefl (FL-inject he ))(idcomtr g)\
)
... | (zero :: (suc zero) :: (zero :: f0 )) |  (zero :: (suc zero) :: (zero :: f0 )) |  record { eq = ge } |  record { eq = he } =
    case1 (ptrans (comm-resp (pFL g ge) (pFL h he)) (FL-inject refl) )
... | (suc zero) :: (zero :: (zero :: f0 )) | (suc zero) :: (zero :: (zero :: f0 )) |  record { eq = ge } |  record { eq = he } =
    case1 (ptrans (comm-resp (pFL g ge) (pFL h he)) (FL-inject refl) )
... | (suc zero) :: (suc zero :: (zero :: f0 )) |  (suc zero) :: (suc zero :: (zero :: f0 )) |  record { eq = ge } |  record { eq = he } =
    case1 (ptrans (comm-resp (pFL g ge) (pFL h he)) (FL-inject refl) )
... | (suc (suc zero)) :: (zero :: (zero :: f0 )) | (suc (suc zero)) :: (zero :: (zero :: f0 )) | record { eq = ge } |  record { eq = he } =
    case1 (ptrans (comm-resp (pFL g ge) (pFL h he)) (FL-inject refl) )
...
```

Figure 1: 3 次対称群のめんどくさい計算

```
open Triple
triple : {i j : ℕ } → (i<3 : i ≤ 3) (j<4 : j ≤ 4) → Triple i<3 j<4 3rot
triple z≤n z≤n =                          record { dba0<3 = # 0 ; dba1<4 = # 4 ; aec0<3 = # 2 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple z≤n (s≤s z≤n) =                      record { dba0<3 = # 0 ; dba1<4 = # 4 ; aec0<3 = # 2 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple z≤n (s≤s (s≤s z≤n)) =                  record { dba0<3 = # 1 ; dba1<4 = # 0 ; aec0<3 = # 3 ; aec1<4 = # 2 ; abc= = pleq _ _ refl }
triple z≤n (s≤s (s≤s (s≤s z≤n))) =              record { dba0<3 = # 1 ; dba1<4 = # 3 ; aec0<3 = # 0 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple z≤n (s≤s (s≤s (s≤s (s≤s z≤n)))) =          record { dba0<3 = # 0 ; dba1<4 = # 0 ; aec0<3 = # 2 ; aec1<4 = # 4 ; abc= = pleq _ _ refl }
triple (s≤s z≤n) z≤n =                      record { dba0<3 = # 0 ; dba1<4 = # 2 ; aec0<3 = # 3 ; aec1<4 = # 1 ; abc= = pleq _ _ refl }
triple (s≤s z≤n) (s≤s z≤n) =                  record { dba0<3 = # 0 ; dba1<4 = # 2 ; aec0<3 = # 3 ; aec1<4 = # 1 ; abc= = pleq _ _ refl }
triple (s≤s z≤n) (s≤s (s≤s z≤n)) =              record { dba0<3 = # 1 ; dba1<4 = # 0 ; aec0<3 = # 3 ; aec1<4 = # 2 ; abc= = pleq _ _ refl }
triple (s≤s z≤n) (s≤s (s≤s (s≤s z≤n))) =          record { dba0<3 = # 0 ; dba1<4 = # 3 ; aec0<3 = # 1 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple (s≤s z≤n) (s≤s (s≤s (s≤s (s≤s z≤n)))) = record { dba0<3 = # 2 ; dba1<4 = # 4 ; aec0<3 = # 0 ; aec1<4 = # 2 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s z≤n)) z≤n =                  record { dba0<3 = # 3 ; dba1<4 = # 0 ; aec0<3 = # 1 ; aec1<4 = # 3 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s z≤n)) (s≤s z≤n) =              record { dba0<3 = # 3 ; dba1<4 = # 0 ; aec0<3 = # 1 ; aec1<4 = # 3 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s z≤n)) (s≤s (s≤s z≤n)) =          record { dba0<3 = # 1 ; dba1<4 = # 3 ; aec0<3 = # 0 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s z≤n)) (s≤s (s≤s (s≤s z≤n))) = record { dba0<3 = # 0 ; dba1<4 = # 3 ; aec0<3 = # 1 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s z≤n)) (s≤s (s≤s (s≤s (s≤s z≤n)))) = record { dba0<3 = # 1 ; dba1<4 = # 4 ; aec0<3 = # 2 ; aec1<4 = # 2 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s (s≤s z≤n))) z≤n =              record { dba0<3 = # 2 ; dba1<4 = # 4 ; aec0<3 = # 1 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s (s≤s z≤n))) (s≤s z≤n) =          record { dba0<3 = # 2 ; dba1<4 = # 4 ; aec0<3 = # 1 ; aec1<4 = # 0 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s (s≤s z≤n))) (s≤s (s≤s z≤n)) = record { dba0<3 = # 0 ; dba1<4 = # 0 ; aec0<3 = # 2 ; aec1<4 = # 4 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s (s≤s z≤n))) (s≤s (s≤s (s≤s z≤n))) = record { dba0<3 = # 2 ; dba1<4 = # 4 ; aec0<3 = # 0 ; aec1<4 = # 2 ; abc= = pleq _ _ refl }
triple (s≤s (s≤s (s≤s z≤n))) (s≤s (s≤s (s≤s (s≤s z≤n)))) =
                                  record { dba0<3 = # 1 ; dba1<4 = # 4 ; aec0<3 = # 0 ; aec1<4 = # 3 ; abc= = pleq _ _ refl }
```

Figure 2: 5 次対称群のめんどくさい計算の半分



```
((# 0) :: ((# 1) :: ((# 0 ) :: f0))) ::#
((# 1) :: ((# 0) :: ((# 0 ) :: f0))) ::#
((# 2) :: ((# 0) :: ((# 0 ) :: f0))) ::#
((# 2) :: ((# 1) :: ((# 0 ) :: f0))) ::#
[]
```

などとできる。Sort されてないものを定義することはできない。

fresh は Set 、つまり命題を返す形なので、値が十分に決まらないと証明するものもわからない。

```
fr6 = FLinsert ((# 1) :: ((# 1) :: ((# 0 ) :: f0))) fr1
```

とできれば良い。FLinsert は単なる線形挿入だが、fresh list を証明する必要がある。

```
FLinsert : {n : ℕ } → FL n → FList n  → FList n
FLfresh : {n : ℕ } → (a x : FL (suc n) ) → (y : FList (suc n) ) → a f< x
    → fresh (FL (suc n)) ⌊ _f<?_ ⌋ a y
    → fresh (FL (suc n)) ⌊ _f<?_ ⌋ a (FLinsert x y)
FLinsert {zero} f0 y = f0 ::# []
FLinsert {suc n} x [] = x ::# []
FLinsert {suc n} x (cons a y x) with FLcmp x a
... | tri≈ ¬a b ¬c  = cons a y x
... | tri< lt ¬b ¬c  = cons x ( cons a y x)
      ( Level.lift (fromWitness lt ) , ttf lt y  x)
FLinsert {suc n} x (cons a [] x) | tri> ¬a ¬b lt  =
    cons a ( x  ::# []  ) ( Level.lift (fromWitness lt) , Level.lift tt )
```

```
FLinsert {suc n} x (cons a y yr) | tri> ¬a ¬b a<x =
    cons a (FLinsert x y) (FLfresh a x y a<x yr )
```

という形で、FLinsert と FLfresh で相互再帰していく。FLfresh は省略している。

⌊ _f<?_ ⌋ は_f<?_のままでは Set で扱いづらいので、それが真なのか偽なのかを表す data である。fromWitness や toWitness で変換する。中には不等号を表す data 構造が入る。fresh の base は ⊤ なので **Level.lift tt** が最後に来る。**R a x × fresh a xs** はみづらいが、直積で不等号と、後に続く fresh の条件リストがある。

11 Permuation の数え上げ

全部の Permutation を Fresh List にする。全部入っていることをデータ構造として表すには以下の data を使う。

```
data Any : List# A R → Set (p ⊔ a ⊔ r) where
  here  : ∀ {x xs pr} → P x → Any (cons x xs pr)
  there : ∀ {x xs pr} → Any xs → Any (cons x xs pr)
```

ここにある、あそこにあるみたいな感じである。

```
AnyFList : (n : ℕ )  → (x : FL n ) →  Any (x ≡_ ) (∀Flist fmax)
```

が示せれば、∀Flist fmax に全部の FL n が入っていることがわかる。

```
x∈FLins : {n : ℕ} → (x : FL n ) → (xs : FList n)
    → Any (x ≡_ ) (FLinsert x xs)
insAny : {n : ℕ} → {x h : FL n } → (xs : FList n)
    → Any (x ≡_ ) xs → Any (x ≡_ ) (FLinsert h xs)
```

などが証明できるのでこれを使う。

12 数え上げの方法

FL n は一つ小さいリストの前に、0 から n までを付け加えることで数え上げられる。この方法だと fresh を作るのに相互再帰は必要ない。しかし、このままでは Agda が停止条件を見つけられなので、**{-# TERMINATING #-}** を付けている。

```
{-# TERMINATING #-}
AnyFList : {n : ℕ }  → (x : FL n ) →  Any (x ≡_ ) (∀Flist fmax)
AnyFList {zero} f0 = here refl
AnyFList {suc zero} (zero :: f0) = here refl
AnyFList {suc (suc n)} (x :: y) = subst ( λ  k → Any (_≡_ (k :: y))
   (Flist (suc n) a<sa (∀Flist fmax) (∀Flist fmax) ))
      (fromℕ<-toℕ _ _) ( AnyFList1 (suc n) (toℕ x) a<sa (∀Flist fmax)
          (∀Flist fmax) fin<n fin<n (AnyFList y) (AnyFList y)) where
 AnyFList1 :  (i x : ℕ) → (i<n : i < suc (suc n) ) →
   (L L1 : FList (suc n) ) → (x<n : x < suc (suc n) ) → x < suc i
     → Any (y ≡_ ) L → Any (y ≡_ ) L1
     → Any (((fromℕ< x<n) :: y) ≡_ ) (Flist i i<n L L1)
 AnyFList1 = ?
```

以下の部分で

```
AnyFList1 (suc i) x (s≤s i<n) (cons a (cons a L x) x)
   L1 x<n (s≤s x<i) (there wh) any with FLcmp a a
... | tri< a ¬b ¬c = insAny _ (
   AnyFList1 (suc i) x (s≤s i<n) (cons a L x) L1 x<n (s≤s x<i) wh any )
```

(cons a (cons a L x) x) は **(cons a L x)** と小さくなってるのだが、Agda はそれを認識してくれない。

13 Commutator の数え上げの方法

さらに、交換子の生成を数え上げる必要がある。差分リストを使えば

```
tl3 : (FL n) → ( z : FList n) → FList n → FList n
tl3 h [] w = w
tl3 h (x ::# z) w = tl3 h z
  (FLinsert ( perm→FL [ FL→perm h , FL→perm x ] ) w )
tl2 : ( x z : FList n) → FList n →  FList n
tl2 [] _ x = x
tl2 (h ::# x) z w = tl2 x z (tl3 h z w)

CommFList  :  FList n →  FList n
CommFList x =  tl2 x x []

CommFListN  : ℕ  →  FList n
CommFListN  0 = ∀Flist fmax
CommFListN  (suc i) = CommFList (CommFListN i)
```

数え上げ自体は簡単で、すべての組の交換子を作って FLinsert すればよい。そのために FLinsert を作ったのだった。

それが正しく、すべての組み合せを含んでいるかを示す必要がある。

```
CommStage→ : (i : ℕ) → (x : Permutation n n )
  → deriving i x → Any (perm→FL x ≡_) ( CommFListN i )
```

つまり、deriving i x ならば、それは CommFListN i に含まれているというわけである。これを示すには、CommFListN を i にそって deriving i x と一緒に分解していく。あとは、tl2 と tl3 が特定の組み合せを含むことを調べに行く。

```
CommStage→ zero x (Level.lift tt) = AnyFList (perm→FL x)
CommStage→ (suc i) .( [ g , h ] ) (comm {g} {h} p q) =
 comm2 (CommFListN i) (CommFListN i) (CommStage→ i g p)
    (CommStage→ i h q) [] where
  G = perm→FL g
  H = perm→FL h
  -- tl2 case
  comm2 : (L L1 : FList n) → Any (G ≡_) L
     → Any (H ≡_) L1 → (L3 : FList n)
     → Any (perm→FL [ g , h ] ≡_) (tl2 L L1 L3)
  comm2  = ?
  -- tl3 case
  commc :  (L3 L1 : FList n)
    →  Any (_≡_ (perm→FL [  FL→perm G , FL→perm H ])) L3
    →  Any (_≡_ (perm→FL [ FL→perm G , FL→perm H ])) (tl3 G L1 L3)
  commc = ?
```

この時に、

```
  comm8 : (L L4 L2 : FList n) → (a : FL n)
     → Any (_≡_ (perm→FL [ g , h ])) (tl2 L4 L1 L2)
     → Any (_≡_ (perm→FL [ g , h ])) (tl2 L4 L1 (tl3 a L L2))
  comm8← : (L L4 L2 : FList n) → (a : FL n)  → ¬ ( a ≡ perm→FL g )
     →  Any (_≡_ (perm→FL [ g , h ])) (tl2 L4 L1 (tl3 a L L2))
     →  Any (_≡_ (perm→FL [ g , h ])) (tl2 L4 L1 L2)
```

などの交換法則が必要になる。つまり、g が a と関係なければ **(tl3 a L L2)** を移動して良い。

このアルゴリズムだと、一度、tl2 のそこまで潜らないと戻り値が確定しない。なので、交換則をつかって行きつ戻りつする必要がある。

おそらく、CommFListN と CommStage→ を同時に生成する見通しの良い方法があると思われる。

14　Fresh List を使った可解の検査

結局、**Any (perm→FL x ≡_) y → x =p= pid** を調べるには FList n が **FL0 ::# []** であることを確認すればよい。

```
CommSolved : (x : Permutation n n) → (y : FList n)
  → y ≡ FL0 ::# [] → (FL→perm (FL0 {n}) =p= pid )
  → Any (perm→FL x ≡_) y → x =p= pid
CommSolved x .(cons FL0 [] (Level.lift tt)) refl eq0 (here eq)
  = FLpid _ eq eq0

stage3FList : CommFListN 3 2 ≡
   cons (zero :: zero :: zero :: f0) [] (Level.lift tt)
stage3FList = refl

solved1 :  (x : Permutation 3 3) → deriving 2 x → x =p= pid
solved1 x dr = CommSolved 3 x ( CommFListN 3 2 )
      stage3FList p0id solved2 where
  solved2 : Any (perm→FL x ≡_) ( CommFListN 3 2 )
  solved2 = CommStage→ 3 2 x dr
```

と簡単に記述することができる。

15　実行時間

Fresh List を使う方法だと、3 次の場合でも 10 秒 (sym3n) でチェックされる。4 次でも 10 秒である。5 次も高速化される可能性がある。

```
agda sym3.agda  258.01s user 2.95s system 98% cpu 4:23.68 total
agda sym3n.agda  9.18s user 0.45s system 95% cpu 10.089 total
agda sym2n.agda  9.09s user 0.35s system 99% cpu 9.454 total
agda sym2.agda  9.34s user 0.50s system 94% cpu 10.448 total
agda sym4.agda  9.38s user 0.37s system 99% cpu 9.784 total
agda sym5.agda  9.04s user 0.34s system 99% cpu 9.427 total
```

sym5 を実際に計算して反例を探すこともできるのだが、型検査時の実行では停止を確認できなかった。しかし、Agda には Haskell 経由で実行バイナリを生成する機能があり、それを用いれば瞬時に計算は終わる。

```
./sym5n  0.00s user 0.01s system 37% cpu 0.044 total
```

16　Agda の証明付きコード

Agda は Haskell に近い構文を持つ、Haskell で実装された定理証明系であり、純関数型プログラミング言語でもある。Agda から実行コードを生成することもできる。証明に特化した Coq よりはプログラマにとって近いしいものかも知れない。

さまざまな計算が「いったい何を計算しているのか」をちゃんと把握することは重要だし、すべてのバグはつまらないバグなので、さまざまな理由でバグが混入する。そこで、計算しているものの仕様を記述して、それを証明するのは回り道であるが有効かも知れない。

一方で、実行時に証明のコードはまったくの無駄とも言える。一方で実行時に型検査する必要はないはずなので、オーバヘッドは存在しない。しかし、Fresh List は **Level.lift tt** を持つので、それは生成される。

```
cons (zero :: zero :: zero :: f0)
(cons (suc zero :: suc zero :: zero :: f0)
 (cons (suc (suc zero) :: zero :: zero :: f0) [] (Level.lift tt))
 (Level.lift tt , Level.lift tt))
(Level.lift tt , Level.lift tt , Level.lift tt)
```

のような形になる。これは実行時に生成されてしまう。これはオーバヘッドになる。ただし、sym5n のようにコンパイルしてしまえば、使われない部分は計算/生成

されない可能性がある。なので実行時のオーバヘッドはほとんどないかも知れない。

証明自体は線形挿入でもかなりめんどうでプログラミングの手間に見合うとは思えない。しかし、ライブラリは重要なので、その手間に見合う可能性もある。また、証明自体は大半の証明は簡単なので機械学習が使える可能性が高い。

置換の表現としての Bijection の扱いは面倒で、同値を調べるには、すべての要素に付いて写像が等しいことを調べる必要がある。一方で、入力を固定すれば、それは単なる定数関数となる。つまり、抽象的なものを具体的なものに置き換えられれば、話は簡単になる。あらゆる抽象的なものはゲーデル的な意味で具体的な構造物を持つはずなので、一般的にそういうことが可能かも知れない。

Agda での同値性は、三つの表現方法がある。一つは refl つまり、Agda のλ項として等しいことである。これは圏論の言葉で言う small に相当する。プログラミングで出てくるもののほんどは、この形ですむ。もう一つは data を使って Sum 型として表す方法がある。例えば、具体的な有限グラフはこの方法が適している。この方法だとエッジの不存在を簡単に証明できる。三つ目として record を持ちても良い。置換の結果が等しいとかはこの方式である。いずれの方式でも問題になるのはλ項の単一化 (Unification) である。

Agda は自然演繹なので推論規則自体は理解しやすい。難しさは単一化にあり、さまざまな問題が生じる。しかも、それはプログラミング上、証明上良く見えない。巨大な変数を含む項が生成されて、計算時間を消費する。現在の Agda はファイル単位での差分実行はあるが、一つのファイル内での差分実行はないので、重い証明が存在する。単一化の計算量は指数計算量で予測は難しい。

もう一つは停止性の問題である。停止性は無視することもできるが、もう少し制御したい。制御の方法は不明だが、必要なのは deduction つまり、減少列を明確にすることなので、外付けがそれを付加できると良い。

Any は任意の要素 x が L : Fresh List に入っていることを示す。これは L が x 集合の Initial Object つまり、L → x となる関数をすべて持っているということになっている。CommStage→ を示す時に使った可換性は Any が自然変換であることを意味している。この部分は証明なので実際の計算には関係ない。しかし、その部分では圏論的なアプローチが有効かも知れない。

コンパイルにより高速に計算が実行されるのは良いが、それを証明時、つまり型検査時に使用することは現状ではできない。Agda は型検査結果を .agdai ファイルに取っておくので、実行時の結果を .agdai に埋め込む方法が考えられる。そのためには、都合良く計算を引き伸ばすような実装が必要になる。

17 終わりに

Agda による対称群の可解性についての証明を行った。手計算では入力と検査時間がかかるが、Fresh List のような証明付きデータ構造を用いることにより検査時間を短縮し、記述も短くすることができた。一方で、リストへの挿入などのコードが複雑になってしまう。しかし、それにより Any のように、Fresh List に必要なものが全部入っていることを確認することができ、それをそのまま証明に使うことがきる。これにより、計算結果を直接に証明で使うことができるようになる。

五次対称群が可解群でない計算は FList を使ってない。deriving が不動点になるなら機械的に FList で証明できる。しかし、それは非可解であることの説明にはならない。abc が他の置換の交換子として表されるという理由はそこからは出てこない。

この例はプログラムの正しさを直接に証明としてコードに埋め込んだ例になっている。

Agda による証明は [6] で見ることができる。

References

[1] AgdaWiki. https://wiki.portal.chalmers.se/agda/main/homepage.

[2] The Coq Proof Assistance. https://coq.inria.fr.

[3] Catarina Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. Higher-Order and Symbolic Computation, Vol. 15, No. 1, pp. 57–90, Mar 2002.

[4] Samir Khuller. Four colors suffice! SIGACT News, Vol. 36, No. 2, p. 43â44, June 2005.

[5] Yoshiki Kinoshita. Agda language. Technical Report PS-2008-014, 独立行政法人 産業技術総合研究所, 2008.

[6] Shinji Kono. https://github.com/shinji-kono/galois.git, 2020.