

Stricter.org

Coqによる定理証明

Coqでスタック指向プログラミング

坂口和彦 著

前書き

本書は、PostScript のようなスタック指向のプログラミング言語を定理証明器 CoQ^{*1}を用いて形式化し、その上で意味のある計算を構成し、証明を付けることを目的とした本です。対象読者としては、CoQ である程度プログラムや証明を記述できる方を想定しています。

PostScript は Adobe Systems 社が開発したプリンタ向けのページ記述言語であり、今でも多くのプリンタ上で PostScript インタプリタが動くことで印刷物が作られています。一般的に目に触れる範囲ではベクタ画像の形式として知られていましたが、今ではその役割のほとんどを PDF に奪われてしまいました。しかし、PostScript プログラミングにはパズルを解くような面白さがあり、私は今でもこのパズルの解き方を学ぶことには大きな意味があると考えています。

PostScript は他の多くのプログラミング言語と比較して、特徴的な部分が数多くあります。

プログラムが木構造を持たない 多くのプログラミング言語では、プログラムは木構造を持ちます。多くの場合それは括弧の対応によって表わされます。PostScript にも{と}の対応による木構造がありますが、これは糖衣構文として実装でき、適切な変換によって完全に排除できます。

スタックを意識してプログラムを書く PostScript にはオペランドスタックが 1 つあり、基本的にはそのスタック上に計算の対象となる値を積み、操作することで計算を進めます。普通の言語では値の入れ替えは簡単にできますが、PostScript ではそれなりに難しい場合もあります。

メタプログラミング PostScript における命令列の実体は配列です^{*2}。配列を作るのと同じ方法で命令列を作り、それを実行することができます。

^{*1} <http://coq.inria.fr/>

^{*2} ただし通常の配列は実行できないので、明示的に実行可能にしなければなりません。

目次

前書き	i
第 1 章 PostScript の基礎	1
1.1 図を描く	1
1.2 スタックを意識してプログラムを書く	2
1.3 メタプログラミング	2
1.4 複雑なプログラムの例	3
第 2 章 スタック指向言語を作る —— Module: PsCore	5
2.1 使用するモジュール	5
2.2 言語の定義	5
2.3 計算が満たす性質	8
2.3.1 <code>decide_eval</code>	8
2.3.2 <code>eval_uniqueness</code>	9
2.3.3 <code>evalrtc_confluence</code>	9
2.3.4 <code>eval_apptail, evalrtc_apptail</code>	10
2.4 評価器による自動証明	11
2.5 手動証明のためのタクティク	13
2.5.1 <code>evalpartial</code> タクティク	13
2.5.2 <code>rtcrefl</code> タクティク	14
2.6 基本的な命令	14
2.6.1 何もしない命令	14
2.6.2 <code>instsnoc</code> 命令	15
2.7 リストで命令列を表現する	15

第 3 章	ブール型を作る — Module: PsBool	17
3.1	真偽値の仕様	17
3.2	真偽値の仕様を満たす命令	18
3.3	真偽値を用いる計算	18
3.3.1	排他的論理和	18
3.3.2	否定	19
3.3.3	分岐	19
第 4 章	自然数を作る — Module: PsNat	21
4.1	自然数の仕様	21
4.2	クオートされた自然数	21
4.3	自然数の仕様を満たす命令	22
4.4	二種類の表現を行き来する	23
4.4.1	自然数からクオートされた自然数へ	23
4.4.2	クオートされた自然数から自然数へ	23
4.5	自然数の定義の妥当性	24
4.6	後者関数	25
4.7	加算	26
4.8	乗算	27
4.9	偶奇判定	27
4.10	ゼロとの比較	28
4.11	前者関数	29
4.12	減算	30
4.13	大小比較	30
4.14	除算	31
第 5 章	PostScript への変換	35
5.1	基本的な命令	35
5.2	ブール型に関する命令	35
5.3	自然数に関する命令	36

5.4	実行例	37
付録 A	共通ライブラリ —— Module: Common	39
A.1	リストの表記	39
A.2	replicate 関数	39
A.3	反射推移閉包に関する補題	40
参考文献		41
後書き		43

第 1 章

PostScript の基礎

この章では、PostScript プログラミングの基礎的な部分について説明します。これは、本書が何を目標としているかということの具体例を含んだ説明にもなっています。

λ 1.1 図を描く

PostScript はベクタ画像を記述することに特化したプログラミング言語です。通常のプログラミング言語であれば図の記述には非標準のライブラリを用いることが多いようですが、PostScript であれば最初から読み込まれている標準ライブラリだけを用いて非常に簡単に図形を記述でき、しかもそれほどまで拡大しても滑らかな図形となります。例えば、

```
0 0 moveto 100 100 lineto stroke
```

と書くだけで (0, 0) と (100, 100) を端点とする線分を描画できます。

PostScript のプログラムは先頭のトークンから順番に解釈されます。PostScript はスタック指向の言語ですから、命令を実行する前にそのオペランドをスタックに積んでおきます。上の例では、2つの 0 は `moveto` 命令のオペランドであり、2つの 100 は `lineto` 命令のオペランドです。どちらも、直後の命令が実行される前に一度スタックに積まれているのです。`moveto` 命令はパスの始点を設定する命令です。`lineto` 命令はパスの末尾に線分を追加します。この2つの命令はどちらも2つの数値をオペランドとして取ります。`stroke` 命令は、それが実行される前までに作成されたパスを実際に描画します。上のプログラムには、

```
100 100 0 0 moveto lineto stroke
```

になるようにオペランドスタックにプッシュします。PostScript の `exch` 命令に対応します。

instcons 命令 オペランドスタックの先頭の 2 つの値を取り出し、後述の `instpair` 命令を用いて組を作り、オペランドスタックにプッシュします。PostScript の命令列 `[3 1 roll /exec cvx exch /exec cvx] cvx` に対応します。

instquote 命令 オペランドスタックの先頭を取り出し、後述の `instpush` 命令を用いて値を作り、オペランドスタックにプッシュします。PostScript の命令列 `[exch] [exch {} /forall cvx] cvx` に対応します。

instexec 命令 オペランドスタックの先頭を取り出し、継続スタックの先頭にプッシュします。PostScript の `exec` 命令に対応します。

instpush i 命令 i をオペランドスタックにプッシュします。PostScript の `{ i }` に対応します。

instpair $i_1 i_2$ 命令 i_1 と i_2 を継続スタックにプッシュします。PostScript の命令列 `$i_1 i_2$` に対応します。

本書ではこの 8 つの命令だけで計算を構成します。

`eval` は 1 ステップ分の計算しか表せないので、より一般的な計算はそれの 0 回以上の繰返し (反射推移閉包) によって表します。これを `evalrtc` とします。

```
Definition evalrtc : relation environment :=
  clos_refl_trans_in environment eval.
```

最後に、`eval` と `evalrtc` に新しい表記を与えましょう。

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

$e1 \mid \Rightarrow e2$ で `eval e1 e2` と同じ意味に、 $e1 \mid \Rightarrow^* e2$ で `evalrtc e1 e2` と同じ意味になります。本書ではこれらをそれぞれ \Rightarrow と \Rightarrow^* で表すことがあります。

ブール型を作る

PsCore モジュールで定義した言語の上で簡単な計算をするため、まずはブール型を作ってみましょう。

```
Require Import ssreflect Common PsCore Basics Relations List.
```

λ 3.1 真偽値の仕様

真偽値の仕様に定義します。偽を `instnop` として真を `instswap` とすると真偽値での分岐は容易になりますが、それぞれの命令としての振舞いを仕様とすると、`instpair` で合成するだけで排他的論理和が計算できるようになります。真と偽それぞれの仕様は、

```
Definition instfalse_spec (i1 : inst) : Prop :=
  forall i2 i3 vs cs,
    (i3 :: i2 :: vs, i1 :: cs) |=>* (i3 :: i2 :: vs, cs).
```

```
Definition insttrue_spec (i1 : inst) : Prop :=
  forall i2 i3 vs cs,
    (i3 :: i2 :: vs, i1 :: cs) |=>* (i2 :: i3 :: vs, cs).
```

と書けます。

命令がある真偽値としての仕様を満たすという述語は、

```
Definition instbool_spec (b : bool) (i : inst) :=
  if b then insttrue_spec i else instfalse_spec i.
```

と書けます。