目次

1 Coq で PostScript プログラミング

坂口 和彦

1.1 はじめに

PostScript や Forth などのプログラミング言語では、スタック上の値の並びを変化させる命令とスタックの先頭のいくつかの値に対する操作の連なりによってプログラムを記述する。このようなプログラムの構成方法を**スタック指向プログラミング** (*stack-oriented programming*) と呼ぶ。ここでは簡単な例として、スタック上に $a\ b\ c\ d$ の順で値が並んでいるのを、何かしらの操作によって $d\ c\ a\ c$ に変化させることを考える。ただし、左側がスタックの先頭であるものとする。

この例では、

- スタックの先頭の値を捨てる pop 命令
- スタックの先頭の値を取り出し、その値をスタックの先頭に 2 回積む copy 命令
- スタックの先頭の $n+m$ 個の値の並びを先頭向きに n 個分回転させる、即ちスタックの先頭の $v_1 \dots v_n\ v_{n+1} \dots v_{n+m}$ を $v_{n+1} \dots v_{n+m}\ v_1 \dots v_n$ に変化させる roll$(n+m, n)$ 命令

の 3 つの命令が使えるものとする。

これらの命令を使って目的の並び換えを実現する方針はいくつか考えられるが、ここではスタックの末尾から先頭に向かって徐々に目的の順番に並べていく。この方針の下では最初に c をスタックの底に移動させるべきだが、c は 2 箇所に出現しているので最初にスタックの先頭に移動させて copy 命令によって複製した上でそのうちの一方をスタックの底に移動させるべきである。そこで、まずは roll$(3, 2)$ copy roll$(5, 1)$ を実行する。すると、スタックの状態は $c\ a\ b\ d\ c$ に変化する。次も同様に先頭の 4 要素に注目し、roll$(4, 2)$ を実行す

```
      instexec, instquote, instcons
    & cs]) |=>* ([:: d; c; a; c], [::])
```

このゴールは単に命令列の最後の [:: instexec] の部分に cs という名前を付けただけであり、仮定 Hcs によれば cs と [:: instexec] は同じなので意味は一切変わっていない。しかし、これに対して直前の自動証明の方法をそのまま適用すると、以下の通り証明不能なゴールが得られる [*5]。

```
do !econstructor.
```

```
...
  ============================
   ([:: instpair (instpair (instpush a) (instpush c)) (instpush d); c], cs)
   |=> ([:: d; c; a; c], [::])
```

補題 1.1 に最初の状態を渡して 1 ステップ分の証明を取り出して適用することによって、より確実な自動化が実現できる。これは具体的にはタクティク記述言語 Ltac を用いて以下のように書ける。

```
Ltac evalstep :=
  match goal with |- ?s |=>* _ =>
    apply evalrtc_refl ||
    match eval hnf in (decide_eval s) with
      or_introl _ (ex_intro _ _ ?p) => apply (@evalrtc_cons _ _ _ p)
    end
  end.
```

evalstep タクティクは、ゴールが s |=>* _ の形である場合に使える。このタクティクはまずは evalrtc_refl の適用を試し、次に decide_eval s を計算することで s から 1 ステップ分の遷移の証明を探す。もし証明が計算できれば、それを適用することで証明を 1 ステップ分進める。さらに、以下の evalauto タクティクによって、evalstep タクティクを繰り返し実行できる。

```
Ltac evalauto := do !evalstep.
```

このタクティクを用いて econstructor の単純な繰返しでは解けなかった問題が解けることを確認する。

[*5] 一見これは証明可能なようにも見えるが、ゴール全体が |=>* ではなく |=> になっている点に注目すると、明らかに証明不能である。

2 古森の問題

平井 洋一

前から気になっているが解けずにいる問題 [13] があるので、載せてみる。形が簡単な論理式は意味も簡単かという気分の問題である。筆者は、つい SSReflect を使ってしまう。

```
Require Import ssreflect eqtype ssrnat.
```

論理結合子が含意しかない命題論理式の話をするので、それを定義する。原子論理式は、なんでもよかったのだが、自然数にしてしまう。

```
Definition pvar := nat.
```

命題論理式をつくるには、原子論理式をもってくるか、命題論理式をふたつもってくるかするとよい。後者は含意というつもり。

```
Inductive fmla : Set :=
| atom : pvar -> fmla
| imp  : fmla -> fmla -> fmla
.
```

つい、記法をつくってしまった。

```
Notation "'&' a " := (atom a) (at level 80).
Notation "a --> b" := (imp a b) (at level 81, right associativity).
```

命題論理式の中の原子論理式に命題論理式を代入することができる。代入を使うと、ヒルベルト流の演繹体系を記述しやすいし、命題論理式の形の複雑さを比べることもできる。

```
Fixpoint substitute (a : pvar) (f : fmla) (g : fmla) : fmla :=
  match g with
    | atom b =>
      if a == b then f else atom b
```

3 TPPmark 2014解説

坂口 和彦

3.1 はじめに

2014 年 12 月 3〜5 日に九州大学西新プラザで定理証明器ユーザの集まり TPP2014 [8] が開催された。TPP は 2005 年から毎年行われており、2009 年以降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、筆者による TPPmark 2014 の Coq での解について説明する。他にも多数の解が公開されているので、もし興味があれば TPP2014 のウェブサイトを参照すると良いだろう。また、筆者は以前にも TPPmark 2013 の解説記事 [15] を書いている。

本章では、表 3.1 に挙げる数式上の表記法を用いる。

表 3.1: 3 章で用いる数式上の表記法

表記法	意味
$p \bmod q$	$p = kq + r \wedge r \leq q$ $(r, k \in \mathbb{N})$ を満たす r
$\left\lfloor \frac{p}{q} \right\rfloor$	$p = kq + r \wedge r \leq q$ $(r, k \in \mathbb{N})$ を満たす k
$q \mid p$	p は q で割り切れる

コラム: TPP2014　TPP2014 は、第 10 回の TPP であることを記念して、以前より 1 日日程を拡大して行われた。筆者は TPP2012 から参加し始めたが、以前と比較すると Coq 以外の証明器の話題が増えたように思う。また、参加者数も以前と比較すると倍くらいなり、参加者の興味の対象も広がっているように感じた。