PVS の紹介

高木 理　　渡邊 宏　　武山 誠

京都産業大学理学部　　産業技術総合研究所

PVS の紹介*

高木 理† 渡邊 宏‡ 武山 誠§

Abstract

本論の目的は，証明支援ツールの 1 つである PVS およびその応用例を紹介することである．

第 1 節および第 2 節において，PVS およびその作業の流れを大まかに説明する．第 3 節および第 4 節では，PVS における最も重要な概念である型および証明に関する説明を行う．第 5 節において，PVS の応用例として，特定の文字列を分類・変換する簡単なプログラムを取り上げ，PVS を用いてそのプログラムの検証を行う．

The purpose of this paper is to introduce PVS, that is one of proof assistant systems, and the application example.

In Sections 1 and 2, we roughly explain PVS and the work flow. In Sections 3 and 4, we review basic concepts of types and proofs that are the most important concepts in PVS. In Section 5, we introduce an application of PVS, that is, we introduce a short program that classifies and interprets certain strings, and verify the program via PVS.

1 PVS の概要

PVS（Prototype Verification System）は SRI（[1] を参照）において開発された対話型の証明支援ツールである[1]．

証明支援ツールとは，特定の数学的議論を計算機上で形式化し，形式化された命題を機械的に演繹（証明）するためのシステムであり，主に次の 2 つの機能を有する．1 つは，数学的概念を表現する機能で，ユーザ (人間) が数学的概念を実装しやすく，理解しやすい言語を提供する．もう 1 つは，形式化された命題を計算機上で形式的に証明するための環境を提供することである．特に，対話的な証明支援ツールは，証明の作成において，各作業ごとにユーザがツールに与える指示に間違いがあるかどうかをツールがチェックし，ユーザは自分の指示の間違いを即座に訂正できる，という利点を持っている．

---

*A short tutorial on PVS.

†Osamu Takaki, 京都産業大学理学部，Faculty of Science, Kyoto Sangyo University.

‡Hiroshi Watanabe, (独) 産業技術総合研究所システム検証研究センター，Research Center for Verification and Semantics (CVS)，National Institute of Advanced Industrial Science and Technology (AIST).

§Makoto Takeyama, (独) 産業技術総合研究所システム検証研究センター，Research Center for Verification and Semantics (CVS)，National Institute of Advanced Industrial Science and Technology (AIST).

[1]開発者の表現を借りると “論理的な形式化および演繹的な検証を行う統合的環境（an intergrated environment for logical specification and deductive verification）” である．

PVS ユーザは，まず対象となる数学的議論を PVS が提供する言語を用いて形式化する．そして，形式化された命題を PVS が提供する環境（演繹体系）のもとで機械的に証明する．

PVS が提供する言語および演繹体系は高階論理に基づいて作られている．より正確には，PVS の言語は（高階の）型付き言語であり，PVS の演繹体系の強さは高階の古典述語論理に相当する．高階の論理とは関数や述語の集まりを領域とする変数を持つ述語論理であり，通常の数学的議論のほとんど全ての部分を，この論理体系において形式化することが出来る．そのため数学的議論の多くの部分を PVS 上で形式化することが出来る．

2 PVS における作業の主な流れ

初めに，PVS ユーザによる基本的作業の流れを紹介する．ここで言う作業とは，特定の数学的議論（数学的概念）を PVS が提供する言語を用いて形式化することと，形式化された証明を構成することである．ユーザは PVS 環境の中で，次の順番で作業を行う．

1. PVS ファイル (拡張子.pvs) を作成する．つまり，対象となる数学的概念を PVS 上の言語を用いて記述し，それを PVS ファイルに記録する．
2. PVS の機能を用いて，文法チェックおよび型チェックを行う．
3. PVS ファイルに記述された命題に対する形式的な証明を PVS の環境の下で構成する．

以下において，上記の 1~3 の内容について順次説明する．

2.1 PVS ファイル

PVS ファイルは次のような理論およびデータ型の宣言によって構成される．(データ型はしばしば「抽象データ型」とも呼ばれる．)

Id [*Theory Formulas*]: THEORY
BEGIN
(Assuming Part)
(Theory Part)
END *Id*

Figure 1: (パラメータ付き) 理論

データ型（図 2）はユーザ自身が新たにデータ型を定義するためのものである．したがって，データ型が記述されていない PVS ファイルもあり得る．(データ型については第 3.4 節で改めて説明する．) 各 PVS ファイルは 1 個以上の理論（図 1）から構成される．

図 1 の *Id* は理論の名前であり，*Theory Formulas* において必要なパラメータを記述する．理論の *Assuming Part* は理論の展開に必要な仮定（公理）を記述す

Id [*Theory Formulas*]: DATATYPE
BEGIN
(Theory Part)
END *Id*

Figure 2: データ型

る．*Theory Part* は理論の本体であり，この中で定義や命題（補題・定理など）が記述される．

なお, 各理論およびデータ型は, 別の理論やデータ型を取り込むこと（IMPORTING）や，逆に取り込ませること（EXPORTING）が出来る．例えば，“A” という理論の中で別の理論 “B” の内容を使いたいときは，A の中に

```
IMPORTING B
```

という記述を加える．

2.2 例その 1: sum.pvs

PVS ファイルの例として，1 つの理論だけからなるファイル (sum.pvs) を挙げよう．

```
sum: THEORY
BEGIN
  n: VAR nat
  f,g: VAR [nat -> nat]
  sum(f,n): RECURSIVE nat =
        (IF n=0
               THEN 0
               ELSE f(n-1) + sum(f, n-1)
        ENDIF)
        MEASURE n
  square(n): nat = n*n
  sum_of_square: LEMMA
        6 * sum(square, n+1)
        = n * (n+1) * (2*n+1)
END sum
```

Figure 3: sum.pvs

この例は，PVS パッケージの中の例の 1 つである sum2.pvs の一部を抜粋したものである．

ここで “sum” は理論の名前であり，この理論は数列の有限和 $\Sigma_{k<(n+1)}k^2$ に関する性質を形式化したものである．

理論名 sum の後ろに括弧が無いので，これはパラメータの無い理論である．また，この理論 sum には仮定部分（Assuming Part）は無く，“BEGIN” の直後から “END” の直前までが理論部分（Theory Part）に相当する．（パラメータや仮定部分のついた PVS ファイルの例は，第 3.3 節で紹介する．）

理論部分において，まず

```
n: VAR nat
```

によって自然数上の変数 n が定義される．“nat” は PVS の基本ライブラリである prelude.pvs において定義されている型であり，自然数の集まりを意味する．（型や prelude.pvs については後の節で説明する．）さらに f,g: VAR [nat -> nat] によって，nat からなる関数型 [nat->nat] を型に持つ変数 f および g が定義される．

次に，

```
sum(f,n): RECURSIVE nat =
    (IF n=0
       THEN 0
       ELSE f(n-1) + sum(f, n-1)
    ENDIF)
    MEASURE n
```

によって，自然数 n と関数 f を受け取り，f の各値を第 0 項から第 n-1 項まで加算した合計を出力する関数 sum を再帰的に定義する．キーワード “RECURSIVE” は再帰的定義であることを意味する．

上記の “MEASURE” は再帰的定義の帰納系（整列順序集合）を指定する．上の例では，sum(f,n) を n に関して再帰的に定義することを意味している．一般に，“MEASURE n” と記述すれば n の領域を帰納系として指定することになるのだが，この帰納系の順序を明示したい場合は

```
MEASURE n by (n の領域に与えられている順序)
```

と書く．

この定義の後の

```
square(n): nat = n*n
```

によって，引数 n に対する square の返り値を指定して，関数 square を定義している．コロンの後ろの “nat” は square の返り値の型が nat であることを意味する．square は [nat -> nat] 型の関数であり，次のように直接この型を明示して定義することも出来る．

```
square: [nat -> nat] = LAMBDA(x: nat): x*x
```

その後の

```
sum_of_square: LEMMA
    6 * sum(square, n+1)
    = n * (n+1) * (2*n+1)
```

で sum に関する性質が補題 (sum_of_square) として記述されている．ユーザはこの補題に対して形式的な証明を与えなくてはならない．形式的証明の仕方は第 4 節で改めて説明する．

2.3 prelude.pvs

PVS 環境の中には，prelude.pvs という名前を持つ PVS ファイルが予め存在する．このファイルの中には，集合や数などの基本的な数学的概念が形式化されている．ユーザは基本的な概念に関しては，自分で定義する必要はなく，prelude.pvs を利用できる．このファイルは何も指定せずに勝手に取り込まれるので，“IMPORTING prelude.pvs” と明示する必要はない．

本稿は prelude.pvs に関して，これ以上の説明は行わない．詳細は，[3] を参照されたい．

2.4 文法チェックおよび型チェック

PVS ファイルを作成したら，ユーザはそのファイルに矛盾が無いかどうかを確かめるために，文法チェックおよび型チェックを行う．

まず文法チェックによって，PVS ファイルの中の記述に文法的な誤りがないかどうかをチェックする．このときのチェックは記号的なものであり，数学的な内容はチェックされない．

次に，PVS ファイルの型チェックを行う．型チェックによって，PVS 上で定義された数学的対象（の意味的な条件）を解析し，定義に矛盾や不備が無いかどうかが確かめられる．ここで何らかの不備が見つかると，PVS はエラーを表示してユーザに対して修正を要求する．

さらには，型チェックにおいて，PVS ファイルの定義や性質に関して，それらが形式的に議論する上で十分かどうかがチェックされる．例えば，A という概念を定義したとき，A の定義が整合的（well-defined）かどうかがチェックされる．そして A が整合的に定義されるための条件などが十分でなかったときは，必要な条件が義務命題 (OBLIGATION) として生成され，TCC（Type-Correctness Condition）にまとめられる．義務命題は，上記の型チェックのエラーとは違い，その場で解決する事を強制されるものではない．しかし，PVS ファイルを完成させる際には，これらの義務命題はすべて証明しなくてはならない．

2.5 PVS における証明の作成

最後の段階として，PVS ファイルの中にある命題と，PVS が自動的に生成した義務命題を証明する．ユーザは PVS が提供する環境のもとで証明作業を行う．

この証明作業の様子を第 4 節で説明する．

3 型（type）について

PVS の言語の持つ型の概念について説明する．PVS で型とは集合を意味する．つまり，型 A を持つ項は A の表す集合の要素と見なされる．2 つの型が等しいのは，それらの型が持つ項が等しいときである．

3.1 型の構成

以下の基本型と六種類の型の構成方法がある.

基本型 [2]

自然数全体を指す型 `nat` やブール値を指す型 `bool` など. ユーザはこれらの型を既に定義されたものとして使うことが出来る. ユーザが作った PVS ファイルの中で, 基本型と同じ名前で新たに型を宣言した場合は, 新しい定義が採用される.

数え上げによる型: `{e_1,e_2,...,e_n}`

項 `e_1,e_2,...,e_n` を持つ型.

述語によって表現される部分型: `{x:T|p(x)}`

PVS においては, 既存の型に対して, その型が表す集合の部分集合を表す部分型 (subtype) が定義できる. 例えば, `p` が型 `T` 上の述語 (型 `[T -> bool]` を持つ項) であるとき, `p(x)` を満たす `T` の要素 `x` を集めた部分集合の型を, `{x:T|p(x)}` と書く. これを `(p)` と略記することも出来る.

関数型: `[X -> Y]`

与えられた型 `X` から別の型 `Y` への関数全体を意味する.

組型: `[X_1,X_2,...,X_n]`

型 `X_1,...,X_n` の直積を意味する. この型の要素は $X_1, \ldots, X_n$ の要素の組 $(x_1, \ldots, x_n)$ からなる. T が組型 `[X_1,...,X_n]` のとき, `proj_i`(T) によって型 `X_i` を表現することが出来る.

レコード型: `[# a_1:X_1,...,a_n:X_n #]`

この型の要素も与えられた型 `X_1,...,X_n` の各要素から成るのだが, 組型と異なり要素の順番は考慮されない. その代わり各 `a_i` が `X_i` を呼び出す役割を果たす (このような `a_i` は記録到達子 (record accessor) と呼ばれる). つまり, `T` がこの形の型であるときは, `a_i(T)` によって型 `X_i` を表現することが出来る. (この型の使用例については付録 A を参照されたし.)

抽象データ型: 第 3.4 節で改めて説明する.

3.2 型宣言

型の宣言は, 大きく分けて以下の 2 種類のものがある.

解釈の無い型宣言: 新たな型を, 集合として一意的に定めず, 何らかの集合を表す型として宣言するとき, このような宣言を行う. 例えば,

```
T: TYPE
```

[2] [4] には "基本型" という記述はないが, 便宜上ここでは prelude.pvs の中に定義されている型を基本型と呼ぶことにする.

によって T を何らかの集合を表す型として宣言することが出来る．また，既存の型 T の部分型を解釈無しで宣言したい場合は，“S: TYPE FROM T”とする．

解釈付きの型宣言: すでに定義された幾つかの型を組み合わせて新たな型を宣言することが出来る．その場合は以下のように（新たな型 T を）宣言する．

T:TYPE = 既存の型の組み合わせによる表現

例えば T を既存の型 nat からなる関数型として宣言したいときは，“T: TYPE = [nat->nat]”とすればよい．

型と集合を同一視する PVS においては，有限集合などをその要素を指定することにより定義する要領で，型を宣言することが出来る．例えば，T を三点集合 r，g，b として型宣言したいときは，T: TYPE = {r,g,b}とすればよい．

(注意) PVS においては，宣言された型は集合と見なされるのだが，それが空集合でないという保証はなされない．そのため，空でない集合を用いた数学的議論を PVS 上で形式化する場合は,（部分）型が空集合を意味しないことを指定する必要が出てくる．そのような場合，例えば，nat の部分型で 0 を含むようなものは

S: TYPE FROM nat CONTAINING 0

と宣言できる．また，空でない何らかの集合を表す型として NONEMPTY_TYPE が用意されている．

3.3 例その 2: group.pvs

型の説明のための例として，1 つの理論だけからなるファイル (group.pvs) を取り上げる．

```
Groups [G : TYPE,
e : G,
o : [G,G->G],
inv : [G->G] ] : THEORY
BEGIN
ASSUMING
  a, b, c : VAR G
  associativity :  ASSUMPTION
      a o (b o c) = (a o b) o c
  unit : ASSUMPTION
      e o a = a AND a o e = a
  inverse : ASSUMPTION
      inv(a) o a = e AND a o inv(a) = e
ENDASSUMING
  left_cancellation: THEOREM
```

```
      a o b = a o c IMPLIES b = c
  right_cancellation: THEOREM
      b o a = c o a IMPLIES b = c
END Groups
```

Figure 4: group.pvs

この例は，PVS パッケージの中の例の 1 つであり，[4] の 60 ページにおいても紹介されている．

この理論は群の定義（公理も含めて）および簡単な命題を記述したものである．

“Groups” という理論の名前の直後に，パラメータが 4 つ与えられている．

```
[G : TYPE,
e : G,
o : [G,G->G],
inv : [G->G] ] ...
```

最初のパラメータは解釈無しの型宣言 “G:TYPE” によって与えられる型 G である．第 2 のパラメータは “e:G” によって与えられる定数（constant）e であり，これは G を型に持つ．さらに，第 3 のパラメータが型 [G,G -> G] を持つ定数 o として与えられている．ここで “[G,G -> G]” は “[[G,G] -> G]” の略記である．第 4 のパラメータ “inv” は関数型 [G->G] を型に持つ．

これらのパラメータは別の理論から理論 Groups を引用する際に用いる．

上記の理論 Groups における “ASSUMING” および “ENDASSUMING” で囲まれた部分が第 2.1 節で説明した仮定部分に相当する．この部分ではキーワード “ASSUMPTION” を使って，仮定を記述する．ASSUMPTION は LEMMA や THEOREM と同様に論理式である．理論 Groups が別の理論 T によって読み込まれたとき，理論 Groups の ASSUMPTION の論理式は T における TCCs の中に義務命題として組み込まれる（第 2.4 節を参照）．これは PVS が ASSUMPTION の内容を証明していない命題として認識していることを意味している．

3.4 抽象データ型

型を集合と見なす PVS においては，帰納的定義により集合を生成するのと同様に型を生成することが出来る．このようにして出来る型を抽象データ型と呼ぶ．

抽象データ型を説明するために，以下の帰納的定義により生成される集合 *Term* を例として挙げる．

(i) vari は *Term* の元である．
(ii) s が *Term* の元ならば f(s) も *Term* の元である．
(iii) *Term* の元は (i)-(ii) 以外には作られない．

このような集合 *Term* を PVS で形式化したいときは，次の図（図 5）のような抽象データ型を定義すればよい．

```
term: DATATYPE
BEGIN
  vari: vari?
  f(content: term): nonvari?
END term
```

Figure 5: *Term* に対する抽象データ型（完全版）

図 5 の “vari” および “f” は，この抽象データ型の要素を生成するための演算子である．このような要素を構成子（constructor）と呼ぶ．

図 5 の “vari?” および “nonvari?” は，この抽象データ型の各要素が vari と f の内で最終的にはどちらの構成子によって作られたのかを確認するためのものである．上の例では，vari のみが vari?を満たし，それ以外の要素（fs の形をとる）は nonvari?を満たす．このような vari?および nonvari?を認識子（recognizer）と呼ぶ．

図 5 の “content” は，f によって構成される要素がどのような要素から作られるのかを示すためのものである．上の例では，nonvari?を満たす t は fs という形を持つはずなので，content(t) は s となる．このような content を到達子（accessor）と呼ぶ．

一般に抽象データ型の定義は，構成子，認識子および到達子によって構成される．構成子は抽象データ型 D を集合と見なしたとき，D の各要素を生成するための演算子である．認識子は各要素がどの構成子により生成されたのかを示す演算子である．そして，到達子は各要素がどの要素から（認識子が示す構成子によって）生成されたのかを示す演算子である．

抽象データ型の定義は，第 2.1 節のように各理論（THEORY）と並行して配置されるか，ある理論の中に組み込まれる形で配置される．PVS ファイル P.pvs において，抽象データ型 D を理論と並行して記述したとき，P.pvs の型チェックを行った際に，PVS は自動的にある PVS ファイルを作成する（この場合は P_adt.pvs という名前の PVS ファイルが作成される）．この PVS ファイルの中に，D を再帰的に定義された集合として扱うための一連の命題によって構成された新たな理論が，PVS によって自動的に作成される．

3.5 例その 3: term.pvs

抽象データ型を用いた PVS ファイルの例として，抽象データ型および理論からなるファイル (term.pvs) を考える．term.pvs の内容は以下の通りである．

```
term: DATATYPE
BEGIN
  vari: vari?
  f(content: term): nonvari?
END term

property_of_term: THEORY
```

```
BEGIN
IMPORTING term
  t: VAR term
  length(t): RECURSIVE nat =
    IF vari?(t)
      THEN 0
      ELSE length(content(t))+1
    ENDIF
    MEASURE t BY <<
  length_of_term: THEOREM
    FORALL (s: term):
      length(f(t))=length(t)+1
END property_of_term
```

Figure 6: term.pvs

このファイルは，“term” という抽象データ型宣言と理論 property_of_term からなり，property_of_term は term を取り入れている．この理論の中の term の長さを出力する関数 length は，term の各要素の構成に関する再帰的定義によって定義されている．ここで “MEASURE t BY <<” の “<<” は抽象データ型 term の構成（定義）に関する順序を意味する（MEASURE については第 2.2 を参照されたい）．

4 PVS における証明

PVS ファイルを作成し，文法チェックと型チェックが終われば，最後の作業として PVS ファイルの中の命題（LEMMA，THEOREM など）を証明する．PVS における証明とは，PVS 特有の証明図（形式化された証明）を作ることに他ならない．

PVS における証明図について説明しよう．そのためにまず準備を行う．

4.1 PVS におけるシークエント

ここでは PVS が提供する言語によって適当な論理式が定義されているとして話を進める．

論理式 $\varphi_1, \ldots, \varphi_n, \psi_1, \ldots, \psi_m$ について，次の形の記号列をシークエントと呼ぶ．

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m \tag{1}$$

(1) において，記号 “$\vdash$” より左側の論理式の列 $\varphi_1, \ldots, \varphi_n$ を前提といい，右側の列 $\psi_1, \ldots, \psi_m$ を結論という．ここで n, m はそれぞれ 0 以上の自然数である．つまり前提や結論が空の列になることもある．

シークエント $\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m$ の真偽値は論理式 $\varphi_1 \wedge \cdots \wedge \varphi_n \Rightarrow \psi_1 \vee \cdots \vee \psi_m$ の真偽値として定義される[3].

シークエント S $(= \varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m)$ に対して，ある $i \leqq n$ および $j \leqq m$ が存在して φ_i と ψ_j が記号列として等しいとき，S は自明であると言う．自明なシークエントは明らかに真（valid）である．

4.2 コマンド

PVS のコマンドは，シークエントと幾つかのパラメータを引数に持ち，シークエントの有限列を出力する写像である．後の節で，代表的なコマンドを幾つか説明する．

シークエント S が，コマンド C によって，シークエントの有限列 $S_1 \cdots S_n$ に写されるとき，$S \to_C S_1 \cdots S_n$ と表すことにする．また，シークエント S および $S_1 \cdots S_n$ について，$S \to_C S_1 \cdots S_n$ が成り立つようなコマンド C が存在するとき，$S \to_* S_1 \cdots S_n$ と表す．後で説明する代表的なコマンドの説明から $S \to_* S_1 \cdots S_n$ のとき，

$$S_1 \wedge S_2 \wedge \cdots \wedge S_n \text{ が真ならば } S \text{ も真}$$

となることが容易に分かるだろう．

4.3 PVS における証明図

各シークエント S に対して，PVS における S の証明図は以下の条件を満たす有限木 T のことである．

1. T の各ノードはシークエントである．特に，T の根（root）はシークエント S である．
2. T の各ノード S_0 について，S_0 が子のノード $S_1, \ldots, S_n$ を持つとき，$S_0 \to_* S_1 \cdots S_n$ が成り立つ．[4]
3. T のリーフは自明なシークエントであり，それ以外のノードは自明なシークエントではない．

一般に，シークエントに対する証明図は存在しても一意的に存在するわけではない．

PVS においてシークエント S を証明することは，S の証明図を構成することである．PVS における証明作りは，PVS との対話により行われる．最初に，証明したいシークエントを根とする．根から順次，ノードにコマンドを適用して，子のノードを生成する．生成されたノードが自明なシークエントでない限り，コマンドを新たに適用して子を生成する．証明図になるまでこの作業を続ける．

実際の証明作業で，PVS が提示するシークエントに対して，適当なコマンドを入力して証明図を構成する様子を，第 4.5 節で簡単に述べたい．その前に，証明図の構成に用いる主要なコマンドを説明する．

---

[3] ここで $\wedge$ は “かつ”，$\vee$ は “または”，$\Rightarrow$ は “ならば” を意味する論理結合記号である．このチュートリアルで扱う論理結合記号は古典論理に関するもの（$\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$）のみである．なお，PVS においては，これらの論理結合記号を表現するための独自の記法が用意されているが，このチュートリアルでは，そのような記法の説明は省略する．

[4] $S_1 \cdots S_n$ は，S の子のノードを左から順番に並べたものとする．

4.4 PVS 上のコマンドの概説

PVS では非常に多くのコマンドが用意されており，本稿の中ですべてのコマンドを説明することは出来ない．ここでは主要なコマンドを 8 つだけ説明する．

assert 証明における作業のうち，決定可能（decidable）な操作（あるいは自明な操作）は assert を用いることによって、ユーザの代わりに PVS に適当に処理させることが出来る．例えば，あるノードにおいて，φ_i または ψ_j が $t = 1 + 2 + 3$ という形をしていて，なおかつ $t = 6$ のときに求めるノードが得られるような場合，assert を行うと自動的に求めたいノードを得ることが出来る．

flatten シークエントの前提あるいは結論のなかの論理式の論理結合子を除いて，論理式の複雑さを弱めるコマンドである．コマンド flatten はシークエント (1) を次のシークエントに写す．

- 前提の中の φ_i が $A \wedge B$ の形のとき:

$$A, B, \varphi_1, \ldots \langle i \rangle \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m,$$

  ここで $\varphi_1, \ldots \langle i \rangle \ldots, \varphi_n$ は

$$\varphi_1, \ldots \varphi_{i-1}, \varphi_{i+1}, \ldots, \varphi_n$$

  の略記である．

- 結論中の φ_j が $A \vee B$ の形のとき:

$$\varphi_1, \ldots, \varphi_n \vdash A, B, \psi_1, \ldots \langle j \rangle \ldots, \psi_m,$$

- 結論の中の φ_j が $A \Rightarrow B$ の形のとき:

$$A, \varphi_1, \ldots, \varphi_n \vdash B, \psi_1, \ldots \langle j \rangle \ldots, \psi_m.$$

split コマンド split はシークエント (1) を次のように 2 つのシークエントに写す：

- 前提の中の φ_i が $A \vee B$ の形のとき，

$$A, \varphi_1, \ldots \langle i \rangle \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m,$$
$$B, \varphi_1, \ldots \langle i \rangle \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m.$$

- 前提の中の φ_i が $A \Rightarrow B$ の形のとき，

$$\varphi_1, \ldots \langle i \rangle \ldots, \varphi_n \vdash A, \psi_1, \ldots, \psi_m,$$
$$B, \varphi_1, \ldots \langle i \rangle \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m.$$

- 結論の中の ψ_j が $A \wedge B$ の形のとき，

$$\varphi_1, \ldots, \varphi_n \vdash A, \psi_1, \ldots \langle j \rangle \ldots, \psi_m,$$
$$\varphi_1, \ldots, \varphi_n \vdash B, \psi_1, \ldots \langle j \rangle \ldots, \psi_m.$$

lemma これまでに作成した命題，あるいは prelude.pvs ファイルにある命題を用いるときに使う．ここで prelude.pvs ファイルはライブラリの役割を果たす．PVS では，証明されていない命題でも，lemma コマンドの対象として使える点に注意されたい．

skolem 前提の中の φ_i が $\exists x.A[x]$ の形をなすとき，skolem コマンドによって φ_i を $A[x_0]$ に置き換えることが出来る．ここで x_0 は証明において用いられていない新たな変数である．結論の中の ψ_i が $\forall x.A[x]$ の形をなすときも同様である．

inst 前提の中の φ_i が $\forall x.A[x]$ の形をなすとき，inst コマンドによって φ_i を $A[t]$ に置き換えることが出来る．ここで t はユーザが指定する適当な term である．ψ_i が $\exists x.A[x]$ の形をなすときも同様である．

replace 前提の中のある論理式 φ_i が等式 $t = s$ の形をしているとき，前提の中の φ_j あるいは結論の中の ψ_k に含まれる t あるいは s を，もう一方の項に置き換える（どの t あるいは s を置き換えるのかを指定することが出来る）．

induct シークエント (1) の結論の中の ψ_i が $\forall x.A[x]$ の形をなし，なおかつ x の型が再帰的に定義される型の場合，その帰納系による数学的帰納法を用いる場合に使われる．例えば x の型が `nat` のとき，次の 2 つのシークエントが得られる：

$$\begin{aligned}&\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, A[0], \ldots, \psi_m,\\ &\varphi_1, \ldots, \varphi_n \vdash\\ &\quad \psi_1, \ldots, A[x_0] \Rightarrow A[x_0 + 1], \ldots, \psi_m.\end{aligned}$$

ここで x_0 は新たな変数である．

注意 ところで，PVS では，否定記号 $\neg$ はコマンドなしで自動的に取り除かれる，つまり前提の φ_i が $\neg A$ の形をしているときは，自動的に φ_i が $\varphi_1, \ldots, \varphi_n$ から除かれ，その代わりに A が結論 $\psi_1, \ldots, \psi_m$ に加えられる．結論の中の ψ_i が $\neg A$ の形をしているときも同様に，ψ_i が結論から除かれて A が前提に加えられる．

上記のコマンドを適用すると，前提の φ_i および結論の ψ_j に出現する論理記号 $(\neg, \wedge, \vee, \Rightarrow, \forall, \exists)$ を一番外側のものから次々に取り除けることに注意されたい．

PVS では他にも様々なコマンドが用意されているが，ここではその説明を割愛する．詳しくは参考文献 [6] を参照されたい．

4.5 PVS における証明の例

PVS システム上ではシークエントは一行ではなく，複数行で表現される．例えば

$$\varphi_1, \ldots, \varphi_n \vdash \psi_1, \ldots, \psi_m \tag{2}$$

は PVS の証明モードの中では縦に並べられて，$\varphi_1, \ldots, \varphi_n$ と $\psi_1, \ldots, \psi_m$ は記号

```
                |-------
```

で区切られる．

以下において，第 3.3 節で取り上げた PVS ファイル group.pvs にある定理 “left_cancellation” の証明を例に説明する．

命題の対話的証明モードに入ると，PVS は図 7 を表示する．

```
(中略)
cl-user(1):
cl-user(2): nil
pvs(9):

left_cancellation :

  |-------
{1}   FORALL (a, b, c: G): a o b = a o c
IMPLIES b = c

Rule?
```

Figure 7: 証明モードの出だし部分

これ以降，“Rule?” の直後に適当なコマンドを入力していき，対話的に証明図を構成していく．

対話的な証明作業の全体は，付録 B に記載する．本節では，証明作業の中で注意すべき点を幾つか述べる．

付録の証明では，replace コマンドが多く使われている．例として証明の一部を図 8 で記述する．

ここで，“Rule? (replace -6 (-6 -4))” がコマンドを入力した部分であり，その上側の部分が，このコマンドの引数となるシークエントを表し，下側の部分が，このコマンドによって導き出されたシークエントを表す．

ここで，各シークエント中の各々の論理式に [1] や{-4}のような番号が割り当てられていることに注意されたい．1 つのシークエントに対して，

```
            |-------
```

より上の論理式（つまり (2) のシークエントにおいては左側の論理式 φ_i）には負の整数 [-1]，[-2]. . . が上に位置する論理式から割り当てられる．一方，下側の論理式（つまり (2) のシークエントにおいては右側の論理式 ψ_j）には正の整数 [1] が割り当てられている．特に，直前のコマンドによって新たに生成された論理式については [番号] の代わりに{番号}によって番号付けがなされる．

```
...(前略)...

[-1]  inv(a!1) o (a!1 o b!1)
= (inv(a!1) o a!1) o b!1
```

```
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
{-4}  inv(a!1) o (a!1 o b!1) = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -6 (-6 -4))
Replacing using formula -6,

[-1]  inv(a!1) o (a!1 o b!1)
= (inv(a!1) o a!1) o b!1
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
{-4}  inv(a!1) o (a!1 o c!1) = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

...(以下略)...
```

Figure 8: replace コマンドの例

コマンド “(replace -6 (-6 -4))” は上側のシークエントの “[-6]” に対応する論理式 “a!1 o b!1 = a!1 o c!1” を用いて，上側のシークエントの “{-4}” に対応する論理式 “inv(a!1) o (a!1 o b!1) = b!1” の中の “a!1 o b!1” を “a!1 o c!1” に置き換えることを要求している．コマンドが適用されると下側のシークエントが結果として得られる．

次の例を見てみよう

```
...(前略)...

{-1}  FORALL (a: G): e o a = a AND a o e = a
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "b!1")
Instantiating the top quantifier in -1 with
the terms: b!1,

{-1}  e o b!1 = b!1 AND b!1 o e = b!1
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1
```

```
...(以下略)...
```

Figure 9: inst コマンドの例

上記の “(inst -1 "b!1"))” は，上側の “{-1}” に対応する論理式が

```
FORALL (a: G): e o a = a AND a o e = a
```

なる形をしているので，この中から FORALL を外し e のところに項 “b!1” を代入するコマンドである．下側の{-1}に注意されたい．

付録 B の最後は assert コマンドを用いた以下の図 10 で締めくくられている．

```
...(前略)...

[-1]  e o c!1 = c!1
[-2]  c!1 o e = c!1
[-3]  inv(a!1) o (a!1 o c!1)
= (inv(a!1) o a!1) o c!1
[-4]  inv(a!1) o (a!1 o b!1)
= (inv(a!1) o a!1) o b!1
[-5]  inv(a!1) o a!1 = e
[-6]  a!1 o inv(a!1) = e
{-7}  c!1 = b!1
[-8]  b!1 o e = b!1
[-9]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (assert)
Simplifying, rewriting, and recording
with decision procedures,
Q.E.D.
```

Figure 10: assert コマンドの例

図 10 の{-7}と [1] は（“=” は対称律を満たす通常の同値関係なので）明らかに等しい意味を持つのだが，記号列として等しくないので，PVS は自明なシークエントとして認識していない．そこで，{-7}の “c!1=b!1” を”b!1=c!1” に置き換えたい．このようなとき，等式 “=” の定義にもどり証明を行うことも出来るが，このような決定可能な操作は assert コマンドで解決できる．

実際には，今回の証明（つまり付録 B の証明）はかなり冗漫なものであり，もっと簡潔な証明図を作ることが出来る．例えば assert コマンドをもっと早い段階で使えば，もっと早く証明が完結できる．また，ここで紹介しなかったコマンド（例えば grind など）を使えば，上記の例よりもはるかに小さな証明図を作ることが出来る．grind の簡単な説明および grind を用いた証明の例を付録の第 C 節に記載したので，そちらも参照されたい．

5　応用例: PVS を用いたプロセスの検証

この節では，PVS の応用例として，特定の文字列[5]を分類・変換する小さなプログラム[6] を取り上げ，その振る舞いを PVS を用いて演繹的に検証することを試みる．

5.1　相互再帰的に定義された文字列を翻訳処理するプログラム

プログラミング言語 C の古典的な教科書である [7] の第 5 章において，C における型宣言を表す文字列を日常的文章に翻訳するプログラムが紹介されている．そのプログラムは C における型の構成規則に従い，例えば以下の入力値 (1)∼(3) に対して以下の (i)∼(iii) を出力する．

(入力値)

(1) `char *foo[3]`

(2) `char (*foo)[3]`

(3) `char (*(*foo[3])())[5]`

(出力値)

(i) `foo: array[3] of pointer to char`

(ii) `foo: pointer to array[3] of char`

(iii)
```
foo: array[3] of pointer to function
     returning pointer to array[5] of char
```

そのプログラムの処理の対象の主な部分は`*foo[3]` や `(*(*foo[3])())[5]` などに該当する部分である．このような文字列の集まりを *dcl* で表す．*dcl* はもう 1 つの集合 *dirdcl* と共に相互再帰的に以下のように定義することが出来る．

$$\textbf{\textit{dcl}} \ni d ::= dd \mid *d$$

$$\textbf{\textit{dirdcl}} \ni dd ::= name \mid (d) \mid dd() \mid dd[size]$$

ここで *name* は変数名を表す文字列で *size* は配列の要素数を表すものとする．

ここで [7] の 150 ページで紹介されている，*dcl* を翻訳するプログラム `pdcl` および `pdirdcl` を以下に記述する（以下は C 言語によるプログラム）．

[5] この節では文字列や数列などはすべて有限列とする．

[6] この節では，簡単のため，C 言語によるプログラムや C 言語の関数も単に「プログラム」と呼ぶことにする．

```
char input[1000]   // 入力値が格納される配列
char output[1000]  // 出力値が格納される配列
char name[100]     // 変数名や配列の要素数などを
                   // 一時的に格納するための配列
char *ip;          // 入力ポインタ

... input に入力値を格納 ...

void
pdcl()
{
  int ns=0;
  while (*ip=='*') {
    ns++;
    ip++;
  }
  dir_pdcl();
  while (ns-- >0)
    strcat(output, " pointer to");
}

void
pdirdcl()
{
  int i;
  if (*ip=='(') {
    ip++;
    pdcl();
    if (*ip != ')') {                    // (#)
        printf("right-paren error\n");
        exit(-1);
    }
    ip++;
  }
  else if (isalpha(*ip)) {
    i=0;
    while(isalpha(*ip))
      name[i++]=*ip++;
    name[i]='\0';
    strcat(output, name);
    strcat(output, ":");
  }
  else {
    printf("pdirdcl error\n");
    exit(-1);
  }
  while (*ip=='(' || *ip=='[') {
  if (*ip=='(') {
    strcat(output, " function returning");
    ip+=2;
```

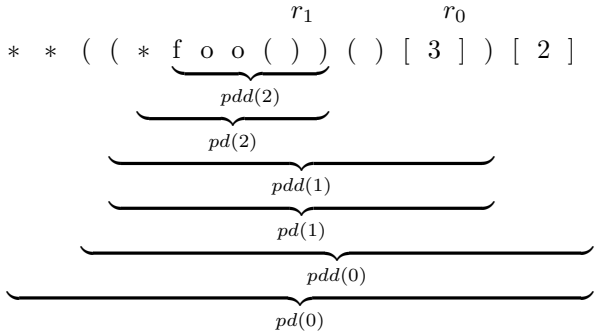
```
  }
  else if (*ip=='[') {
    ip++;
    i=0;
    while(*ip!=']')
      name[i++]=*ip++;
    name[i]='\0';
    strcat(output, " array[");
    strcat(output, name);
    strcat(output, "] of");
    ip++;
  }
 }
}
```

これら 2 つのプログラムは固定された入力ストリームおよび入力ストリーム内のアドレスを表すポインタを参照しながら，互いのプログラムを呼び出し，ポインタ（の指す位置）を移動させながら出力ストリームに書き込みを行うという振る舞いを持つ.（[7] における本来の `pdcl` および `pdirdcl` は入力ストリームとポインタを参照する代わりに，入力のための別のプログラムを呼び出しているが，簡単の為，その部分は書き換えている.）

例えば，入力ストリーム内の文字列が

```
**((*foo())()[3])[2]
```

のとき，`pdcl` と `pdirdcl`（が生成するプロセス）がどのように振舞うのかを以下の図で表す.



上記の図において，各 $pd(i)$ および $pdd(i)$ は `pdcl` および `pdirdcl` によって i 番目に生成されるプロセスを意味する．そして，上記の $\underbrace{\cdots}_{pd(i)}$ および $\underbrace{\cdots}_{pdd(i)}$ は $pd(i)$ および $pdd(i)$ が実行されているときの入力ポインタが指す範囲を表す.

実際のところ，上記のポインタの指す範囲から最後の 1 文字分を除いたところにある文字列が $pd(i)$ および $pdd(i)$ による翻訳処理の対象となることが分かる．そこで，その対象部分となる文字列を $L_{pd}(i)$ および $L_{pdd}(i)$ で表し，取り除かれ

た最後の位置（つまり $pd(i)$ および $pdd(i)$ の実行中に入力ポインタが指す最後の位置）を $T_{pd}(i)$ および $T_{pdd}(i)$ で表す．

さらに，上記の図式の各 r_i はプロセス $pdd(i)$ における条件判定 `(#)` が行われる時点での入力ポインタの指す位置を表す．この例の場合，各 r_i 上に “`)`” が格納されていること，さらに，$r_i = T_{pd}(i+1)$ になっていることに注意されたい．

さて，これらのプログラム `pdcl` および `pdirdcl` に対して以下の点を重視した検証を試みる．

(I) 入力ストリームにある格納された文字列 *str* が *dcl* の要素であるとき，各プロセス $pd(i)$ および $pdd(i)$ が $L_{pd}(i)$ および $L_{pdd}(i)$ を正しく翻訳処理するか? （つまり，各プロセスを実行しているとき，入力ポインタは先の図式に示された範囲内を指しているか?）

(II) 特に，(*str* が *dcl* の要素であるとして）プロセス $pdd(i)$ における条件判定 `(#)` が行われる時点での入力ポインタが “`)`” を指しているか?

(III) 最終的に `pdcl` は，入力値 *str* が *dcl* の元ならば適当な翻訳文を返し，さらに，*str* が *dcl* の元でないならばエラーを返すか?

この後の 2 つの節において，入力データとしての *dcl* および *dirdcl* やプログラム `pdcl` および `pdirdcl` を PVS 上で形式する．その際，検証の対象となる *dcl* や `pdcl` の抽象化（簡略化）を試みる．

5.2 *dcl* の抽象化および PVS による形式化

プログラム `pdcl` および `pdirdcl` は，ある入力ストリーム（文字列）を受け取って何らかの文字列を返す関数と解釈することが出来る．さらに、各文字は自然数でコード化できるので，*dcl* および *dirdcl* は自然数列の集合として，さらに `pdcl` および `pdirdcl` は自然数列上の関数として，形式化することが出来るだろう．しかしながら，今回は簡単のために，これらをそのまま形式化せず，`pdcl` や `pdirdcl` に対する実際のコーディングを眺めながら *dcl* や `pdcl` を適当に抽象化することにする．

まず $pdd(2)$ における `foo()` の処理の様子を見てみよう．$pdd(2)$ は，まず入力ポインタが `foo()` の最初の “`f`” を指していることを確認して，文字列 `foo` をそのまま出力ストリームに加えている．さらに，`foo` の直後の `()` の翻訳処理を行い，この `()` の直後の位置に入力ポインタを移して終了する．この `foo` および `()` の処理は $pdd(2)$ の中だけで連続的に処理されるので，`foo` のような変数の名前に相当する文字列や，その直後につなげられるかも知れない，`()` と `[···]` によって作られる文字列がどのようなものであるのかは，今回の検証方針から見れば，あまり重要でないと言える．そこで，`foo()` のような部分は `n` の一文字で表すことにする．

次に前節の，$pdd(1)$ における `(*foo())()[3]` の処理の様子を見てみよう．$pdd(1)$ は，まず入力ポインタが `(*foo())()[3]` の最初の “`(`” を指していることを確認し，入力ポインタの指す先を “`(`” の次の “`*`” に移して $pd(2)$ を実行する．$pd(2)$ の実行によって `(*foo())()[3]` の中の`*foo()` の翻訳処理がなされ，入力ポインタは`*foo()` の直後の “`)`” を指しているはずである．そこで $pdd(1)$ は作業をそのまま続行して `(*foo())` の直後の `()[3]` の翻訳処理を行い，`()[3]` の直後に入力ポインタを移して終了する．

このとき，(*foo()) の直後の ()[3] の翻訳処理は同一プロセス内で連続的に行われるので，()[3] のような () と [···] で作られるこの部分がどのような列をなしているのかは，やはり重要でないと言える．そこで，このような部分は “0 以上の () と [···] による文字列” として b の一文字で表すことにする．

次に先の入力文字列の例**((*foo())()[3])[2] の中の**の部分に注目する．この部分は 1 つのプロセス $pd(0)$ において，連続的に処理されると考えてよい．つまり，**のような “*” が連続する部分について，それがいくつ連続しているかは重要でないと言える．よってこのような部分は a の一文字で表すことにする．

以上の簡略化に従えば，例えば

```
**((*foo())()[3])[2]
```

は

```
a((an)b)b
```

で置き換えられる．

さらに *dcl* および *dirdcl* の定義を観察すると，b が付けられるのは，(変数名の直後を除けば）常に (···) の形をした *dirdcl* の要素の直後であることが分かる．そして，b が 0 以上の長さを持つ列を表していることを考えれば，“)b” は “)” と同一視しても問題ないと言える（つまり，先に述べた検証項目 (I)∼(III) に関して言えば，b については考慮しなくてもよいと言える）．

そこで *dcl* および *dirdcl* を簡略化した dcl_0 および $dirdcl_0$ を以下のように改めて定義する．

$$\boldsymbol{dcl}_0 \ni d ::= dd \mid \mathtt{a}\ dd$$

$$\boldsymbol{dirdcl}_0 \ni dd ::= \mathtt{n} \mid (d)$$

ここで dcl_0 および $dirdcl_0$ を以下のように PVS 上で形式化する．作業をさらに簡単にするために，“n” を 1，“a” を 2，“(” を 3，“)” を 4 でコード化することにより，dcl_0 および $dirdcl_0$ を自然数列の集合として定義する．

ここで注意しなくてはならないことは，PVS には相互再帰的な定義を直接行うための言語は準備されていないことである．実際，PVS の言語に関するリファレンス [4] の第 3.4 節の中に，“PVS では限られた場合での再帰定義が許される．しかし相互再帰（mutual recursion）は認められておらず，関数は全域的でなければならない” という記述がある（19 ページの上から 13 行目）．そこで以下のような関数 P_d を定義することにより，dcl_0 と $dirdcl_0$ を同時に定義する．

```
P_d(str:list[nat]): RECURSIVE [bool, bool] =
IF cons?(str)
    THEN
    IF car(str)=1 AND cdr(str)=null
        THEN (TRUE, TRUE)
    ELSIF car(str)=2 AND proj_2(P_d(cdr(str)))
        THEN (TRUE, FALSE)
    ELSIF car(str)=3
        AND proj_1(P_d(middle(str)))
        AND nth(str,length(str)-1)=4
```

```
        THEN (TRUE, TRUE)
    ELSE (FALSE, FALSE)
    ENDIF
ELSE (FALSE, FALSE)
ENDIF
MEASURE length(str)
```

ここで上記の middle は以下の 2 つの定義によって形式化される自然数列上の関数である．

```
tailcut(str: list[nat]): RECURSIVE list[nat] =
    IF null?(str) OR null?(cdr(str)) THEN null
    ELSE cons(car(str), tailcut(cdr(str)))
    ENDIF
    MEASURE str by <<

middle(str:list[nat]): RECURSIVE list[nat] =
    IF length(str)<3 THEN null
    ELSE cons(car(cdr(str)), middle(cdr(str)))
    ENDIF
    MEASURE length(str)
```

実際のところ，middle は（長さが n の）自然数列 str に対して，str の両側の成分を除いて出来る（長さが $n-2$ の）自然数列を返す関数である．つまり，この関数は (d) の形をした $dirdcl_0$ の要素を d に変形する関数と考えてよい．

P_d は自然数列の集合 list[nat] からブール値の直積集合 [bool,bool] への関数として定義されている．そして，自然数列 str に対して，str が（自然数列の集まりとしての）dcl_0 に含まれるときは P_d(str) の第 1 成分が真になるように，$dirdcl_0$ に含まれるときは P_d(str) の第 2 成分が真になるように定義されている．実際，

```
sdcl: TYPE =
  (LAMBDA (str:list[nat]): proj_1(cls2(str)))

sdd: TYPE =
  (LAMBDA (str:list[nat]): proj_2(cls2(str))
```

によって定義された PVS における型 sdcl および sdd は，自然数列の集まりとしての dcl_0 および $dirdcl_0$ と集合として等しいことが，PVS 上で証明できる．

5.3 関数の形式化

dcl_0 と $dirdcl_0$ の次は pdcl および pdirdcl の形式化を行いたい．第 5.1 で紹介したプログラム pdcl および pdirdcl は翻訳処理された出力値を英文として出力していたが，今回の検証項目 (I)∼(III) に関して言えば，出力がどのような形で表されるかは重要ではない．そこで，dcl_0 の各要素の形に対応して出力処理されるこ

とを抽象的に表現するために（さらにはチュートリアルであることを踏まえて），出力データの集まりを第 3.4 節で紹介した抽象データ型によって形式化する．

実際には，先に定義された dcl_0 および $dirdcl_0$ を抽象データ型によって形式化し直すことになる．そのための準備として以下のような抽象データ型を定義する．

```
ex_adcl: DATATYPE
BEGIN
  E: error?
  N: name?
  A(d:ex_dcl): ast?
  B(d:ex_dcl): back?
END ex_dcl
```

ここで “N” は $dirdcl_0$ における “n” を，“A” は dcl_0 における “a” を，“B” は $dirdcl_0$ における “(” および “)” の組を表す．また “E” は dcl_0 以外の自然数列を意味するために用意したものである．

さらに第 5.2 における関数 P_d と同様な関数 P_ad を以下に定義する

```
P_ad(x:ex_adcl): RECURSIVE [bool,bool] =
  IF name?(x) THEN (TRUE, TRUE)
    ELSIF ast?(x)  AND proj_2(P_ad(d(x)))=TRUE
          THEN (TRUE, FALSE)
    ELSIF back?(x) AND proj_1(P_ad(d(x)))=TRUE
          THEN (TRUE, TRUE)
  ELSE (FALSE, FALSE)
  ENDIF
MEASURE x by <<
```

この P_ad を用いて dcl_0 および $dirdcl_0$ を抽象データ型として表現された型 adcl および adirdcl を以下のように定義する．

```
adcl?: PRED[ex_adcl] =
  LAMBDA (x:ex_adcl): proj_1(P_ad(x))=TRUE

add?: PRED[ex_adcl] =
  LAMBDA (x:ex_adcl): proj_2(P_ad(x))=TRUE

adcl: TYPE = (adcl?)
add: TYPE = (add?)
```

ここで sdcl および sdd のときと同様，adcl および add が dcl_0 および $dirdcl_0$ としての性質を満たすことが証明できる．

さて，入力と出力に関するデータを表現する PVS 上の項が定義できたので，これらを使って pdcl および pdirdcl を PVS 上で関数として定義する．この場合もやはり pdcl と pdirdcl を同時に表現できる関数を以下のように再帰的に定義することによって形式化できる．

```
definition_of_funSDCL[input: list[nat]]: THEORY
BEGIN
IMPORTING ex_adcl
  len: nat = length(input)
  funSDCL(flg: below(2), n:below(len)):
  RECURSIVE [nat, ex_adcl] =
    IF flg=1 THEN
      IF n+1<len AND nth(input,n)=2
      THEN  (proj_1(funSDCL(0,n+1)),
             A(proj_2(funSDCL(0,n+1))))
      ELSE  funSDCL(0,n)
      ENDIF
    ELSE
      IF n+1<len AND nth(input,n)=3
      THEN  LET m=proj_1(funSDCL(1,n+1))
      IN
        IF m<len AND input(m)=4
        THEN (m+1, B(proj_2(funSDCL(1,n+1))))
        ELSE (0,E)
        ENDIF
      ELSIF   nth(input,n)=1
      THEN  (n+1, N)
      ELSE   (0,E)
      ENDIF
    ENDIF
    MEASURE 2*(len-n)+flg
END definition_of_funSDCL
```

ここで `below(len)` は `input` の長さ未満の自然数の集合を意味する型である．

ここで `funSDCL` を用いて `pdcl` および `pdirdcl` が生成するプロセス $pd(i)$ および $pdd(i)$ を，PVS 上で以下のように形式化する．

```
fdcl(str:list[nat], n:nat): [nat, ex_dcl] =
IF null?(str) OR n >= length(str) THEN (0,E)
ELSE funSDCL[str](1,n)
ENDIF

fdirdcl(str:list[nat], n:nat): [nat, ex_dcl] =
IF null?(str) OR n >= length(str) THEN (0,E)
ELSE funSDCL[str](0,n)
ENDIF
```

これらの形式化されたプロセスを用いて，第 5.1 節で述べられた最終的な検証項目 (III)（の前半）を形式化すると以下のようになる．（ここでは `pdcl` が生成する最初のプロセス $pd(0)$ と `pdcl` を同一視している．）

```
correctness_left: THEOREM
  FORALL (str:sdcl): dcl?(proj_2(fdcl(str,0)))
```

5.4 証明の概要

この節において先の定理 correctness_left に対する証明の概要について述べる．

上記の定理を実際に証明するためには，これまでに形式化した概念の他に，第 5.1 節で述べた $T_{pd}(i)$ および $T_{pdd}(i)$ や，$L_{pd}(i)$ および $L_{pdd}(i)$ を計算するための概念を PVS 上で定義する必要がある．つまり，検証項目 (III) を証明するためには他の検証項目を含め $T_{pd}(i)$ や $L_{pd}(i)$ などに関する幾つかの補題を証明しなくてはならない．

例えば，検証項目 (II) はプロセス $pd(i)$ および $pdd(i)$ が最終的にポインタの先を $T_{pd}(i)$ および $T_{pdd}(i)$ に移すという補題を証明することによって直接確かめられる．この補題を形式化するために，まず，dcl_0 の各項 d に対して d の中の n の直後の位置を示す関数 Np:[sdcl->nat] を以下に定義する．(各 d が n を唯一つ含むことは，d の長さに関する帰納法によって PVS 上で簡単に証明できる．)

```
Np(str:sdcl): RECURSIVE nat =
  IF car(str)=1
  THEN 0
  ELSIF car(str)=2 THEN Np(cdr(str))+1
  ELSE Np(middle(str))+1
  ENDIF
  MEASURE length(str)
```

さらに，これを用いて，第 5.1 節で定義された $T_{pd}(i)$ および $T_{pdd}(i)$ を計算するための関数 Tp を以下に定義する．

```
Tp(str:sdcl, n:below(Np(str)+1)):
RECURSIVE nat =
  IF null?(str) OR n=0 THEN length(str)
  ELSE
    IF   car(str)=2 THEN Tp(cdr(str),n-1)+1
    ELSIF  car(str)=3 THEN Tp(middle(str),n-1)+1
    ELSE  n+1
    ENDIF
  ENDIF
  MEASURE length(str)
```

Tp は T_{pd} あるいは T_{pdd} を PVS 上で定義したものではなく，証明（計算）の結果，T_{pd} や T_{pdd} がとる値を表すものであることに注意されたい（このことは，L_{pd}，L_{pdd} および後で定義される Np についても同様である）．実際のところ，T_{pd} とは $pd(i)$ が定めるポインタの指す位置，つまり，

```
proj_1(fdcl(str,i))
```

であり，T_{pdd} は $pdd(i)$ が定めるポインタの指す位置，つまり，

```
proj_1(fdirdcl(str,i))
```

である．

（他の部分についても同じことだが）Np や Tp を使って表される T_{pd} や T_{pdd} に関する補題を示すために，Np や Tp そのものに関するいくつかの補題を PVS 上で証明しなくてはならない．しかし，それらに関する説明は省略する．

さて，これらの関数を用いて検証項目 (II) に対応する 2 つの補題を以下のように形式化する．

```
le_fdcl_Tp: LEMMA
  FORALL (str:sdcl, k:below(Np(str)+1)):
    proj_1(fdcl(str,k))=Tp(str,k)

le_fdirdcl_Tp: LEMMA
  FORALL (str:sdd, k:below(Np(str)+1)):
    proj_1(fdirdcl(str,k))=Tp(str,k)
```

続けて，検証項目 (I) のような，$pd(i)$ および $pdd(i)$ の出力結果がどのようになるかを示す補題を形式化しよう．そのためにまず，第 5.1 節で述べた $L_{pd}(i)$ および $L_{pdd}(i)$ が結果的にとる値（自然数列）を表す関数 subo を以下に定義する．

```
subo(str:sdcl, n:nat): RECURSIVE sdcl =
  IF n=0 THEN str
  ELSIF (sdcl?(subo(str,n-1))
         AND car(subo(str,n-1))=2)
               THEN cdr(subo(str,n-1))
  ELSIF (sdd?(subo(str,n-1))
         AND car(subo(str,n-1))=3)
               THEN middle(subo(str,n-1))
  ELSE  subo(str,n-1)
ENDIF
MEASURE n
```

これを用いて，検証項目 (I) に対応する一連の補題を以下に形式化する．

```
le_fdcl_subo: LEMMA
  FORALL (str:sdcl, k:below(Np(str)+1)):
    proj_2(fdcl(str,k+1))
      =proj_2(fdcl(subo(str,1), k))

le_fdirdcl_subo: LEMMA
  FORALL (str:sdcl, k:below(Np(str)+1)):
    sdd?(subo(str,k)) IMPLIES
    proj_2(fdirdcl(str,k+1))
      =proj_2(fdirdcl(subo(str,1), k))

le_main_left: LEMMA
```

```
FORALL (str:sdcl, flg:below(2)):
  ( flg=0 IMPLIES
      dcl?(proj_2(funSDCL[str](flg,0))) )
  AND
  ( (sdd?(str) AND flg=1) IMPLIES
      dirdcl?(proj_2(funSDCL[str](flg,0))) )
```

大まかな流れとしては，T_{pd} および T_{pdd} に関する補題 le_fdcl_Tp および le_fdirdcl_Tp を証明し，この補題から補題 le_fdcl_subo および le_fdirdcl_subo を証明する．
そして，これら 4 つの補題を用いて，主補題となる le_main_left を 2*length(str)+flg に関する帰納法で証明する．そして，この補題から直接定理 correctness_left が得られる．

5.5 この節のまとめ

今回は 1 組のプログラム pdcl および pdirdcl に注目して，それらのプログラムの入出力データおよびプログラムそのものを適当に抽象化した上で，PVS 上で形式化した．そして，それらを用いて第 5.1 の検証項目 (I)∼(III)（の前半）を形式化し，PVS 上で証明を行った．

本来ならば，抽象化・形式化の作業と並行して，その抽象化と形式化が妥当であることを数学的に（より厳密には PVS 上で）証明する必要があるだろうが，ここではそのような作業は割愛した．

また，今回の検証対象はプログラム（コード）だったので，検証を行う上でプログラムのアルゴリズムを考える必要があった．しかしながら，演繹的な検証の特徴を考えると，もっと抽象的なレベルで検証作業を行った方が（あるいはもっと抽象的な検証対象を選んだ方が），効率の良い検証が出来たかも知れない．

今回は入力データが数列の形をしていて，しかも 1 番前の成分から順次中身を参照するというアルゴリズム上の制約にこだわったために，作業が予想以上に長引いた．実際，入力データも adcl および adirdcl によって形式化したものを扱うとすると，今回述べた作業よりもはるかに作業が簡単になる．しかしその作業で得られる検証結果は，pdcl の検証の趣旨から考えると不満なものに感じられたので，今回の方針ではそうしなかった．

実際の検証作業に入る前に，検証対象をどのように形式化またはモデル化するか，あるいは演繹的な検証作業にとって効率の良い対象かどうかを吟味することは，検証作業そのものと同様に重要だと言える．

6 本論のまとめ

本論の最初の 2 節において，[2]∼[4] の内容の一部を要約して，PVS とその作業の大まかな流れについて説明した．さらに，[4]∼[6] に基づき，第 3 節において PVS における型の概念について述べ，第 4 節において PVS の証明の構成，特に，基本的なコマンドについて説明した．そして，最後の節において，*dcl* を翻訳するプログラム pdcl および pdirdcl を取り上げ，PVS を用いてこれらのプログラムの検証を行った．また，その作業のための抽象化についても議論した．

PVS をより詳しく知りたい読者は，[2]，[4] および [6] を参照されたい．[2] は PVS の基本を説明したチュートリアルであり，[4] は PVS の言語に関するマニュアル，[6] は証明コマンドに関するマニュアルである．また [5] では PVS における証明の概念が分かりやすくまとめられている．

謝辞 本論は文部科学省科学技術振興調整費（若手任期付支援）の支援のもとで作製されました．また，本論の初期の版について，多くのアドバイスを与えて下さった査読者の方々に感謝いたします．

References

[1] SRI Computer Science Laboratory における PVS の web サイト（PVS の開発 · 配布元）, http://pvs.csl.sri.com/.

[2] Crow, J., Owre, S., Rushby, J. M., Shankar, N. and Srivas, M.：A tutorial Introduction to PVS, in *Workshop on Industrial-Strength Formal Specification Techniques*, 1995, http://pvs.csl.sri.com/.

[3] Owre, S. and Shankar, N.:The PVS Prelude Library (CSL Technical Report SRI-CSL-03-01), 2003, http://pvs.csl.sri.com/.

[4] Owre, S., Shankar, N., Rushby, J. M. and Stringer-Calvert, D. W. J.：PVS Language Reference, 2001, http://pvs.csl.sri.com/.

[5] Peled, D.：*Software Reliability Methods* (Texts in Computer Science), Springer Verlag, 2001.

[6] Shankar, N., Owre, S., Rushby, J. M. and Stringer-Calvert, D. W. J.：PVS Prover Guide, 2001, http://pvs.csl.sri.com/.

[7] Kernighan, B. W. and Ritchie, D. M.：*The C programming Language 2nd Edition*, Prentice Hall P T R, （訳本: 石田晴久訳, プログラミング言語 C ANSI 規格準拠, 共立出版）.

A レコード型を用いた理論の例

この節では，Kripke 構造に関する基本的な性質を形式化した PVS ファイルを紹介する．

まず，Kripke 構造とある種の抽象化に関する定義および命題を，通常の方法で以下に記述する．

定義 1 AP を原子命題の集合とする．

定義 2 AP 上の Kripke 構造は以下の 4 つのデータの組である．

(i) 集合 X (状態集合);

(ii) Y $(\subset X)$ (初期状態集合);

(iii) $R \subset X \times X$ (状態遷移);

(iv) $L : X \to 2^{AP}$

以下，Kripke 構造はすべて AP 上のものとする．

定義 3 集合 X，X 上の関数 h および Kripke 構造 $M = (X, Y, R, L)$ が与えられたとして，Kripke 構造 $M_\alpha = (X_\alpha, Y_\alpha, R_\alpha, L_\alpha)$ を以下に定義する．

(i) $X_\alpha := h(X),\ Y_\alpha := h(Y)$；

(ii) 任意の $a, b \in X_\alpha$ について

$$R_\alpha(a,b) \Leftrightarrow_{df}$$
$$\exists\, x, y \in X\ (h(x) = a \wedge h(y) = b \wedge R(x,y))；$$

(iii) 任意の $a \in X_\alpha$ について，

$$L_\alpha(a) := \bigcup \{L(x) : h(x) = a\}.$$

このような M_α を h による M の抽象化と呼ぶ．

定義 4 (1) Kripke 構造 $M_1 = (X_1, Y_1, R_1, L_1)$ および $M_2 = (X_2, Y_2, R_2, L_2)$ について，$H \subset X_1 \times X_2$ が M_1 と M_2 の模倣関係であるとは，H が以下の条件を満たすことを言う．

(i) 任意の $s_1 \in Y_1$ について，ある $s_2 \in Y_2$ が存在して $H(s_1, s_2)$ を満たす．

(ii) 任意の $(s_1, s_2) \in H$ について次が成り立つ．

- $L_1(s_1) \subset L_2(s_2)$；
- $\forall\, t_1 \in X_1\ (\ R_1(s_1, t_1) \Rightarrow$
  $\exists\, t_2 \in X_2\ (\ R_2(s_2, t_2) \wedge H(t_1, t_2)\ )\ )$.

(2) Kripke 構造 M_1 および M_2 について，M_1 と M_2 の模倣関係が存在するとき，M_2 は M_1 を模倣するという．

定理 任意の Kripke 構造 M および M 上の関数 h について，h による M の抽象化は M を模倣する．

上記の定義 1∼4 および定理を PVS 上で形式化すると，以下の図 A にあるような PVS ファイル（`abs_sim.pvs`）が得られる．

`abs_sim.pvs` の始めの部分で，集合 `X` について，`X` を状態集合とする（`AP` についての）Kripke 構造の全体を型 `KMTYP[X]` として定義する．さらに，各 Kripke 構造 `KM∈KMTYP[X]` および `X` 上の写像 `h` に対して，`KM` の `h` による抽象化 `abs_map: KMTYP[X]→KMTYP[Im_hX]` を定義する．ここで `Im_hX` は `h` による `X` の像である．

さらに，2 つの Kripke 構造 `KM1∈KMTYP[X]` と `KM2∈KMTYP[Im_hX]` 間の模倣性（similarity）を定義し，最後に定理として，任意の `KM1∈KMTYP[X]` について，`abs_map(KM1)` が `KM1` を模倣することが記述されている．

この PVS ファイルでは，Kripke 構造全体をレコード型

```
[# iset: setof[X], rel: setof[[X,X]],
             ass: [X -> setof[AP]] #]
```

によって定義し，2 番目の理論 “abstract_KM” の最後の部分,

```
  abs_map: [ KMTYP[X] -> KMTYP[Im_hX]] =
    LAMBDA (KM1: KMTYP[X]):
      (# iset:= abs_ISET(KM1),
         rel:= abs_REL(KM1),
         ass:= abs_ASS(KM1) #)
```

にある “(# iset:= abs_ISET(KM1), ... #)” によって, レコード型KMTYP[Im_hX] を持つ項が定義されていることに注意されたい.

```
Atomic_prop: THEORY
BEGIN
  AP: TYPE
END Atomic_prop

Kripke_model[X: TYPE]: THEORY
BEGIN
IMPORTING Atomic_prop
  KMTYP: TYPE =
    [# iset: setof[X], rel: setof[[X,X]],
                    ass: [X -> setof[AP]] #]
  KM: VAR KMTYP
  nonempty_iset: AXIOM
    FORALL (K: KMTYP): EXISTS (x:X): K`iset(x)
  totality_of_rel: AXIOM
    FORALL (K: KMTYP): FORALL (x:X):
                  EXISTS (y:X): K`rel(x,y)
END Kripke_model

abstract_KM[X:TYPE, Y:TYPE, h:[X->Y]]: THEORY
BEGIN
IMPORTING Kripke_model
  Im_hX : TYPE = {y:Y | EXISTS (x:X): h(x)=y}
  KM: VAR KMTYP[X]
  KM_a: VAR KMTYP[Im_hX]
  abs_ISET(KM): setof[Im_hX]
    = {y: Im_hX|
      EXISTS (x:X): h(x)=y AND KM`iset(x)}
  abs_REL(KM): PRED[[Im_hX, Im_hX]] =
    LAMBDA (y_0: Im_hX, y_1: Im_hX):
    EXISTS (x_0: X, x_1: X):
      h(x_0) = y_0 AND h(x_1) = y_1
                AND KM`rel(x_0, x_1)
  abs_ASS(KM): [Im_hX -> setof[AP]] =
    LAMBDA (y: Im_hX): {p: AP| EXISTS (x:X):
            member(p, KM`ass(x)) AND y=h(x)}
```

```
  abs_map: [ KMTYP[X] -> KMTYP[Im_hX]] =
    LAMBDA (KM1: KMTYP[X]):
      (# iset:= abs_ISET(KM1),
         rel:= abs_REL(KM1),
         ass:= abs_ASS(KM1) #)
END abstract_KM

abs_sim [X:TYPE, Y:TYPE, h:[X->Y]]: THEORY
BEGIN
IMPORTING abstract_KM
  Im_hX : TYPE = {y:Y | EXISTS (x:X): h(x)=y}
  KM_1: VAR KMTYP[X]
  KM_2: VAR KMTYP[Im_hX]
  simulation?(KM_1, KM_2):PRED[PRED[[X,Im_hX]]]
    = LAMBDA(B:PRED[[X,Im_hX]]) :
    (FORALL (x:X): KM_1`iset(x) IMPLIES
     EXISTS (y:Im_hX): KM_2`iset(y) AND B(x,y))
    AND
     FORALL (x:X,y:Im_hX): B(x,y) IMPLIES
     (
     subset?(KM_1`ass(x), KM_2`ass(y))
     AND FORALL (x_1:X): KM_1`rel(x,x_1)
          IMPLIES EXISTS (y_1:Im_hX):
           (KM_2`rel(y,y_1) AND B(x_1,y_1))
     )
  sim_KM?(KM_1, KM_2): bool =
    EXISTS (R: PRED[[X, Im_hX]]):
              simulation?(KM_1, KM_2)(R)
  abs_sim: THEOREM
    FORALL (KM1: KMTYP[X]):
              sim_KM?(KM1, abs_map(KM1))
END abs_sim
```

(図 A: abs_sim.pvs)

上記の定理は PVS 上で実際に証明されているが，その説明は省略する．

B left_cancellation の証明

以下のデータは group.pvs における命題 left_cancellation を PVS 上で証明した仮定を記述したものである．(ページの節約のため，非本質的な一部の記述を省略している．)

```
  |-------
{1}   FORALL (a, b, c: G): a o b = a o c IMPLIES b = c

Rule? (skolem!)
Skolemizing,
```

```
  |-------
{1}   a!1 o b!1 = a!1 o c!1 IMPLIES b!1 = c!1

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,

{-1}  a!1 o b!1 = a!1 o c!1
  |-------
{1}   b!1 = c!1

Rule? (lemma "unit")
Applying unit

{-1}  FORALL (a: G): e o a = a AND a o e = a
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "b!1")
Instantiating the top quantifier in -1 with the terms:
 b!1,

{-1}  e o b!1 = b!1 AND b!1 o e = b!1
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,

{-1}  e o b!1 = b!1
{-2}  b!1 o e = b!1
[-3]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "inverse")
Applying inverse

{-1}  FORALL (a: G): inv(a) o a = e AND a o inv(a) = e
[-2]  e o b!1 = b!1
[-3]  b!1 o e = b!1
[-4]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "a!1")
Instantiating the top quantifier in -1 with the terms:
 a!1,

{-1}  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-2]  e o b!1 = b!1
[-3]  b!1 o e = b!1
[-4]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,

{-1}  inv(a!1) o a!1 = e
{-2}  a!1 o inv(a!1) = e
[-3]  e o b!1 = b!1
[-4]  b!1 o e = b!1
[-5]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -1 (-1 -3) RL)
Replacing using formula -1,

[-1]  inv(a!1) o a!1 = e
[-2]  a!1 o inv(a!1) = e
{-3}  inv(a!1) o a!1 o b!1 = b!1
[-4]  b!1 o e = b!1
[-5]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "associativity")
Applying associativity

{-1}  FORALL (a, b, c: G): a o (b o c) = (a o b) o c
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
[-4]  inv(a!1) o a!1 o b!1 = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1
```

```
Rule? (inst -1 "inv(a!1)" "a!1" "b!1")
Instantiating the top quantifier in -1 with the terms:
 inv(a!1), a!1, b!1,

{-1}  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
[-4]  inv(a!1) o a!1 o b!1 = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -1 (-1 -4) RL)
Replacing using formula -1,

[-1]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
{-4}  inv(a!1) o (a!1 o b!1) = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -6 (-6 -4))
Replacing using formula -6,

[-1]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-2]  inv(a!1) o a!1 = e
[-3]  a!1 o inv(a!1) = e
{-4}  inv(a!1) o (a!1 o c!1) = b!1
[-5]  b!1 o e = b!1
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "associativity")
Applying associativity

{-1}  FORALL (a, b, c: G): a o (b o c) = (a o b) o c
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  inv(a!1) o a!1 = e
[-4]  a!1 o inv(a!1) = e
[-5]  inv(a!1) o (a!1 o c!1) = b!1
[-6]  b!1 o e = b!1
[-7]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "inv(a!1)" "a!1" "c!1")
Instantiating the top quantifier in -1 with the terms:
 inv(a!1), a!1, c!1,

{-1}  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  inv(a!1) o a!1 = e
[-4]  a!1 o inv(a!1) = e
[-5]  inv(a!1) o (a!1 o c!1) = b!1
[-6]  b!1 o e = b!1
[-7]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -1 (-1 -5))
Replacing using formula -1,

[-1]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  inv(a!1) o a!1 = e
[-4]  a!1 o inv(a!1) = e
{-5}  (inv(a!1) o a!1) o c!1 = b!1
[-6]  b!1 o e = b!1
[-7]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -3 (-3 -5))
Replacing using formula -3,

[-1]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  inv(a!1) o a!1 = e
[-4]  a!1 o inv(a!1) = e
{-5}  e o c!1 = b!1
[-6]  b!1 o e = b!1
[-7]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1
```

```
Rule? (lemma "unit")
Applying unit

{-1}  FORALL (a: G): e o a = a AND a o e = a
[-2]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-3]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-4]  inv(a!1) o a!1 = e
[-5]  a!1 o inv(a!1) = e
[-6]  e o c!1 = b!1
[-7]  b!1 o e = b!1
[-8]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "c!1")
Instantiating the top quantifier in -1 with the terms:
 c!1,

{-1}  e o c!1 = c!1 AND c!1 o e = c!1
[-2]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-3]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-4]  inv(a!1) o a!1 = e
[-5]  a!1 o inv(a!1) = e
[-6]  e o c!1 = b!1
[-7]  b!1 o e = b!1
[-8]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,

{-1}  e o c!1 = c!1
{-2}  c!1 o e = c!1
[-3]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-4]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-5]  inv(a!1) o a!1 = e
[-6]  a!1 o inv(a!1) = e
[-7]  e o c!1 = b!1
[-8]  b!1 o e = b!1
[-9]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (replace -1 (-1 -7))
Replacing using formula -1,

[-1]  e o c!1 = c!1
[-2]  c!1 o e = c!1
[-3]  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-4]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-5]  inv(a!1) o a!1 = e
[-6]  a!1 o inv(a!1) = e
{-7}  c!1 = b!1
[-8]  b!1 o e = b!1
[-9]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
Q.E.D.
```

C　grind を用いた left_cancellation の証明

以下のデータは先と同じ命題 `left_cancellation` をやはり PVS 上で証明したものである．ただし，先の証明の中の flatten や replace などの単純な論理式の変形に関する一連のコマンドは一切使わず，代わりに grind を用いている．

grind は flatten，split，skolem!，inst?，replace などの単純な操作を場当たり的に，出来る限り繰り返すという操作である．grind は単純な作業を一括して行うので，上手に用いると非常に便利である．しかし，不適当に grind を用いると，PVS が行う証明作業（各コマンド毎に PVS がシークエントを処理する作業）が無限ループに陥る可能性がある．

前回はコマンドを 22 回用いたことに対して，今回用いたコマンドは `skosimp*`（1 回），補題の呼び出しおよび補題への項の割り当て（それぞれ 5 回ずつ），`grind`

（1 回）となっている．今回は証明に必要な一連の補題とそれらの補題をどのように使うのかを予め指定しておいたので，残りの単純な操作は一度の grind で一括して行うことが出来た．

```
left_cancellation :

  |-------
{1}   FORALL (a, b, c: G): a o b = a o c IMPLIES b = c

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,

{-1}  a!1 o b!1 = a!1 o c!1
  |-------
{1}   b!1 = c!1

Rule? (lemma "inverse")
Applying inverse

{-1}  FORALL (a: G): inv(a) o a = e AND a o inv(a) = e
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "a!1")
Instantiating the top quantifier in -1 with the terms:
 a!1,

{-1}  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-2]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "unit")
Applying unit

{-1}  FORALL (a: G): e o a = a AND a o e = a
[-2]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-3]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "b!1")
Instantiating the top quantifier in -1 with the terms:
 b!1,

{-1}  e o b!1 = b!1 AND b!1 o e = b!1
[-2]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-3]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "unit")
Applying unit

{-1}  FORALL (a: G): e o a = a AND a o e = a
[-2]  e o b!1 = b!1 AND b!1 o e = b!1
[-3]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-4]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "c!1")
Instantiating the top quantifier in -1 with the terms:
 c!1,

{-1}  e o c!1 = c!1 AND c!1 o e = c!1
[-2]  e o b!1 = b!1 AND b!1 o e = b!1
[-3]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-4]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "associativity")
Applying associativity

{-1}  FORALL (a, b, c: G): a o (b o c) = (a o b) o c
[-2]  e o c!1 = c!1 AND c!1 o e = c!1
[-3]  e o b!1 = b!1 AND b!1 o e = b!1
[-4]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-5]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "inv(a!1)" "a!1" "b!1")
Instantiating the top quantifier in -1 with the terms:
 inv(a!1), a!1, b!1,
```

```
{-1}  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-2]  e o c!1 = c!1 AND c!1 o e = c!1
[-3]  e o b!1 = b!1 AND b!1 o e = b!1
[-4]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-5]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (lemma "associativity")
Applying associativity

{-1}  FORALL (a, b, c: G): a o (b o c) = (a o b) o c
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  e o c!1 = c!1 AND c!1 o e = c!1
[-4]  e o b!1 = b!1 AND b!1 o e = b!1
[-5]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (inst -1 "inv(a!1)" "a!1" "c!1")
Instantiating the top quantifier in -1 with the terms:
 inv(a!1), a!1, c!1,

{-1}  inv(a!1) o (a!1 o c!1) = (inv(a!1) o a!1) o c!1
[-2]  inv(a!1) o (a!1 o b!1) = (inv(a!1) o a!1) o b!1
[-3]  e o c!1 = c!1 AND c!1 o e = c!1
[-4]  e o b!1 = b!1 AND b!1 o e = b!1
[-5]  inv(a!1) o a!1 = e AND a!1 o inv(a!1) = e
[-6]  a!1 o b!1 = a!1 o c!1
  |-------
[1]   b!1 = c!1

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.



Run time  = 8.85 secs.
Real time = 186.29 secs.


nil
pvs(20):
```

PVS の紹介
(算譜科学研究速報)
発行日：2005 年 4 月 6 日
編集・発行：独立行政法人産業技術総合研究所関西センター尼崎事業所
システム検証研究センター
同連絡先：〒661-0974 兵庫県尼崎市若王寺 3-11-46
e-mail：informatics-inquiry@m.aist.go.jp

A short tutorial on PVS
(Programming Science Technical Report)
April 6, 2005
Research Center for Verification and Semantics (CVS)
AIST Kansai, Amagasaki Site
National Institute of Advanced Industrial Science and Technology (AIST)
3-11-46 Nakouji, Amagasaki, Hyogo, 661-0974, Japan
e-mail: informatics- inquiry@m.aist.go.jp