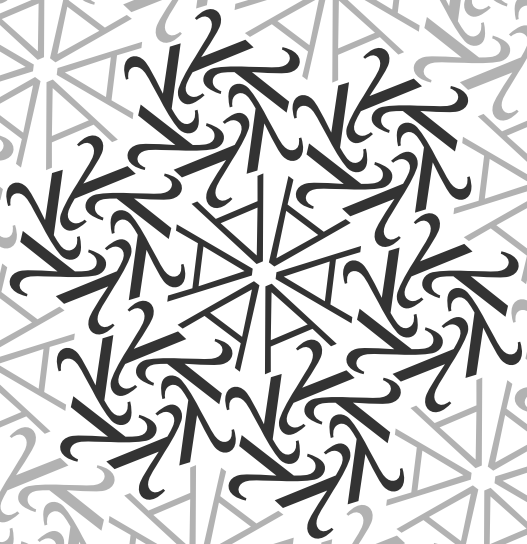
Coqによる定理証明

2013
12



坂口和彦
平井洋一
石井大海

Tsukuba Coq Users' Group

前書き

プログラムの性質や数学の定理の正当性を計算機の力で自動的にチェックしたい。そうすればプログラムのバグを確実に取り除くのは簡単になり、数学の論文に間違いが含まれていないか長い期間かけて調べる必要もない。本書は茨城県つくば市周辺の証明士が集まって書いたものであり、上に掲げた目的を実現するための様々な技法が詰まっている。

我々は Tsukuba Coq Users' Group という名前で活動しているが、その名前の通り本書の著者の多くは Coq ユーザである。Coq というのは計算機上で証明を記述するためのソフトウェアであり、フランス国立情報学自動制御研究所 (INRIA) を中心としていくつかのフランス国内の研究機関が共同で開発している。この種類のソフトウェアは定理証明支援系 (proof assistant) や定理証明器 (theorem prover) と呼ばれ、Coq 以外にも Isabelle, Agda, HOL, Mizar など様々な定理証明のためのシステムが存在している。Coq の特徴は、その理論的背景に型付き λ 計算があり依存型 (dependent types) を扱えること、対話的に証明を進める機能を持つこと、証明の自動化ができること、記述したプログラムを OCaml のプログラムに変換できることなどである。対話的な証明の記述にはタクティクと呼ばれる証明記述用の言語を用いる。Coq を使った有名な研究としては変換の前後でプログラムの意味を保存すると保証されている C コンパイラ CompCert [20] や、四色定理の証明、Feit-Thompson の定理の証明 [11] などが挙げられる。他にもプログラミング言語の型システムの性質の検証や C などの手続き型言語で記述されたプログラムの検証など、様々なことに応用されている。

もしあなたが計算機上で証明を書きプログラムの性質や数学の定理の証明を検証すること、特に Coq を使ってそれを実現することに興味があれば、この本はそのための良いヒントを含んでいるかもしれない。ただし、本書は Coq の使い方について基本的な事項を解説することを目的としていない。Coq のリファレンスマニュアル [25] やいくつかの教科書 [2, 4, 5, 23, 24] はそのような情報を補うのに役立つだろう。

SSReflect

SSReflect (small scale reflection) [10] とは、Microsoft Research と INRIA が開発している Coq の拡張とライブラリ集である。これは元々四色定理の証明の形式化のために開発されたものであり、現在は Feit-Thompson の定理の証明を含めた有限群の分類の形式化にその基盤として使われている。本書に掲載している Coq のコードのほとんどの部分は SSReflect を前提としている。

本書を執筆している時点での SSReflect の最新のリリースは 1.4 である。1.5 のリリース候補版 (1.5rc1) ではライブラリの一部分が取り除かれ、その部分は Mathematical Components Library という別のライブラリになっている。本書ではこれらを合わせて SSReflect と呼ぶことがある。

SSReflect の拡張の部分は、Coq の多くのタクティクの改良版や便利な構文、コマンドなどを提供する。改良されたタクティクを使うと、通常の Coq と比べて証明の保守性を高め [*1]、証明を簡潔に記述することが可能になる。ライブラリの部分は、Coq の標準ライブラリの一部分の代替、有限グラフに関する計算と補題集、有限長の列に関する $\sum, \prod$ などの「大きい演算子」を一般化した計算、有限群などの代数の形式化を含んでいる。

必要なソフトウェアのインストール

本書に掲載されている Coq のコードは、全て Coq 8.4pl2, SSReflect 1.5rc1, Mathematical Components Library 1.5rc1 の上で検証している。Coq と SSReflect のビルドには OCaml と Camlp5[*2] が必要である。以下でそれらのソフトウェアのビルド、インストール手順を説明する。また、以下のインストール手順は全て下にあるものが上に依存する関係にあるので、上から順番に作業を進める必要がある。

[*1] より具体的には、Coq やライブラリのバージョン、定義などが変わって証明が通らなくなったときに、その原因を特定しやすくなる。

[*2] Coq 自体はつい最近 Camlp4 にも対応するようになったが、SSReflect をビルドするには今でも Camlp5 が必要である。

目次

1 乱数列の網羅性

坂口 和彦

1.1 はじめに

この章では、**線形合同法** (*linear congruential generators*, *LCGs*) という疑似乱数列を生成するアルゴリズムについて、それが生成する乱数列が網羅的であることの条件の証明を試みる。結論から言うと証明はまだ未完成であるが、参考にした証明の多くの部分を形式化できた。この章の全ての内容を理解するには Coq と SSReflect に習熟していることが必須だが、形式化を簡単に進めるための問題の読み替えなどの技法は、一般的な数学の知識と Coq に関する基礎的な知識だけで理解できるように努めた。

線形合同法の形式化とその性質の証明には、以下のライブラリを使用した。

```
Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq choice fintype div
               path bigop prime binomial ssralg zmodp.
Import GRing.Theory.
```

証明の途中で記述する項に余計なプレースホルダー (_) を書いたり暗黙の引数を明示的に書いたりするのをなるべく避けるため、また Coq が表示する項を読みやすくするために、以下の設定をする。

```
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

これらの設定は SSReflect のライブラリの実装の中でも良く使われている。意味は良く分からなくても、なんとなく暗黙の引数が扱いやすくなるというつもりで普段書いている証明の先頭に書いておくのも悪くないだろう。

この章では、表 1.1 に挙げる数式上の表記法を導入する。

1.8　補題 P

補題 P のステートメントは以下の通りである。

$$\forall p \in \mathbb{P}, x \in \mathbb{N}, e > 0 \,.\, 2 < p^e \implies$$
$$p^e \mid x \wedge p^{e+1} \nmid x \implies p^{e+1} \mid (x+1)^p - 1 \wedge p^{e+2} \nmid (x+1)^p - 1$$

さて、この命題を見ると、$p^e \mid x \wedge p^{e+1} \nmid x$ という形をした部分が二箇所ある。これは直感的には「x が素数 p でちょうど e 回割り切れる」という意味であるが、このちょうど何回割り切れるかを計算する関数 `logn` が用意されている [*6]。これを使うと、上の命題を以下のように変形できる。

$$\forall p \in \mathbb{P}, x \in \mathbb{N}, e > 0 \,.\, 2 < p^e \implies$$
$$\mathrm{logn}(p, x) = e \implies \mathrm{logn}(p, (x+1)^p - 1) = e + 1$$

さらに e を $\mathrm{logn}(p, x)$ で置き換え、$2 < p^{\mathrm{logn}(p,x)}$ を同値な命題 $\delta[p = 2] < \mathrm{logn}(p, x)$ で置き換えると、

$$\forall p \in \mathbb{P}, x \in \mathbb{N} \,.\, \delta[p = 2] < \mathrm{logn}(p, x) \implies$$
$$\mathrm{logn}(p, (x+1)^p - 1) = \mathrm{logn}(p, x) + 1$$

となる。これの証明は、以下の通りである。

```
Lemma LemmaP p x :
  prime p -> (p == 2) < logn p x -> logn p (x.+1 ^ p).-1 = (logn p x).+1.
Proof.
  move: p x => [| []] // p [| []]; rewrite ?logn0 ?logn1 // => x H H0.
  rewrite expSS addSn /= /index_iota subn1 /= !expnS /=
          big_cons bin1 expn1 (mulnC p.+2) (iota_addl 2 0) big_map.
  have/(eq_bigr _) -> i: true ->
      'C(p.+2, i.+2) * x.+2 ^ i.+2 = x.+2 * (x.+2 * ('C(p.+2, i.+2) * x.+2 ^ i))
    by rewrite !expnS 2!(mulnCA x.+2).
  rewrite -!big_distrr /= addnCA -!mulnDr lognM //= -[X in _ = X]addn1; f_equal.
  move: H0; rewrite lognE H /=; case: ifP => //.
  rewrite dvdn_eq; move/eqP/esym => H0 H1.
```

[*6] ただし、$\mathrm{logn}(p, 0) = 0$ である。また、素数でない n に対しては $\mathrm{logn}(n, m) = 0$ である。

2 Coq で memoize

平井 洋一

注意。この章のコードをそのまま打ち込んでも動かないので、行間を埋める覚悟がないばあいは、写経しないことをおすすめする。

2.1 幸福な時代の面白いゲーム

いまはなんとよい時代であろうか。とうとう証明が動くのである (まあ 20 世紀後半から証明は動くのであるけれども)。こういう面白い時代にあって、動く証明を書かずに暮らしている人は、おもしろいものを逃がしているか、もっとおもしろいものをしっているか、どちらかである。どちらだろう。もっとおもしろいことがあるなら教えてもらいたい気がする。ゲームとかかもしれない、Twitter にはゲームをしている人がたくさんいる。

この幸福な時代にあってゲーム [6] のことを考える。二人で交互に着手を繰り返し、着手できないほうが負けになるゲームのことを考える。二人のプレイヤーのことを Left と Right と呼ぶことにする。

のちのち、小さなゲームを組み合わせて大きなゲームを作りたい。大きなゲームでは Left と Right が交互に着手を繰り返すとしても、小さなゲームでは Left が連続して着手をすることもありうる。この状況に対応するためには各ゲームについて、Left に可能な着手と Right に可能な着手と両方判明する必要がある。計算をしたいので、ゲームは有限とする。つまり、選択肢は有限個だし、各プレイも有限長だし、それどころか、各ゲームに定数があって、二人のプレイヤーが協力してできる限りプレイを引き伸ばそうとしても、その定数を越えた数の着手をつくることはできないようにする [*1]。

[*1] 各プレイが有限でも、そのような定数が存在しない場合がある。たとえば、最初のプレイヤーが自然数を唱え、その自然数だけの着手を双方で行うような場合である。

3 Haskellによる定理証明

石井 大海

3.1 はじめに

この記事では、関数型プログラミング言語 Haskell の型システムを用いて、証明付きのプログラムを記述する方法を紹介します。想定読者層は Haskell のプログラムを読み書きでき、Coq や Agda にもちょっとくらい触ったことがあるかないか程度の人です。「Haskell のプログラム」というのがどの範囲であるのかはちょっと述べづらいですが、GADTs や型族をちょっと触ったことがある、という程度なら全く問題ないでしょう。まだ触ったことがなくても、何となくわかるような解説にしたつもりではいます。

以下、特に断りの無い限りは Haskell Platform 2013.2.0.0 付属の GHC 7.6.3 を仮定して話を進めます。

3.1.1 記号対照表

ソースコードを入力する際には、以下の変換表を参考にしてください。

表 3.1: 本章で用いるコードの表記法

記号	コード	記号	コード	記号	コード	記号	コード
λ	\	$\rightarrow$	->	$\leftarrow$	<-	$\Rightarrow$	=>
$\wedge$	/\	$\vee$	\/	$\approx$	:=:	$\cong$	=~=
$\circ$	.	$\forall$	forall	$\equiv$	===	$\exists$	Exists
$\preceq$	<<=	$\leq$	<=				

ログラミングを再現する **Singleton パターン**について解説していきたいと思います。

3.3.2 Singleton パターン

さて、前節までの機能、特に DataKinds と GADTs を使えば値→型方向の依存型は Haskell でも実現できることを見ました。更に、型→値方向での依存型をエミュレートするするために導入された手法が **Singleton パターン**です。Singleton パターンの基本的な考え方は、「**型に対応する値をただ一つだけもち、構造としても同型となるようなデータ型を用意してやる**」というものです。

……ちょっとよくわかりませんね。そこで、自然数とベクトルの例を通じて詳しく見てみましょう。

まず、以下のプログラムを動かすに当り、「おまじない」として以下のプラグマを先頭に挿入して、必要な言語拡張を有効にしておいてください：

```
{-# LANGUAGE DataKinds, GADTs, PolyKinds, ScopedTypeVariables, TypeFamilies #-}
{-# LANGUAGE TypeOperators, UndecidableInstances                              #-}
```

さて、自然数の定義は次でした：

```
data Nat = Z | S n deriving (Read, Show, Eq, Ord)
```

こうすると、DataKinds 拡張により Nat を型レベルへ昇格したものが得られるのでした。これに対応するシングルトンは次のようになります：

```
data SNat (n :: Nat) where
  SZ :: SNat Z
  SS :: SNat n → SNat (S n)
```

SNat は Nat 種の型を引数に取るデータ型です。例えば、`SZ :: SNat Z`、`SS SZ :: SNat (S Z)`、`SS (SS (SS SZ)) :: SNat (S (S (S Z)))` のようになります。このように、自然数 `n :: Nat` に対し、SNat n 型の**値は唯一つしか存在せず**、更に型に現れる自然数の構造（Z や S (S (S Z)) など）と項の構造（SZ や SS (SS (SS SZ))）は**同型**になっています。こうすることにより、SNat n に対する**パターンマッチを行うことで型の情報を値から復元できる**よ

4 TPPmark 2013解説

坂口 和彦

2013 年 11 月 21〜22 日に信州大学工学部キャンパスで定理証明器ユーザの集まり TPP 2013 [21] が開催された。TPP は 2005 年から毎年行われており、2009 年からは参加者が解いて解を比較するための問題が出題されている。この問題が TPPmark である。本章では、TPPmark 2013 の解法と Coq による解答例を説明する。

4.1 問題

本書向けに分かりやすく書き直した TPPmark 2013 の問題を以下に示す。オリジナルの問題は TPP 2013 のウェブサイトを参照。

> 4×4 の格子状のボードの上に 1 から 15 のコマが配置され、残りの一つのマスが空いているとする。空白部分と辺を共有して接しているマスのコマを空白部分に移動する操作を繰り返し、図 4.1 のように左上から右下に向かって昇順にコマが並び右下隅に空白がある配置（これを初期配置と呼ぶ）に変形するパズルは **15 パズル**という名前で知られている。
>
> この 15 パズルの性質として、正しい操作の繰り返しで初期配置に到達できない配置が存在することが知られている。初期配置の 14, 15 を入れ替えた図 4.2 の配置がそのようなパズルとしての解が存在しない配置であることを証明せよ。

このような invariant の定義に、置換の**偶奇性** (*parity*) を用いる。置換を有限個の異る要素同士の互換の合成で表そうとしたとき、その互換の数は偶数もしくは奇数のどちらか一方に定まることが知られている。これが偶数である置換を**偶置換** (*even permutation*)、奇数である置換を**奇置換** (*odd permutation*) と呼ぶ。置換の偶奇性と合成は、自然数の偶奇性と加算と同じ性質を満たす。

さて、15 パズルの状態遷移について考えると、一回の状態遷移で必ず偶奇性が反転することが分かる。また、空白部分の座標の偶奇も変化するので、これらの xor を不変量とする。図 4.1 と図 4.2 のボードの状態では空白部分の座標は同じであり、それぞれ偶置換と奇置換である。よって、この方針で問題無く証明できると考えられる。

4.4　証明

不変量を以下の通り定義する。

```
Definition invariant (b : board) : bool :=
  let (ex, ey) := b^-1%g empty_space in odd_perm b (+) odd ex (+) odd ey.
```

odd_perm は置換が奇置換であれば true、偶置換であれば false を返す。実際には coercion が定義されているので odd_perm は書かなくても良い。

4.4.1　状態遷移は不変量を保存する

補題 next_invariant を証明する。

```
Lemma next_invariant b1 b2 :
  b2 \in puzzle_next b1 -> invariant b1 = invariant b2.
Proof.
```

まずは puzzle_next と invariant を展開しよう。

```
  rewrite /puzzle_next /invariant.
```