Proof Development using PVS

Parvati Chandrasekhar Iyer
parvati.iyer@gmail.com

Abstract

PVS stands for Prototype Verification System. Developed by SRI, this system integrates a specification language with support tools and a theorem prover.

This tutorial is not intended to be an exhaustive introduction to PVS. New users are advised to refer to the manuals and tutorials cited in the references for basic details regarding the language syntax and prover commands. This tutorial is targeted at those users who have tried developing preliminary proofs using PVS, and who need some worked out examples to gain more confidence using the theorem prover.

1 Example 1 : Direct Proof

In this section, we deal with a simple property of prime numbers, presented as an exercise problem in [3].

Prove that the sum of two prime numbers, each greater than 2, is not a prime number.

The first step to proving a property is to formally specify it. We first check the prelude file [M-x view-prelude-file] to find whether any specification of prime numbers is available. While a prime number specification is not available, we see that a predicate *divides* is available.

Using this, we proceed to specify our theory as :

```
primes : THEORY

BEGIN

n,d,p,q : VAR nat

prime?(n) : bool =
FORALL d : NOT d = 1 AND d < n => NOT divides(d,n)

primality_testing : LEMMA prime?(7)

sum_primes : LEMMA prime?(p) AND prime?(q)
AND p>2 and q > 2 => NOT prime?(p+q)
```

```
END primes
```

primality_testing is a putative theorem we use to perform a preliminary check on the correctness of our specification.

On starting the proof checker, we have :

```
|----
{1}   prime?(7)
```

PVS-proofs are based on **Sequent Calculus**, where we begin a proof with the sequent to be proved, and simplify this sequent at each step till it is trivially true.

A sequent can be generally represented as

```
[-1]  C
[-2]  D
  |-------
{1}   A
{2}   B
```

The formulae above the dashed line are called the *antecedents* (C and D), while those below the line (A and B) are called the *consequents* or *succeedents.* A sequent is true if the conjunction of the antecedents implies the disjunction of the consequents. Since $A \rightarrow B \equiv \neg A \vee B$, the simplest route to proving a sequent to be true would be to prove that any antecedent is false, or any consequent is true, or that any antecedent is equal to its consequent.

Having imbibed this bare-bones version of sequent calculus, let us continue with the proof of *primality_testing*.

```
primality_testing :

  |-------
{1}   prime?(7)
```

The start sequent consists of a single consequent. To simplify this sequent, we obviously need to tell the proof checker what the predicate ‘prime’ means. We use the *expand* command[TAB e] for this.

```
Rule? (expand "prime?")
Expanding the definition of prime?,
this simplifies to:
```

```
primality_testing :

  |-------
{1}   FORALL d: NOT d = 1 AND d < 7 => NOT divides(d, 7)
```

Since we have a universally quantified consequent, we need to apply *Skolemization.*

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
primality_testing :

{-1}  d!1 < 7
{-2}  divides(d!1, 7)
  |-------
{1}   d!1 = 1
```

Looking at this sequent, we can see that we need to prove the formula {-2} to be false, and then we are done.

On expanding the definition of divides, we get

```
Rule? (expand "divides")
Expanding the definition of divides,
this simplifies to:
primality_testing :

[-1]  d!1 < 7
{-2}  EXISTS x: 7 = d!1 * x
  |-------
[1]   d!1 = 1
```

Again, since we have an existentially quantified antecedent, we need to skolemize. This gives us

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
primality_testing :

[-1]  d!1 < 7
```

```
{-2}  7 = d!1 * x!1
  |-------
[1]   d!1 = 1
```

At this step, we need to prove that given that $d!1 \neq 1$ and $d!1 < 7$, $d!1 * x!1 \neq 7$. We proceed to split the current goal into two sub-goals using the *case* construct [TAB c].

```
Rule? (case "d!1=2")
Case splitting on
   d!1 = 2,
this yields  2 subgoals:
primality_testing.1 :

{-1}  d!1 = 2
[-2]  d!1 < 7
[-3]  7 = d!1 * x!1
  |-------
[1]   d!1 = 1
```

Note that the subgoal is now named as *primality_testing*.1 . These numeric suffixes in the subgoal name are useful in tedious or long proofs to help keep track of how much has been proved so far.

We now see that {-2} can easily be proved false with some simplification. The *assert* [TAB a]command provides this simplification. Use commands like *assert* and *flatten* [TAB f] to simplify your sequent at any stage of the proof. They also help to make a sequent more readable.

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of primality_testing.1.
```

The second sub-goal is

```
primality_testing.2 :

[-1]  d!1 < 7
[-2]  7 = d!1 * x!1
  |-------
{1}   d!1 = 2
[2]   d!1 = 1
```

Proceeding in a similar manner, by considering the various cases, we can easily establish that 7 is a prime.

This proof was an exceedingly simple one, and does not demonstrate the capabilities of PVS. I traced this proof merely to have the chance to introduce the amateur user to the simplest and most useful PVS commands.

We now proceed to prove the main goal of this section.

```
sum_primes :

  |-------
{1}   FORALL (p, q: nat):
        prime?(p) AND prime?(q) AND p > 2 AND q > 2 => NOT prime?(p + q)
```

Again we deal with the universally quantified consequent by skolemization.

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
sum_primes :

{-1}  prime?(p!1)
{-2}  prime?(q!1)
{-3}  p!1 > 2
{-4}  q!1 > 2
{-5}  prime?(p!1 + q!1)
  |-------
```

To prove this sequent true, we obviously need to prove one of the antecedents false. I choose {-5}, you could choose any other. Our task is then to prove {-5} to be false, assuming the other formulae to be true. At this step, rather than expand the definition of 'prime?' it might help to think about how you would go about such a proof on paper. Remember that PVS is only a proof-checker,the definitive steps that direct the proof must be provided by you.

One way to go about this proof is to realise that if $p!1$ and $q!1$ are primes and are greater than 2, they must definitely be odd. If they are both odd, their sum is even, and cannot be a prime.That's the gist of our proof. Now to mechanise this proof,we need to check how to represent odd and even numbers. We also need to use in our proof the fact that the sum of two odd numbers is even, and that a prime number greater than 2 is definitely odd. A search reveals that the concepts of odd and even numbers have been defined in the PVS prelude theories. We now save and exit the current proof, and proceed to define the auxiliary lemmas we need.

not_prime_even : LEMMA $even?(n)$ ***AND*** $n > 2 \rightarrow$ ***NOT*** $prime?(n)$

sum_odd_numbers : LEMMA *odd?*(p) ***AND*** *odd?*$(q) \rightarrow$ *even?*$(p+q)$

Although the proof of *sum_primes* is not complex, this technique of recognising which lemmas would help simplify the current proof is highly useful when dealing with more complicated proofs. Also, as in any programming discipline, such a strategy provides modularity and proof reuse. It is especially useful when several subgoals of a given goal seem to need a common sequence of steps for simplification.

We could choose to prove these lemmas right now; or if we are confident of having specified them correctly, we can proceed with the main proof. I'll proceed with the main proof — the proof of these lemmas can be undertaken as an exercise. Note that all the PVS specification and proof files for all the examples and exercises dealt with in this tutorial are available in [4].

On resuming the proof of *sum_primes*, we apply the lemma 'not_prime_even'.

```
Rule? (lemma "not_prime_even")
Applying not_prime_even
this simplifies to:
sum_primes :

{-1}  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-2]  prime?(p!1)
[-3]  prime?(q!1)
[-4]  p!1 > 2
[-5]  q!1 > 2
[-6]  prime?(p!1 + q!1)
  |-------
```

The lemma command introduces a new antecedent, which may help to prove the sequent true. Since we need to prove that both p!1 and q!1 are even, we need two copies of the statement -1, for this we use the *copy* command.

```
Rule? (copy -1)
Copying formula number: -1,
this simplifies to:
sum_primes :

{-1}  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-2]  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
```

A universally qualified antecedent needs to be instantiated with a suitable value to prove the sequent true. Consequently, we instantiate -1 with $p!1$ and -2 with $q!1$ as follows.

```
Rule? (inst -1 "p!1")
Instantiating the top quantifier in -1 with the terms:
 p!1,
this simplifies to:
sum_primes :

{-1}  even?(p!1) AND p!1 > 2 => NOT prime?(p!1)
[-2]  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_primes :

[-1]  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-2]  prime?(p!1)
[-3]  prime?(q!1)
[-4]  p!1 > 2
[-5]  q!1 > 2
[-6]  prime?(p!1 + q!1)
  |-------
{1}   even?(p!1)

Rule? (inst -1 "q!1")
Instantiating the top quantifier in -1 with the terms:
 q!1,
this simplifies to:
sum_primes :

{-1}  even?(q!1) AND q!1 > 2 => NOT prime?(q!1)
[-2]  prime?(p!1)
[-3]  prime?(q!1)
[-4]  p!1 > 2
[-5]  q!1 > 2
[-6]  prime?(p!1 + q!1)
  |-------
```

```
[1]   even?(p!1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_primes :

[-1]  prime?(p!1)
[-2]  prime?(q!1)
[-3]  p!1 > 2
[-4]  q!1 > 2
[-5]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
{2}   even?(q!1)
```

The additional lemma we introduced talks of the sum of two odd numbers. We also need to inform PVS that a number that is not even is odd. We do this using the 'even_or_odd' lemma in the naturalnumbers prelude theory.

```
Rule? (lemma "naturalnumbers.even_or_odd")
Applying naturalnumbers.even_or_odd
this simplifies to:
sum_primes :

{-1}  FORALL (x: int): even?(x) IFF NOT odd?(x)
[-2]  prime?(p!1)
[-3]  prime?(q!1)
[-4]  p!1 > 2
[-5]  q!1 > 2
[-6]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)
```

The rest of the proof is a cakewalk, and the intermediate steps are pretty self-explanatory.

```
Rule? (copy -1)
Copying formula number: -1,
this simplifies to:
sum_primes :

{-1}  FORALL (x: int): even?(x) IFF NOT odd?(x)
[-2]  FORALL (x: int): even?(x) IFF NOT odd?(x)
[-3]  prime?(p!1)
```

```
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (inst -1 "p!1")
Instantiating the top quantifier in -1 with the terms:
 p!1,
this simplifies to:
sum_primes :

{-1}  even?(p!1) IFF NOT odd?(p!1)
[-2]  FORALL (x: int): even?(x) IFF NOT odd?(x)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_primes :

{-1}  odd?(p!1)
[-2]  FORALL (x: int): even?(x) IFF NOT odd?(x)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (inst -2 "q!1")
Instantiating the top quantifier in -2 with the terms:
 q!1,
this simplifies to:
sum_primes :
```

```
[-1]  odd?(p!1)
{-2}  even?(q!1) IFF NOT odd?(q!1)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_primes :

[-1]  odd?(p!1)
{-2}  odd?(q!1)
[-3]  prime?(p!1)
[-4]  prime?(q!1)
[-5]  p!1 > 2
[-6]  q!1 > 2
[-7]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (lemma "sum_odd_numbers")
Applying sum_odd_numbers
this simplifies to:
sum_primes :

{-1}  FORALL (p, q: nat): odd?(p) AND odd?(q) => even?(p + q)
[-2]  odd?(p!1)
[-3]  odd?(q!1)
[-4]  prime?(p!1)
[-5]  prime?(q!1)
[-6]  p!1 > 2
[-7]  q!1 > 2
[-8]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (inst -1 "p!1" "q!1")
```

```
Instantiating the top quantifier in -1 with the terms:
 p!1, q!1,
this simplifies to:
sum_primes :

{-1}  odd?(p!1) AND odd?(q!1) => even?(p!1 + q!1)
[-2]  odd?(p!1)
[-3]  odd?(q!1)
[-4]  prime?(p!1)
[-5]  prime?(q!1)
[-6]  p!1 > 2
[-7]  q!1 > 2
[-8]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
sum_primes :

{-1}  even?(p!1 + q!1)
[-2]  odd?(p!1)
[-3]  odd?(q!1)
[-4]  prime?(p!1)
[-5]  prime?(q!1)
[-6]  p!1 > 2
[-7]  q!1 > 2
[-8]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (lemma "not_prime_even")
Applying not_prime_even
this simplifies to:
sum_primes :

{-1}  FORALL (n: nat): even?(n) AND n > 2 => NOT prime?(n)
[-2]  even?(p!1 + q!1)
[-3]  odd?(p!1)
[-4]  odd?(q!1)
[-5]  prime?(p!1)
[-6]  prime?(q!1)
[-7]  p!1 > 2
```

```
[-8]  q!1 > 2
[-9]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

Rule? (inst -1 "p!1+q!1")
Instantiating the top quantifier in -1 with the terms:
 p!1+q!1,
this simplifies to:
sum_primes :

{-1}  even?(p!1 + q!1) AND p!1 + q!1 > 2 => NOT prime?(p!1 + q!1)
[-2]  even?(p!1 + q!1)
[-3]  odd?(p!1)
[-4]  odd?(q!1)
[-5]  prime?(p!1)
[-6]  prime?(q!1)
[-7]  p!1 > 2
[-8]  q!1 > 2
[-9]  prime?(p!1 + q!1)
  |-------
[1]   even?(p!1)
[2]   even?(q!1)

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

Such a direct approach to a proof is probably the most natural and intuitive proof strategy.

2 Example 2 : Proofs using Induction

In this section, we prove the correctness of the formula for the sum of terms in a geometric progression.

For all natural numbers $a > 1$**,** $1+a+a^2+........a^(n-1) = (a^n-1)/(a-1)$

This property is specified as follows

```
induction : THEORY

BEGIN

a,n : VAR posnat

%sum of first n terms of a geometric progression
```

```
geom(a,n) : RECURSIVE posnat =
  (IF n=1
THEN 1
ELSE a^(n-1)+geom(a,n-1)
ENDIF)
MEASURE n

geom_sum : THEOREM a/=1 => geom(a,n)  = (a^n-1)/(a-1)

END induction
```

Note the use of a recursive construct to specify the sum of a series. The measure simply indicates a natural number that decreases each time the function is invoked recursively. For more details on how to specify recursive functions, see [5, 6].

Our goal in this section is to prove 'geom_sum'.

```
geom_sum :

  |-------
{1}   FORALL (a, n: posnat): a /= 1 => geom(a, n) = (a ^ n - 1) / (a  1)
```

As mentioned in [3], if a statement to be proved is of the form $\forall\ n >= n_0 : P(n)$, where n_0 is some fixed integer then mathematical induction could be applied to develop the corresponding proof.

We thus proceed to develop an induction-based proof.

```
Rule? (induct "n")
Inducting on n on formula 1,
this yields  4 subgoals:
geom_sum.1 :

  |-------
{1}   n!1 > 0
{2}   FORALL (a): a /= 1 => geom(a, n!1) = (a ^ n!1 - 1) / (a  1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of geom_sum.1.

geom_sum.2 :

  |-------
{1}   0 > 0 IMPLIES
```

```
       (FORALL (a): a /= 1 => geom(a, 0) = (a ^ 0 - 1) / (a - 1))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of geom_sum.2.

geom_sum.3 :

  |-------
{1}   FORALL j:
        (j > 0 IMPLIES
          (FORALL (a): a /= 1 => geom(a, j) = (a ^ j - 1) / (a - 1)))
         IMPLIES
         j + 1 > 0 IMPLIES
          (FORALL (a):
             a /= 1 => geom(a, j + 1) = (a ^ (j + 1) - 1) / (a  1))
```

We now have to tackle the third subgoal. An assert makes this more readable.

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3 :

  |-------
{1}   FORALL j:
        (j > 0 IMPLIES
          (FORALL (a): a /= 1 => geom(a, j) = (a ^ j - 1) / (a - 1)))
         IMPLIES
         (FORALL (a):
            a /= 1 => geom(a, 1 + j) = (a ^ (1 + j) - 1) / (a  1))
```

This step is similar to the start of earlier proof 'sum_primes'. We again skolemize to eliminate the universal quantification.

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
geom_sum.3 :

{-1}  j!1 > 0 IMPLIES
       (FORALL (a): a /= 1 => geom(a, j!1) = (a ^ j!1 - 1) / (a - 1))
  |-------
{1}   a!1 = 1
{2}   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1 - 1)
```

```
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3 :

[-1]  j!1 > 0 IMPLIES
       (FORALL (a): a /= 1 => geom(a, j!1) = (a ^ j!1 - 1) / (a - 1))
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1  1)
```

We are now at the induction step in a typical proof using mathematical induction. If the result holds true for $j!1$ terms of the geometric progression, we need to show that its true for $1 + j!1$ terms.

To simplify the formula -1, we can use the ground command. It invokes propositional simplification and simplifies the resultant cases.

```
Rule? (ground)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
geom_sum.3.1 :

{-1}  FORALL (a): a /= 1 => geom(a, j!1) = (a ^ j!1 - 1) / (a - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1  1)
```

We need to instantiate the quantifier in -1 with a suitable constant, in this case $a!1$

```
Rule? (inst -1 "a!1")
Instantiating the top quantifier in -1 with the terms:
 a!1,
this simplifies to:
geom_sum.3.1 :

{-1}  a!1 /= 1 => geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3.1 :
```

```
{-1}  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1  1)
```

In the following steps, we realise that we will need to use the laws of indices. These have been stated and proved in the prelude theory *exponentiation*.

```
Rule? (lemma "exponentiation.expt_plus")
Applying exponentiation.expt_plus
this simplifies to:
geom_sum.3.1 :

{-1}  FORALL (i, j: int, n0x: nzreal): n0x ^ (i + j) = n0x ^ i * n0x ^ j
[-2]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1 - 1)

Rule? (inst -1 "1" "j!1" "a!1")
Instantiating the top quantifier in -1 with the terms:
 1, j!1, a!1,
this simplifies to:
geom_sum.3.1 :

{-1}  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-2]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-2]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1 - 1)
```

We now use the formula -1 to rewrite formula [2] using the replace command.

```
Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-2]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
{2}   geom(a!1, 1 + j!1) = (a!1 ^ 1 * a!1 ^ j!1 - 1) / (a!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-2]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ 1 * a!1 ^ j!1 - 1) / (a!1  1)
```

We now need to invoke another lemma from the exponentiation prelude theory.

```
Rule? (lemma "expt_x1")
Applying expt_x1
this simplifies to:
geom_sum.3.1 :

{-1}  FORALL (x: real): x ^ 1 = x
[-2]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ 1 * a!1 ^ j!1 - 1) / (a!1 - 1)

Rule? (inst -1 "a!1")
Instantiating the top quantifier in -1 with the terms:
 a!1,
this simplifies to:
geom_sum.3.1 :

{-1}  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
```

```
[2]   geom(a!1, 1 + j!1) = (a!1 ^ 1 * a!1 ^ j!1 - 1) / (a!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 ^ 1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
[2]   geom(a!1, 1 + j!1) = (a!1 ^ 1 * a!1 ^ j!1 - 1) / (a!1 - 1)

Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ 1 = a!1
{-2}  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
{2}   geom(a!1, 1 + j!1) = (a!1 * a!1 ^ j!1 - 1) / (a!1  1)
```

With the formulae now considerably simplified, we need to somehow use the expression for *geom*$(a!1, j!1)$ to prove the sequent true. For this, we expand *geom* only in the formula [2].

```
Rule? (expand "geom" 2)
Expanding the definition of geom,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
{2}   (IF j!1 = 0 THEN 1 ELSE geom(a!1, j!1) + a!1 ^ j!1 ENDIF) =
        (a!1 * a!1 ^ j!1 - 1) / (a!1  1)
```

The *lift-if* command helps to lift the branching structure in 2 to the top level so that propositional simplification can be applied. This gives us

```
Rule? (lift-if 2)
Lifting IF-conditions to the top level,
this simplifies to:
geom_sum.3.1 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   a!1 = 1
{2}   IF j!1 = 0 THEN 1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
      ELSE geom(a!1, j!1) + a!1 ^ j!1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
      ENDIF
```

Based on the branches in 2, at this stage it would be useful to consider two cases : when $j!1 = 0$ and when $j!1 \neq 0$. This is achieved using the *split* command as follows

```
Rule? (split 2)
Splitting conjunctions,
this yields  2 subgoals:
geom_sum.3.1.1 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
{1}   j!1 = 0 IMPLIES 1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
[2]   a!1 = 1

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
geom_sum.3.1.1 :

{-1}  j!1 = 0
[-2]  a!1 ^ 1 = a!1
[-3]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-4]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
{1}   1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
[2]   a!1 = 1

Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
```

```
geom_sum.3.1.1 :

[-1]  j!1 = 0
[-2]  a!1 ^ 1 = a!1
{-3}  a!1 ^ (1 + 0) = a!1 * a!1 ^ 0
{-4}  geom(a!1, 0) = (a!1 ^ 0 - 1) / (a!1 - 1)
  |-------
{1}   1 = (a!1 * a!1 ^ 0 - 1) / (a!1 - 1)
[2]   a!1 = 1

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of geom_sum.3.1.1.
```

Now we deal with the second subgoal generated by the *split* command.

```
geom_sum.3.1.2 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
{1}   NOT j!1 = 0 IMPLIES
       geom(a!1, j!1) + a!1 ^ j!1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
[2]   a!1 = 1
```

The command flatten performs disjunctive simplification leading to a list of formulae with no disjuncts. We use it here to simplify 1.

```
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
geom_sum.3.1.2 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
{1}   j!1 = 0
{2}   geom(a!1, j!1) + a!1 ^ j!1 = (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
[3]   a!1 = 1
```

We can now use the value of geom(a!1,j!1) provided in -3 to prove the sequent to be true.

```
Rule? (replace -3)
```

```
Replacing using formula -3,
this simplifies to:
geom_sum.3.1.2 :

[-1]  a!1 ^ 1 = a!1
[-2]  a!1 ^ (1 + j!1) = a!1 * a!1 ^ j!1
[-3]  geom(a!1, j!1) = (a!1 ^ j!1 - 1) / (a!1 - 1)
  |-------
[1]   j!1 = 0
{2}   (a!1 ^ j!1 - 1) / (a!1 - 1) + a!1 ^ j!1 =
       (a!1 * a!1 ^ j!1 - 1) / (a!1 - 1)
[3]   a!1 = 1

Rule? (ground)
Applying propositional simplification and decision procedures,

This completes the proof of geom_sum.3.1.2.

This completes the proof of geom_sum.3.1.
```

The name of the next goal indicates that we now face the second subgoal created when we used the command *ground*.

```
geom_sum.3.2 :

  |-------
{1}   j!1 > 0
[2]   a!1 = 1
[3]   geom(a!1, 1 + j!1) = (a!1 ^ (1 + j!1) - 1) / (a!1  1)

Rule? (expand "^")
Expanding the definition of ^,
this simplifies to:
geom_sum.3.2 :

  |-------
[1]   j!1 > 0
[2]   a!1 = 1
{3}   1 = (expt(a!1, 1 + j!1) - 1) / (a!1 - 1)

Rule? (expand "expt")
Expanding the definition of expt,
this simplifies to:
geom_sum.3.2 :

  |-------
```

```
[1]   j!1 > 0
[2]   a!1 = 1
{3}   1 = (a!1 * expt(a!1, j!1) - 1) / (a!1 - 1)

Rule? (expand "expt")
Expanding the definition of expt,
this simplifies to:
geom_sum.3.2 :

  |-------
[1]   j!1 > 0
[2]   a!1 = 1
{3}   1 = (a!1 - 1) / (a!1 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of geom_sum.3.2.


This completes the proof of geom_sum.3.

geom_sum.4 :

  |-------
{1}   FORALL (n: nat):
        n > 0 IMPLIES (FORALL (a): a /= 1 IMPLIES (a - 1) /= 0)
{2}   FORALL (a): a /= 1 => geom(a, n!1) = (a ^ n!1 - 1) / (a - 1)
```

In this case, we see that the formula 1 can easily be shown to be true.

```
Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
geom_sum.4 :

{-1}  n!2 > 0
{-2}  (a!1 - 1) = 0
  |-------
{1}   a!1 = 1
{2}   a!2 = 1
{3}   geom(a!2, n!1) = (a!2 ^ n!1 - 1) / (a!2 - 1)

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of geom_sum.4.

Q.E.D.
```

We thus have proved the result to calculate the sum of the first n terms of a geometric progression where the first term is 1.

The following two exercises are aimed at giving you more practice in developing proofs using induction.

Exercise 1 : Prove that the sum of the first n ***natural numbers is*** $n * (n + 1)/2$

Exercise 2 : Prove that 3 ***divides*** $(n^3 - n)$ ***for every positive integer*** n

3 Example 3 : Proofs using some common data structures

Its highly unlikely that you will need to use PVS to prove simple properties or established results like those in the previous sections. You would need it to prove other more advanced theories, with possibly advanced data structures.

In this section, we'll prove a property of a deterministic finite automaton using PVS. This section aims at giving you some insights into how to write more complex specifications, and how proofs based on these specifications are essentially similar to those we've already seen.

Let us prove the following property, provided as an exercise problem in [2].

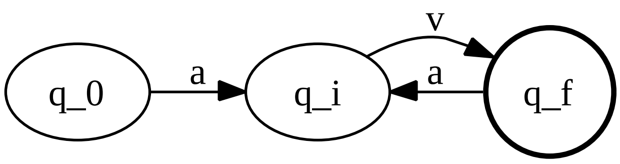

Figure 1: Example 3

Let $A = (Q, \sigma, \delta, q_0, q_f)$ ***be a deterministic finite automaton, where*** Q ***is the set of states*** σ ***is the input alphabet*** δ ***is the transition function*** q_0 ***is the start state*** q_f ***is a final state contained in the singleton set of final states***

Suppose that for all a ***in sigma, we have*** $\delta(q_0, a) = delta(q_f, a)$

Show that if x ***is a non-empty string accepted by*** A***, then*** $x.x$ ***(***x ***concatenated with*** x***) is also accepted by*** A***.***

Let us first understand what we are required to prove.

Consider an automaton as shown in the Figure 1. Q_0 is the start state, q_i is an intermediate state, and q_f is the final state.

As mentioned in the problem statement, both q_0 and q_f have a transition to the same state(here q_i) on the application of any input symbol a.

Let v be a string that causes a transition from q_i (through zero or more intermediate states) to q_f. Let $w = av$. Then, given that w is accepted by this automaton A, we need to prove that ww is also accepted by this automaton.

Now that the problem statement is clear, let us proceed to the specify a deterministic finite automaton.

```
auto [state , input : TYPE+] : THEORY

BEGIN

dfa : TYPE = [# states : finite_set[state], alphabet : finite_set[input], start : state

dfa1 : VAR dfa
rho : VAR {a: finseq[input]| a`length >0}
path : VAR {a : finseq[state] | a`length > 1}
i : VAR nat
set_states : VAR finite_set[state]
set_inputs : VAR finite_set[input]
al : VAR input
st : VAR state

%define the notion of acceptance of a string by an NFA

first_node(path) : state =
path`seq(0)

last_node(path) : state =
path`seq(path`length-1)

belongs(path,set_states) : bool =
(FORALL i : i< path`length => member(path`seq(i),set_states))

belongs(rho, set_inputs) : bool =
(FORALL i : i< rho`length => member(rho`seq(i),set_inputs))

accepts(rho,dfa1) : bool =
(IF NOT belongs(rho,dfa1`alphabet)
THEN FALSE
ELSE
(EXISTS path :
belongs(path,dfa1`states) AND
first_node(path)= dfa1`start AND
member(last_node(path),dfa1`final) AND
path`length = rho`length + 1 AND
(FORALL i : i< rho`length => path`seq(i+1)= dfa1`delta(path`seq(i),rho`seq(i))))
ENDIF)

meaningful_dfa(dfa1) : bool =
subset?(dfa1`final,dfa1`states) AND member(dfa1`start,dfa1`states)

property_dfa : THEOREM meaningful_dfa(dfa1) AND dfa1`final = singleton(st) AND (FORALL a

END auto
```

Here, we introduce *Parametrized Theories.* Since we do not know how states and the input alphabet are to be represented in the implementation, and since their representation does not affect the proof we provide these types as parameters to the theory auto.

The five-tuple representation of the DFA in the problem statement is captured using the *record* type. We have assigned the fields in the record suitable names. Note that finite sets are defined in the *finite_sets* prelude theory. Also, the transition function *delta* has been defined as a function that takes as input a state and an input symbol, and returns a state.

We then declare the variables we will need in our specification. For simplicity, here I have assumed that any string (or sequence of input symbols) is non-empty. I have also assumed that any sequence of states contains at least two states. The interested reader is welcome to extend the specification to include the cases I have ignored.

The six functions in the theory help define the notion of acceptance of a string by a DFA. Note that this is obviously not the only way to define acceptance. In fact, [2] have provided an inductive definition for the acceptance of a string by a DFA. I found the definition above easier and more amenable to proof development, but that's a personal opinion. Note however, that the proof is highly influenced by the specification, and certain specifications are more useful than others for certain proofs.

The function *accepts* in the specification basically states that a string ρ is accepted by a dfa $dfa1$ if every symbol in ρ is a member of the input alphabet of $dfa1$ AND There exists a sequence of states ,namely *path* such that every state in *path* is a member of $dfa1$'s set of states the first state of *path* is the start state of $dfa1$ the last state of *path* is a final state *path*'s length exceeds *rho*'s length by 1 For every integer $i < \rho$'s length : A transition from the i^{th} state of path on the i^{th} symbol in ρ leads to the $(i+1)^{th}$ state of *path*.

The theorem *property_dfa* states formally the property we seek to prove.

Now that we are familiar with the specification, the proof is pretty straightforward. Although much lengthier than the earlier proofs, it doesn't make use of any new commands as such. Consequently, I have avoided including the proof in this document. Interested readers can find the worked out proof in the references.

As a closing note, I include a few generic tips that didn't find mention in the earlier examples :

- Keep the big picture in mind. Use the [M-x xpr] command to draw the proof tree as you work on the proof if it suits you. Else, you the numeric suffixes on the proof goals to keep track of how far you've managed to proceed through the proof. Solve the simpler subgoals first, working on the tougher ones later. It helps to make the problem more manageable and still keeps your spirits high.
- Control the branching factor at each step. If a command leads to more than 5–6 subgoals, its probably the wrong tactic and there is a simpler solution. A premature use of *ground* or *grind* leads to a flood of subgoals, avoid trying to work your way through these.
- The *hide* command is very useful in improving the readability of a sequent, and in helping the user get an idea regarding how to proceed with the proofs. Note that *hide* does not delete the formulae, it simply makes the specified formulae invisible and removes them temporarily from consideration. Use this command when you are confident that certain formulae would not contribute towards proving the sequent true.Note that any hidden formulae can be redisplayed later using the *reveal* command. The *undo* command helps to retrace your steps in a proof. The proving task is much simplified by the use of these commands.

4 Conclusion

This tutorial has been prepared with the earnest wish that you have an easier time learning to use PVS. I am but a novice user of PVS, and there is a high likelihood that this document contains technical errors. Moreover, most of the generic statements I have made about how to write proofs using PVS are highly based on my personal experiences and opinions. As I enrich my set of experiences, I hope to refine my understanding and opinions, and provide more useful hints and guidelines in this document. Nevertheless, if you come across any errors in this document, I would be grateful to be made aware of them. Likewise, any suggestions for improving the text, providing better examples and exercises, and illustrating new proof techniques would be highly appreciated.

5 Acknolwedgments

I would like to thank Dr.Uday Khedker, Associate Professor, Dept. of Computer Science and Engineering, IIT Bombay for encouraging me to write this tutorial. Some of my work on PVS was funded by the IIT Bombay Summer Internship Scheme.

References

[1] Aho, A.V., Sethi R., and Ullman J.D., 1986. *Compilers — Principles, Techniques and Tools.* Addison-Wesley.

[2] Hopcroft, J.E., Motwani, R. and Ullman, J.D., 2000. *Introduction to Automata Theory, Languages and Computation.* Pearson Addison-Wesley, Boston, MA, USA.

[3] Kolman, B., Busby R.C., Ross S.C., 1999. *Discrete Mathematical Structures — 4th Edition.* Prentice Hall.

[4] Iyer, P.C., 2005. *Proof Development Using PVS.* http://profile.iiita.ac.in/ipchandrashekar_02/publications/

[5] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas, 1995. *A Tutorial Introduction to PVS.*Workshop on Industrial-Strength Formal Specification Techniques. Boca Raton, Florida.

[6] Ricky W. Butler, 1993. *An Elementary Tutorial on Formal Specification and Verification Using PVS.*. NASA Technical Memorandum 108991. NASA Langley Research Center, Hampton, VA 23681.