定理証明支援系Coq上での対話的スタック指向プログラミング

坂口和彦[1,a)]

概要：本発表では，スタック指向プログラミング言語 PS0 と定理証明支援系 Coq 上に実装した PS0 の対話的プログラミング環境を紹介する．この対話的プログラミング環境では，ユーザはまず目的のプログラムがスタックの状態をどう変化させるかを表す形でプログラムの仕様を与え，次にプログラムの断片や証明を順番に与えていくことでプログラムを完成させる．このとき，記述するべき残りのプログラムの仕様は今までに入力したプログラムでどのような状態のスタックに遷移するかを表しているため，一般化されたスタックの状態を見ながらプログラムを記述できる．また，PS0 のプログラムを PostScript のプログラムに変換し，Ghostscript 等の PostScript インタプリタの上で実行できることを確認した．

キーワード：Coq，スタック指向プログラミング

Interactive Stack-oriented Programming in the Coq Proof Assistant

Sakaguchi Kazuhiko[1,a)]

Abstract: We introduce a stack-oriented programming language PS0 and its interactive programming environment implemented in the Coq proof assistant. In this programming environment, first, a user writes the specification of a program as to how the program mutates the stack. Next, the user interactively fills unspecified part of the program. Finally, the user obtains the complete program that proved satisfying the specification. At each step, the environment shows the specification of the program that we should fill. The specification describes the property of the initial and final state of the stacks. Therefore, we can write the program with the generalized state of the stack given. We also implemented a translator from PS0 programs to PostScript programs. It was confirmed that PostScript programs generated by the translator can be executed on PostScript interpreters (e.g. Ghostscript) properly.

Keywords: Coq, stack-oriented programming

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計算の命令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
- スタックの先頭の値の並び $v_1 \ldots v_n\, v_{n+1} \ldots v_{n+m}$ を $v_{n+1} \ldots v_{n+m}\, v_1 \ldots v_n$ に変化させる $\mathrm{roll}(n+m, n)$ 命令

の 3 つの命令が使えるものとする．

[1] 筑波大学 情報学群 情報科学類
College of Information Science, School of Informatics, University of Tsukuba
[a)] sakaguchi@score.cs.tsukuba.ac.jp

図 1　PS0 言語の構文とスモールステップ操作的意味論
Fig. 1　The syntax and small-step operational semantics of the PS0 language

$$
\begin{array}{lll}
i_1, i_2 ::= \mathrm{pop} & & (v_1\overline{v}, \mathrm{pop}\ \overline{i}) \mapsto (\overline{v}, \overline{i}) \\
\quad | \ \mathrm{copy} & & (v_1\overline{v}, \mathrm{copy}\ \overline{i}) \mapsto (v_1 v_1 \overline{v}, \overline{i}) \\
\quad | \ \mathrm{swap} & & (v_1 v_2 \overline{v}, \mathrm{swap}\ \overline{i}) \mapsto (v_2 v_1 \overline{v}, \overline{i}) \\
\quad | \ \mathrm{cons} & & (v_1 v_2 \overline{v}, \mathrm{cons}\ \overline{i}) \mapsto (\mathrm{pair}(v_2, v_1)\,\overline{v}, \overline{i}) \\
\quad | \ \mathrm{quote} & & (v_1\overline{v}, \mathrm{quote}\ \overline{i}) \mapsto (\mathrm{push}(v_1)\,\overline{v}, \overline{i}) \\
\quad | \ \mathrm{exec} & & (v_1\overline{v}, \mathrm{exec}\ \overline{i}) \mapsto (\overline{v}, v_1\overline{i}) \\
\quad | \ \mathrm{push}(i_1) & & (\overline{v}, \mathrm{push}(v_1)\ \overline{i}) \mapsto (v_1\overline{v}, \overline{i}) \\
\quad | \ \mathrm{pair}(i_1, i_2) & & (\overline{v}, \mathrm{pair}(i_1, i_2)\ \overline{i}) \mapsto (\overline{v}, i_1 i_2 \overline{i})
\end{array}
$$

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 $cabdc$ に変化する．次も同様に先頭の 4 要素に注目し，roll$(4,2)$ を実行すると，$bdcac$ に変化する．あとは先頭の b を取り除くだけなので，pop を実行するとスタックの状態は目的の $dcac$ に変化する．

この例から分かるように，スタック上の値の並びの操作だけによって目的の仕様を満たすプログラムを構成するのは非常に難しい．更に，現実のプログラムの仕様はより抽象的なスタックの状態を使って書き表されるので，それぞれの時点での状態をメモしながらプログラムを書き進めたとしてもバグを埋め込みやすく，もしバグを埋め込まずにプログラムを書き終えられるとしても非常に手間がかかると考えられる．一方で，プログラムの今編集しようとしている部分でスタックの状態が一般的にどういう形をしているかを表示できれば，スタック指向プログラミングはより簡単になるはずである．本研究では，スタック指向プログラミング言語 PS0 を定義し，定理証明器 Coq[2] とその拡張 SSReflect[1] を用いてスタックの状態を表示しながらプログラムが記述できる PS0 のプログラミング環境 [3] を開発した．

2. PS0 言語の定義

PS0 言語の**命令** (*instruction*) とスモールステップ操作的意味論を図 1 の通り定義する．PS0 ではプログラムだけではなくデータもこの命令を用いて表現する．PS0 プログラムの実行中の状態は命令列 2 つの組であり，命令列は命令の 0 個以上の並びである．ここで定義されている操作的意味論 $\mapsto$ は状態上の二項関係である．$\mapsto$ の反射推移閉包を $\mapsto^*$ で表記する．

Coq での PS0 言語の定義を以下に示す．まず，命令は帰納的データ型として記述できる．

```
Inductive inst : Set :=
  | instpop  | instcopy  | instswap
  | instcons | instquote | instexec
  | instpush of inst
  | instpair of inst & inst.
```

状態は 2 つの命令列の組である．

```
Notation state := (seq inst * seq inst)%type.
```

この定義での`%type` は表記法を解釈するための**スコープ** (*scope*) の指定である．

```
Locate "*".
```

```
Notation            Scope
"x * y" := prod x y  : type_scope

"m * n" := muln m n  : nat_scope
                       (default interpretation)
...
```

直積の意味で`_ * _`を使いたければ，`type_scope` にする必要があるので，`%type` を付ける．一方，自然数の乗算 (`muln`) の意味で使うためには `nat_scope` にする必要があるが，`default interpretation` と書いてあるので，他のスコープになっていない限りは明示的に`%nat` を付ける必要は無い．

PS0 の操作的意味論は，状態上の二項関係として定義できる．

```
Inductive eval : relation state :=
  | evalpop   : forall i vs cs,
    eval (i :: vs, instpop :: cs) (vs, cs)
  | evalcopy  : forall i vs cs,
    eval (i :: vs, instcopy :: cs)
         (i :: i :: vs, cs)
  | evalswap  : forall i1 i2 vs cs,
    eval (i2 :: i1 :: vs, instswap :: cs)
         (i1 :: i2 :: vs, cs)
  | evalcons  : forall i1 i2 vs cs,
    eval (i2 :: i1 :: vs, instcons :: cs)
         (instpair i1 i2 :: vs, cs)
  | evalquote : forall i vs cs,
    eval (i :: vs, instquote :: cs)
         (instpush i :: vs, cs)
  | evalexec  : forall i vs cs,
    eval (i :: vs, instexec :: cs) (vs, i :: cs)
  | evalpush  : forall i vs cs,
    eval (vs, instpush i :: cs) (i :: vs, cs)
  | evalpair  : forall i1 i2 vs cs,
    eval (vs, instpair i1 i2 :: cs)
         (vs, i1 :: i2 :: cs).
```

eval とその反射推移閉包をそれぞれ|=>と|=>*で書けるようにする．

```
Notation evalrtc :=
  (clos_refl_trans_1n state eval).
Infix "|=>" :=
  eval (at level 50, no associativity).
Infix "|=>*" :=
  evalrtc (at level 50, no associativity).
```

状態を構成する 2 つの命令列の直感的な意味は，スタックとこれから実行するプログラムである．図 1 に示した操作的意味論の規則はそれぞれ 8 つの命令に対応する実行の規則であり，左辺のプログラムの先頭がその命令になっている．また，どのような状態に対しても $\mapsto$ で到達できる状態の有無は容易に判定できる．よって，明らかに以下の 2 つの補題が成り立つ．

補題 1 任意の状態 s_1 について，$s_1 \mapsto s_2$ を満たす状態 s_2 が存在するか，存在しないかのどちらかが成り立つ [*1]．

```
Lemma decide_eval s1 :
  decidable (exists s2 : state, s1 |=> s2).
```

補題 2 任意の状態 s について，$s \mapsto s'$ を満たす状態 s' は唯一に定まる．

```
Lemma eval_uniqueness s1 s2 s3 :
  s1 |=> s2 -> s1 |=> s3 -> s2 = s3.
```

補題 1 は次節の内容で計算を目的として使うので，この証明は Qed ではなく Defined で終わらせる必要がある [*2]．また，上に示した以外にも以下の補助的な補題を証明した．

```
Lemma evalrtc_refl s : s |=>* s.
Lemma evalrtc_refl' s1 s2 : s1 = s2 -> s1 |=>* s2.
Lemma evalrtc_step s1 s2 : s1 |=> s2 -> s1 |=>* s2.
Lemma evalrtc_cons s1 s2 s3 :
  s1 |=> s2 -> s2 |=>* s3 -> s1 |=>* s3.
Lemma evalrtc_trans s1 s2 s3 :
  s1 |=>* s2 -> s2 |=>* s3 -> s1 |=>* s3.
```

3. 簡単な PS0 プログラムの例と実行による証明

本節では，第 1 節で考えたのと同じプログラムを PS0 で構成することを考える．即ち，仕様 $\forall a\,b\,c\,d.(a\,b\,c\,d, \bar{i}) \mapsto^* (d\,c\,a\,c, \epsilon)$ [*3] を満たす命令列 $\bar{i}$ を考えれば良い．後半では，得られたプログラムが仕様を満たしていることを Coq 上で検証するが，その際に証明を自動化するためのタクティクを提供する．

第 1 節で示したのとは順番を少し変更し，先に b を捨てる．$\bar{i}$ の先頭の 2 命令を swap pop とすると，それらの命令を実行した後のスタックは $a\,c\,d$ となる．これ以降も，同様に $\bar{i}$ の残りの部分の先頭の命令を考え，それらを実行し終えた時点でのスタックの状態がどうなるかを考えることでプログラムを構成する．次は c を複製してそのうち一方をスタックの末尾に移動したいが，PS0 の命令はどれも単体ではスタックの 3 番目以降の値を直接扱うことができないので，push や pair などの構造を使って複数の値を 1 つの値にまとめる必要がある．例えば，スタックの先頭に積まれた $a\,b$ に対して quote swap quote cons を実行することで $\mathrm{pair}(\mathrm{push}(b), \mathrm{push}(a))$ が得られるが，これを後で実行することによって $a\,b$ を復元できる．同様に $\mathrm{pair}(\mathrm{push}(c), \mathrm{pair}(\mathrm{push}(b), \mathrm{push}(a)))$，$\mathrm{pair}(\mathrm{pair}(\mathrm{push}(d), \mathrm{push}(c)), \mathrm{pair}(\mathrm{push}(b), \mathrm{push}(a)))$ のように入れ子にすることで，いくらでも多くの値を 1 つの値にまとめて扱うことが可能である．この仕組みを利用して，swap quote quote copy cons swap quote quote cons exec を実行することで $\mathrm{push}(a)\,\mathrm{push}(c)\,\mathrm{push}(c)\,d$ のように順序を入れ替えられる．まだ c をスタックの末尾に移動し終えていないが，先に swap cons を実行して目的の状態に含まれる $c\,a$ に展開される命令を作ることにする．この時点でのスタックは $\mathrm{pair}(\mathrm{push}(c), \mathrm{push}(a))\,\mathrm{push}(c)\,d$ である．次に quote cons swap quote cons exec を実行するとスタックは $d\,\mathrm{pair}(\mathrm{push}(a), \mathrm{push}(c))\,c$ となり，漸く c をスタックの末尾まで移動させることができた．あとは quote cons exec を実行すれば，スタックは目的の $d\,c\,a\,c$ になる．

目的のプログラムが書けたので，次はこれが実際に仕様通りになっていることを Coq 上で検証する．

```
Lemma example1 a b c d :
  ([:: a; b; c; d], [::
   instswap; instpop; instswap; instquote;
   instquote; instcopy; instcons; instswap;
   instquote; instquote; instcons; instexec;
   instswap; instcons; instquote; instcons;
   instswap; instquote; instcons; instexec;
   instquote; instcons; instexec]) |=>*
  ([:: d; c; a; c], [::]).
```

[:: ...] は SSReflect の seq モジュールで提供されているリストの表記法である．証明の最初の段階として，反射推移閉包に対する 1 ステップの分解をする．$s_1 \mapsto^* s_2$ を適当な s_1' に関して $s_1 \mapsto s_1'$ と $s_1' \mapsto^* s_2$ に分解して証明しようとすると，通常は s_1' を明示的に与えて証明をする必要がある．しかし，証明を進めるごとにこの状態を明示的に与えるのはあまりにも労力が大き過ぎる．そこで Coq の eapply タクティクを使うことで，この s_1' の中身が何で

[*1] Coq では公理を追加しない限り排中律は成り立たないので，このような性質は別の方法を用いて証明する必要がある．
[*2] Qed で証明を終わらせた場合にはその証明項が計算としては使えない状態になるが，Defined とすると計算にも使えるようになる．
[*3] ここでの ϵ は空列を意味する．

あるかを明らかにするのを先送りにする．

```
Proof.
  eapply evalrtc_cons.
```

```
2 subgoals, subgoal 1 (ID 511)

  a : inst
  b : inst
  c : inst
  d : inst
  ============================
   ([:: a; b; c; d],
   [:: instswap; instpop; instswap; instquote;
       instquote; instcopy; instcons; instswap;
       instquote; instquote; instcons; instexec;
       instswap; instcons; instquote; instcons;
       instswap; instquote; instcons; instexec;
       instquote; instcons; instexec]) |=> ?510

subgoal 2 (ID 512) is:
 ?510 |=>* ([:: d; c; a; c], [::])
```

先送りにした結果として，ゴールの一部分に?510 という項が出現した [*4]．これは Coq の**存在変数** (*existential variable*) である．存在変数 $?x$ を含む命題 $P[?x]$ は，$?x$ を適当な中身で埋めることによって $P[?x]$ を真にできることを意味する．即ち，おおよそ $\exists x.P[x]$ と同じ意味である．ただし，$\forall x.\exists y.\forall z.P$ の証明をする上で y の具体的な構成に x は使って良いが z は使ってはいけないのと同様に，存在変数の中身を埋めるために使って良い変数，仮定は限られている．Show Existentials コマンドによって，存在変数の中身を埋めるのにどの変数が使えるかを表示できる．

```
  Show Existentials.
```

```
...
Existential 3 =
?510 : [a : inst  b : inst  c : inst  d : inst
       |- state]
```

この結果によれば，存在変数?510 の中身を埋めるために使える変数は a, b, c, d の 4 つ，即ちここで見えている全ての変数である．

1 番目のサブゴールは明らかに swap の実行規則 evalswap の形をしているので，これを apply する．実は，この操作をすることで自動的かつ適切に存在変数?510 を埋められる．

```
  by apply evalswap.
```

```
1 subgoals, subgoal 1 (ID 512)

  a : inst
  b : inst
  c : inst
  d : inst
  ============================
   ([:: b; a; c; d],
   [:: instpop; instswap; instquote; instquote;
       instcopy; instcons; instswap; instquote;
       instquote; instcons; instexec; instswap;
       instcons; instquote; instcons; instswap;
       instquote; instcons; instexec; instquote;
       instcons; instexec])
   |=>* ([:: d; c; a; c], [::])
```

SSReflect での by tac は tac; done と同じ意味である．done はある種の自動化のタクティクであり，もし done が与えられたゴールを証明できなかった場合にはエラーになる．よって，by tac と書くことでそこで 1 つのサブゴールを閉じられることを明示でき，証明のメンテナンス性向上に役立つ．

さて，今示した手順で残りの証明を進めていくと，残りの命令数の 2 倍の 44 個のタクティクを書き並べることになる．しかし，今見た証明の 2 つのステップはどちらもコンストラクタを適用しているだけなので，単に econstructor タクティクを繰り返し適用することで全ての証明を終えられる．

```
  do !econstructor.
```

```
No more subgoals.
```

しかし，この方法での自動証明はゴールの形を少し変えるだけで通用しなくなる．以下にその例を示す．

```
  Undo. (* do !econstructor. *)
  set cs := [:: instexec];
    move: {1 3}cs (eq_refl cs); subst cs => cs Hcs.
```

```
...
  cs : seq inst
  Hcs : cs = [:: instexec]
  ============================
   ([:: b; a; c; d],
   [:: instpop, instswap, instquote, instquote,
       instcopy, instcons, instswap, instquote,
       instquote, instcons, instexec, instswap,
       instcons, instquote, instcons, instswap,
       instquote, instcons, 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

[*4] この 510 という番号は，異なる番号になる可能性がある．

れる [*5].

```
  do !econstructor.
```

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

補題 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．

```
  Undo. (* do !econstructor. *)
  evalauto.
```

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

直前と非常に良く似た形のゴールだが，|=>だった部分が|=>*になっている．cs を書き換えた上で，もう一度 evalauto を実行すると証明を終えられる．

```
  rewrite Hcs; evalauto.
Qed.
```

補題 2 によれば PS0 プログラムの実行は分岐し得ないので，evalauto タクティクによって証明可能なゴールが証明不能なゴールに書き換わることは無い．

[*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．

4. 対話的プログラミング

第 3 節で示した方法でプログラムと証明を記述すると，一回で正しいプログラムを書くのが困難であるという前提の下ではプログラムの修正と証明を交互に行うことになる．本節では，証明を進めるごとに対応する部分のプログラムを埋める形でプログラムと証明を記述する方法と，そのスタイルで証明を記述するためのタクティクを提案する．

前節までと同じ例を考えるが，今回は「あるプログラムが存在して仕様を満たす」という形で命題を記述する．

```
Lemma example2 :
  { p | forall a b c d,
    ([:: a; b; c; d], p) |=>*
    ([:: d; c; a; c], [::]) }.
```

この命題は，$\exists \overline{p}.\forall a\, b\, c\, d.(a\, b\, c\, d, \overline{p}) \mapsto^* (d\, c\, a\, c, \epsilon)$ の意味である．証明の最初のステップでは，$\overline{p}$ を存在変数に変える．

```
Proof.
  eexists => a b c d.
```

```
1 subgoals, subgoal 1 (ID 1424)

  a : inst
  b : inst
  c : inst
  d : inst
  ============================
   ([:: a; b; c; d], ?1419)
   |=>* ([:: d; c; a; c], [::])
```

このプログラムの最初のステップは，前節では swap となっていた．それに対応する証明を，ここでは以下のように記述する．

```
  eapply evalrtc_cons; first by apply evalswap.
```

```
...
  ============================
   ([:: b; a; c; d], ?1432)
   |=>* ([:: d; c; a; c], [::])
```

tac1; first tac2 と書くと tac1 を適用して最初のサブゴールに対して tac2 を適用するので，first by tac は「最初のゴールを tac で閉じる」と読める．

直前のステップとは左辺のスタックの先頭 2 要素が入れ替わっており，プログラムに相当する部分の存在変数の番号も変わっている．これで証明が 1 命令分進んだと言える．この手順をより一般化したタクティクを以下のように記述できる．

```
Tactic Notation
  "evalpartial" constr(H) "by" tactic(tac) :=
    eapply evalrtc_trans;
```

```
    [ by (apply evalrtc_step; eapply H) ||
      eapply H; tac | ];
    subst_evars.

Tactic Notation "evalpartial" constr(H) :=
  evalpartial H by idtac.
```

evalpartial H by tac は，H で証明を進めて最後のサブゴール以外に tac; done を適用するタクティクである．ここでの H は s |=> s’ の形でも s |=>* s’ の形でも良い．また，evalpartial H は evalpartial H by idtac の意味である．evalpartial タクティクの定義には subst_evars というタクティクを使っているが，これは存在変数の具体化 (instantiation) を強制するためのタクティクである．このタクティクの定義と使い方は，第 5 節で説明する．

evalpartial タクティクを使うことで，残りの証明は以下のように書ける．

```
  evalpartial evalpop. evalpartial evalswap.
  evalpartial evalquote. evalpartial evalquote.
(* somehow *)
  evalpartial evalquote. evalpartial evalcons.
  evalpartial evalexec. evalauto.
Defined.
```

ここまでの例では，プログラム p やそれに対応する存在変数は命令列であった．evalpartial evalpair によってプログラムの先頭の命令を 2 つに分割できるのを利用すると，この p がプログラムの先頭にある命令である場合に対する evalpartial と同じようなタクティクが書ける．

```
Tactic Notation
  "evalpartial’" constr(H) "by" tactic(tac) :=
    evalpartial evalpair; evalpartial H by tac.

Tactic Notation "evalpartial’" constr(H) :=
  evalpartial’ H by idtac.
```

最後に，「プログラムが存在することの証明」から具体的なプログラムが取り出せることを確認する．証明を Defined で終えているのはこのためであり，これを Qed とすると証明からプログラムを取り出せなくなる．

```
Eval compute in (proj1_sig example2).
```

```
     = [:: instswap; instpop; instswap; instquote;
           instquote; instcopy; instcons;
           instswap; instquote; instquote;
           instcons; instexec; instswap; instcons;
           instquote; instcons; instswap;
           instquote; instcons; instexec;
           instquote; instcons; instexec]
     : seq inst
```

本節で見たプログラミング手法を用いて，何もしない命令 nop と，cons の引数の順序が入れ替わった命令 snoc を定義する．これらは第 6 節以降で利用する．

```
Lemma exists_nop :
  { instnop | forall vs cs,
    (vs, instnop :: cs) |=>* (vs, cs) }.

Notation instnop := (proj1_sig exists_nop).
Notation evalnop := (proj2_sig exists_nop).
```

```
Lemma exists_snoc :
  { instsnoc | forall i1 i2 vs cs,
    (i1 :: i2 :: vs, instsnoc :: cs) |=>*
    (instpair i1 i2 :: vs, cs) }.

Notation instsnoc := (proj1_sig exists_snoc).
Notation evalsnoc := (proj2_sig exists_snoc).
```

この定義では，得られたプログラムと証明それぞれに対して Notation を使って名前を付けている．これ以降の内容でも，何かの仕様を満たすプログラムが存在することを証明するごとに上と同じようにしてそのプログラムと証明それぞれに名前を付ける．これ以降の内容ではこの 2 行は省略する．

5. 存在変数の具体化

第 4 節で使われている subst_evars タクティクは，存在変数の具体化を強制するのに有用ということであった．本節では，このタクティクの使い方と定義方法を説明する．

以下に subst_evars タクティクが必要となる例を示す．

```
Goal forall a, a = size [::tt; tt] -> a = 2.
Proof.
  move=> a H.
  (eapply eq_trans; first apply H); simpl.
```

```
1 subgoals, subgoal 1 (ID 1714)

  a : nat
  H : a = size [:: tt; tt]
  ============================
   size [:: tt; tt] = 2
```

この例では，一番最後のタクティクで simpl を実行しているはずだが，なぜかその後のゴールは size [:: tt; tt] という部分を含んでいる．これは意図した通りであれば simpl によって計算されて 2 になるはずの部分であり，そのまま残っているのはバグのような不思議な振舞いに見える．もう少し証明の詳細を見ると，eapply eq_trans によってゴール a = 2 が a = ?n と?n = 2 に分けられており，その直後の first apply H によって分けられたうちの前者は apply H によって a = size [:: tt; tt] と単一化されている．直感的にはこの時点で?n に size [:: tt; tt] が代入されるべきだが，実際には後者での

この代入は一連のタクティクの並びを全て実行し終えた後，即ち simpl の直後に行われており，このために size [:: tt; tt] は simpl によって 2 に置き換えられないまま残っている．subst_evars タクティクはこのような場面において存在変数への代入の強制に使える．

```
  Undo.
  (eapply eq_trans; first apply H);
    subst_evars; simpl.
```

```
...
  ============================
   2 = 2
```

```
  done.
Qed.
```

subst_evars タクティクは以下のように定義できる．

```
Ltac subst_evars :=
  match goal with |- _ => idtac end.
```

match のこの振舞いに関しては探した限りではリファレンスマニュアル [2] には書かれていないが，存在変数が具体化されていないために match に失敗するのは不便であるため，このような実装になっていると考えられる．実際，我々は上述の問題に対して match goal with ... でゴールを出力させようとしたときにこの解決方法を発見した．

6. データ型の定義

PS0 におけるブール型や自然数型などのデータ型の定義方法を考える．具体的な命令の形を以って定義する方法とチャーチ数のように外延的に定義する方法が考えられるが，ここでは後者に近い考え方を採用する．

自然数型の定義を始める前に，準備として以下の定義を導入する．

```
Notation instseqc' := (foldl instpair).
Notation instseqc := (instseqc' instnop).
Notation instnseqc' n i1 i2 :=
  (iter n (flip instpair i1) i2).
Notation instnseqc n i := (instnseqc' n i instnop).
```

instseqc' i il は，実行することでプログラム列の先頭に i :: il の形で展開される命令である．instseqc il は instseqc' instnop il であり，instnop は何もしない命令なのでこの形ではプログラム列の先頭には単に il が残る．instnseqc' n i1 i2 は実行することでプログラム列の先頭に i2 :: nseq n i1，即ち i2 の後に i1 が n 個続く形に展開される命令である．instnseqc n i は instnseqc' n i instnop である．これらの定義に関して，以下が成り立つ．

```
Lemma evalseqc' il i vs cs :
  (vs, instseqc' i il :: cs) |=>*
  (vs, i :: il ++ cs).
Lemma evalseqc il vs cs :
  (vs, instseqc il :: cs) |=>* (vs, il ++ cs).
Lemma app_instseqc' il1 il2 i :
  instseqc' i (il1 ++ il2) =
  instseqc' (instseqc' i il1) il2.
Lemma app_instseqc il1 il2 :
  instseqc (il1 ++ il2) =
  instseqc' (instseqc il1) il2.
Lemma instnseqc_eq n i1 i2 :
  instseqc' i2 (nseq n i1) = instnseqc' n i1 i2.
Lemma evalnseqc' n i1 i2 vs cs :
  (vs, instnseqc' n i1 i2 :: cs) |=>*
  (vs, i2 :: nseq n i1 ++ cs).
Lemma evalnseqc n i vs cs :
  (vs, instnseqc n i :: cs) |=>*
  (vs, nseq n i ++ cs).
```

スタックの先頭に積まれた命令 i を instnseqc n i に変化させる命令を，自然数 n を表す命令とする．

```
Definition instnat_spec
  (n : nat) (i1 : inst) : Prop :=
  forall i2 vs cs,
    (i2 :: vs, i1 :: cs) |=>*
    (instnseqc n i2 :: vs, cs).
```

instnat_spec n i が成り立つことを，論理式の上では $\lceil n \rceil(i)$ と表記する．この定義の上での自然数を表す命令は，それが表す自然数の分の繰り返しの計算として利用できる．

```
Lemma exists_inst_repeatn :
  { inst_repeatn : inst |
    forall n i1 i2 vs cs, instnat_spec n i2 ->
    (i1 :: vs, i2 :: inst_repeatn :: cs) |=>*
    (vs, nseq n i1 ++ cs) }.
```

この定義が自然数型の定義として無意味なものではないことを示したい．まず，どんな自然数に対してもその自然数に対応する PS0 の命令が存在することを証明する．

```
Lemma exists_instnat :
  forall n, { i : inst | instnat_spec n i }.
```

この命題は auto で使えると便利なので，ヒントデータベースに登録する．

```
Hint Resolve ((fun n => eval_instnat n) :
  forall n, instnat_spec n (instnat n)).
```

次に，自然数型の定義を満たすどんな命令も同時に異なる 2 つの数を表し得ないことを証明する．これを満たしていない極端なケースとしては全ての自然数の表現になっているような命令が存在する場合が考えられる．もしそうだとすると，結果が自然数であるようなどんな計算に対してもその命令を返せば良いことになってしまい，自然数型の定義としては妥当ではない．

図 2　テンプレートの定義
Fig. 2　The definition of templates

$$
\begin{aligned}
t_1, t_2 ::= \ & pop \\
| \ & copy \\
| \ & swap \\
| \ & cons \\
| \ & quote \\
| \ & exec \\
| \ & push(t_1) \\
| \ & pair(t_1, t_2) \\
| \ & hole(n) \qquad\qquad (n \in \mathbb{N})
\end{aligned}
$$

```
Lemma instnat_eqmap : forall n m i,
  instnat_spec n i -> instnat_spec m i -> n = m.
```

同様にして，ブール型は以下のように定義できる．

```
Definition
  instbool_spec (b : bool) (i1 : inst) : Prop :=
  forall i2 i3 vs cs,
    (i3 :: i2 :: vs, i1 :: cs) |=>*
    ((if b then [:: i2; i3]
           else [:: i3; i2]) ++ vs, cs).

Lemma exists_bool b :
  {instbool : inst | instbool_spec b instbool }.

Hint Resolve ((fun b => evalbool b) :
  forall b, instbool_spec b (instbool b)).

Lemma instbool_eqmap : forall b1 b2 i,
  instbool_spec b1 i -> instbool_spec b2 i ->
  b1 = b2.
```

7. テンプレート

ここまでに出てきた PS0 プログラムの例の多くは，スタックの先頭に積まれた決められた数の値に対して複製，削除，並び換え，定数値の挿入，push や pair などの構造の追加を行うだけのものであった．一般的にこのようなプログラムは，スタックの値をいくつ消費し，消費した値をどのような順序，構造に並び換えるかという情報から自動生成できると考えられる．この範囲のプログラムの仕様を記述するために，図 2 のテンプレートを導入する．

Coq でのテンプレートの定義は以下のようになる．

```
Inductive instt : Set :=
  | insttpop | insttcopy | insttswap
  | insttcons | insttquote | insttexec
  | insttpush of instt
  | insttpair of instt & instt
  | instthole of nat.
```

テンプレートの定義は命令の定義に $hole(n)$ を加えただけの帰納的定義になっている．テンプレート t の直感的な意味は，スタックの先頭にある長さ n の命令列 $\bar{i}$ を消費して t のそれぞれの $hole(m)$ を i_m で置き換えて得られる命令 [*6] を作る計算である [*7]．また，テンプレートから目的の計算が自動的に得られると仮定すると，$t_1 \dots t_n\, t_{n+1} \dots t_{n+m}$ の $n+m$ 個のテンプレートから，スタックの先頭に $t_1[\bar{i}] \dots t_n[\bar{i}]$ を積み，プログラムの先頭には $t_{n+1}[\bar{i}] \dots t_{n+m}[\bar{i}]$ を追加するような命令も作れるはずである．

実際に，Coq 上でいくつかの決定手続きとタクティクを記述することでこれを実現できる．以下に使い方の例を示す．

```
Lemma example3 :
  { p |
    forall i1 i2 i3 vs cs,
    (i1 :: i2 :: i3 :: vs, p :: cs) |=>*
    (instpair i1 instsnoc ::
     instpair i3 (instpush i1) :: vs, cs) }.
Proof.
  eexists => i1 i2 i3 vs cs.
```

```
1 subgoals, subgoal 1 (ID 1752)

  i1 : inst
  i2 : inst
  i3 : inst
  vs : seq inst
  cs : seq inst
  ============================
   ([:: i1, i2, i3 & vs], ?1746 :: cs)
   |=>* ([:: instpair i1 instsnoc,
             instpair i3 (instpush i1)
           & vs], cs)
```

今まで，この種類の証明はそれなりの工夫が必要だったが，evaltemplate タクティクを使うことでテンプレートによる証明ができる．

```
  evaltemplate 3
    [:: insttpair (instthole 0)
          (instt_of_inst instsnoc);
        insttpair (instthole 2)
          (insttpush (instthole 0))]
    (Nil instt).
```

```
...
  ============================
   ([:: instpair i1 (instpair instswap instcons),
        instpair i3 (instpush i1)
      & vs], cs)
   |=>* ([:: instpair i1
               (instpair instswap instcons),
```

*6　これを $t[\bar{i}]$ と表記する．
*7　ただし全ての $hole(m)$ は $m < n$ でなければならない．

```
            instpair i3 (instpush i1)
          & vs], cs)
```

```
  evalauto.
Defined.
```

evaltemplate の 1 番目の引数は消費する命令列，2 番目の引数はスタック側に展開したい命令列を表すテンプレートの列，3 番目の引数はプログラム側に展開したい命令列を表すテンプレートの列である．instt_of_inst は，命令をテンプレートに変換する関数である．これによって，単に作りたい値と同じ形のテンプレートを書くことで目的のプログラムと証明が得られるようになった．ただし，テンプレートによって作られる命令列はサイズが大きくなるので，注意が必要である [*8]．

```
Eval compute in (inst_size (proj1_sig example3)).
```

```
     = 172
     : nat
```

8. 繰返しを含むプログラムの記述

本節では，自然数の加算を例に取り，PS0 での繰返しを含むプログラムの記述方法について説明する．まず，準備として自然数に対する後者関数を定義する．

```
Lemma exists_inst_succn :
  { inst_succn : inst |
    forall n i1 vs cs, instnat_spec n i1 ->
    exists i2 : inst, instnat_spec n.+1 i2 /\
    (i1 :: vs, inst_succn :: cs) |=>*
    (i2 :: vs, cs) }.
```

この命題は今までに見てきたものと比べると少し複雑であるが，一部分に注目すると $\lceil n \rceil(i_1) \Rightarrow \exists i_2.\lceil n+1 \rceil(i_2) \wedge (i_1\,\overline{v}, \mathrm{succ}\,\overline{c}) \mapsto^* (i_2\,\overline{v}, \overline{c})$ となっているので，後者関数の仕様として妥当であることが分かる．これの証明は，i2 を存在変数に変えて内側の/\を分解する必要があることに注意すれば，それ以降は今まで通りの方法で簡単にできる．

上で定義した後者関数を使って，加算を定義する．

```
Lemma exists_inst_addn :
  { inst_addn : inst |
    forall n m i1 i2 vs cs,
    instnat_spec n i1 -> instnat_spec m i2 ->
    exists i3 : inst, instnat_spec (n + m) i3 /\
    (i2 :: i1 :: vs, inst_addn :: cs) |=>*
    (i3 :: vs, cs) }.
Proof.
  eexists => n m i1 i2 vs cs H H0.
```

```
1 subgoals, subgoal 1 (ID 1531)
```

```
  n : nat
  m : nat
  i1 : inst
  i2 : inst
  vs : seq inst
  cs : seq inst
  H : instnat_spec n i1
  H0 : instnat_spec m i2
  ============================
   exists i3 : inst,
     instnat_spec (n + m) i3 /\
     ([:: i2, i1 & vs], ?1522 :: cs)
     |=>* (i3 :: vs, cs)
```

今回はこの論理式の形を保ったまま証明を進めることにする．実はここまでに出てきた evalauto, evalpartial, evaltemplate などのタクティクは，$\exists i.P(i) \wedge s_1 \mapsto^* s_2$ や $\exists i_1.P_1(i_1).\wedge(\exists i_2.P_2(i_2) \wedge s_1 \mapsto^* s_2)$ などのゴールに対しても，s_1 が i_1 や i_2 を含まない限りは対応するように拡張できる．逆に s_1 が i_1 や i_2 を含む論理式をプログラムの仕様として考えることは無いので，この制限が問題になることは無いと考えられる．

i2 に対して後者関数を n 回適用することによって目的の計算をする．まずは第 6 章で定義した自然数による繰り返しの命令が使える形にスタックを変形する．

```
  evalpartial' evalswap.
  evalpartial' evalpush.
  evalpartial' evalswap.
  evalpartial' evalexec.
```

```
...
  ============================
   exists i : inst,
     instnat_spec (n + m) i /\
     ([:: ?1615, i2 & vs], [:: i1, ?1678 & cs])
     |=>* (i :: vs, cs)
```

この形の状態で eval_inst_repeatn を適用すると，?1615 を n 回実行することになる．この後の証明を埋めていく過程で，?1615 は inst_succn になる予定である．

```
  evalpartial (eval_inst_repeatn n).
```

```
...
  ============================
   exists i : inst,
     instnat_spec (n + m) i /\
     (i2 :: vs, nseq n ?1615 ++ cs)
     |=>* (i :: vs, cs)
```

残りは n に関する帰納法で証明できる．i1 やそれに関する仮定はもう使わないので消してしまっても良い．

```
  elim: n m i2 H0 {i1 H} => [ | n IH] m i1 H /=.
```

[*8] inst_size は，命令のコンストラクタ数を返す関数である．

```
2 subgoals, subgoal 1 (ID 1729)

  vs : seq inst
  cs : seq inst
  m : nat
  i1 : inst
  H : instnat_spec m i1
  ============================
   exists i : inst,
     instnat_spec (0 + m) i /\
     (i1 :: vs, cs) |=>* (i :: vs, cs)

subgoal 2 (ID 1733) is:
 exists i : inst,
   instnat_spec (n.+1 + m) i /\
   (i1 :: vs, ?1615 :: nseq n ?1615 ++ cs)
   |=>* (i :: vs, cs)
```

最初のゴールはほぼ自明である．

```
  - by evalauto.
```

```
1 subgoals, subgoal 1 (ID 1733)

  vs : seq inst
  cs : seq inst
  n : nat
  IH : forall (m : nat) (i2 : inst),
       instnat_spec m i2 ->
       exists i : inst,
         instnat_spec (n + m) i /\
         (i2 :: vs, nseq n ?1615 ++ cs)
         |=>* (i :: vs, cs)
  m : nat
  i1 : inst
  H : instnat_spec m i1
  ============================
   exists i : inst,
     instnat_spec (n.+1 + m) i /\
     (i1 :: vs, ?1615 :: nseq n ?1615 ++ cs)
     |=>* (i :: vs, cs)
```

i1 に対して 1 を足したいはずなので，その準備をする．inst_succn_proof の命題は 1 を足したものに相当する命令が存在することを示す部分と，計算を進めることによってその命令を得られることを示す部分に分かれているので，これを使うためには分解する必要がある．

```
  - edestruct (inst_succn_proof m i1) as
      [i2 [H1 H2]]; auto.
```

```
...
  i2 : inst
  H1 : instnat_spec m.+1 i2
  H2 : (i1 :: ?1744, inst_succn :: ?1745)
       |=>* (i2 :: ?1744, ?1745)
  ============================
   exists i : inst,
     instnat_spec (n.+1 + m) i /\
     (i1 :: vs, ?1615 :: nseq n ?1615 ++ cs)
     |=>* (i :: vs, cs)
```

この操作によって得られた H2 を適用する．

```
    evalpartial H2.
```

```
...
  ============================
   exists i : inst,
     instnat_spec (n.+1 + m) i /\
     (i2 :: vs, nseq n inst_succn ++ cs)
     |=>* (i :: vs, cs)
```

残りは帰納法の仮定 IH から明らかである．

```
    by rewrite addSnnS; apply IH.
Defined.
```

これとほぼ同様の方法を用いて，自然数の乗算，偶奇判定，0 との比較，前者関数，減算，大小比較，除算，最大公約数の計算などが記述できる．

9. PostScript プログラムへの変換

本節では，PS0 プログラムを PostScript プログラムに変換する方法と，それによって得られる PostScript プログラムを GNU M4 によって別の PostScript プログラムに埋め込む方法を示す．

以下の pscode_of_inst 関数によって，PS0 の命令を PostScript プログラムとして解釈可能な文字列に変換できる．

```
Fixpoint pscode_of_inst (i : inst) : string :=
  match i with
    | instpop        => "pop"
    | instcopy       => "dup"
    | instswap       => "exch"
    | instcons       => "cons"
    | instquote      => "quote"
    | instexec       => "exec"
    | instpush i     =>
      "{" ++ pscode_of_inst i ++ "}"
    | instpair i1 i2 =>
      pscode_of_inst i1 ++ " " ++ pscode_of_inst i2
  end%string.
```

これで，以下のように Coq 上で PS0 の命令を PostScript プログラムに変換できるようになった．

```
Eval compute in (pscode_of_inst (instnat 3)).
```

```
     = "dup dup dup pop {{pop} pop} exch cons exch
        cons exch cons"%string
     : string
```

ただし，pair, cons, quote は PostScript の命令として元々存在するものではないので，PostScript 側で以下の定義が必要になる．

```
/pair { [ 3 2 roll /exec cvx 4 3 roll /exec cvx ]
        cvx exec } def
/cons { [ 3 1 roll /pair cvx ] cvx } def
/quote { [ exch ] cvx } def
```

bash スクリプト coq2ps によって，PS0 プログラムから PostScript プログラムへの変換ができるようにした．これは GNU M4 から呼び出すためのものであるが，単体でも使える．このスクリプトは，-m オプションで必要なモジュールを渡し，最後に型が inst となるような Coq の項を渡すことで呼び出せる．

```
$ ./coq2ps -m 'FormalPS.Template,ssreflect,seq' \
  'proj1_sig (exists_inst_fill_template 3
  [:: instthole 2; instthole 1; instthole 0] [::])'
{{pop} pop} quote exch quote dup ... pop exec
```

これによって得られたプログラムは，そのまま Ghostscript で実行できる．

```
$ gs -sDEVICE=nullpage -q lib.ps
GS>/a /b /c pstack
/c
/b
/a
GS<3>{{pop} pop} quote exch quote dup ... pop exec
GS<3>pstack
/a
/b
/c
GS<3>
```

以下の GNU M4 マクロによって，PostScript プログラムに PS0 プログラムを埋め込めるようになる．

```
define(‘EMBED’,
    ‘syscmd(./coq2ps -m "MODULES" "$1" | tr -d "\n
        ")dnl’
)dnl
define(‘SET_MODULES’,
    ‘define(‘MODULES’, ‘$1’)dnl’
)dnl
define(‘ADD_MODULES’,
    ‘define(‘MODULES’, MODULES ‘$1’)dnl’
)dnl
SET_MODULES(‘’)dnl
```

PS0 のブール値に関するモジュールを PostScript のライブラリとして使えるようにするためには，以下のように記述する．

```
% boolean values ADD_MODULES(FormalPS.Bool)

/boolfalse  {EMBED(instbool false)} def
/booltrue   {EMBED(instbool true)} def
/boolnot    {EMBED(instnot)} def
/boolif     {EMBED(instif)} def
/boolexecif {EMBED(instexecif)} def
...

/boolenc { { booltrue }{ boolfalse } ifelse } def
/booldec { { true }{ false } boolexecif } def
```

ADD_MODULES マクロによって，Coq の項を目的の PS0 プログラムとして正しく解釈するのに必要なモジュールを追加できる．EMBED マクロによって，そこに書いた Coq の項を PostScript プログラムに変換して埋め込める．boolenc 命令は PostScript のブール値を PS0 のブール値に変換する命令であり，booldec 命令はその逆である．このような命令は，自然数に対しても以下のように記述できる．

```
/natenc {
  {EMBED(instnat 0)} exch { natsucc } repeat } def
/natdec { 0 exch { 1 add } exch exec exec } def
```

このライブラリを使うことで，証明付きの PostScript プログラムを実際に Ghostscript 上で実行できるようになる．

```
GS>12 natenc 21 natenc natadd natdec ==
33
GS>31 natenc 14 natenc natsub natdec ==
17
GS>96 natenc 42 natenc natgcd natdec ==
6
```

10. おわりに

本研究では，PS0 というスタック指向プログラミング言語を定義し，Coq 上で PS0 の対話的プログラミング環境を実装した．この開発環境はプログラムを書いているときに今書いている部分に対応するスタックの状態が一般的にどういう形をしているかを表示できるが，単にそれを見て楽にプログラムが書けるだけではなく，プログラムを書き終えた時点でそのプログラムが目的の仕様を満たしていることが Coq によって保証される．また，スタックの一般的な形を表現するための枠組みを何も無い状態から構築するのは大きな手間がかかると考えられるが，それらも Coq の持つ論理や Coq 上で記述した関数を用いて表現できているため，全体として非常にシンプルな実装になっている．

また，PS0 プログラムを PostScript プログラムに変換し，埋め込むための仕組みをシェルスクリプトと GNU M4 を用いて実装した．Coq 上で記述した自然数やブール値を扱うための PS0 のライブラリを PostScript プログラムに変換し，Ghostscript などの PostScript インタプリタ上で実行できることを確認した．

参考文献

[1] Gonthier, G., Mahboubi, A. and Tassi, E.: A Small Scale Reflection Extension for the Coq system, Research report, INRIA (2008).

[2] The Coq Development Team: *The Coq Proof Assistant Reference Manual* (2013).
available at http://coq.inria.fr/V8.4pl4/refman/.

[3] 坂口和彦 : formalized-postscript (2012–2014).
available at https://github.com/pi8027/formalized-postscript.