

# Recursion, Induction, and Iteration

César A. Muñoz

NASA Langley Research Center  
Cesar.A.Munoz@nasa.gov

PVS Tutorial 2017



# Outline

Recursive Definitions

Induction Proofs

Induction-Free Induction

Recursive Judgements

Iterations

Inductive Definitions

Mutual Recursion and Higher-Order Recursion

## Recursive Definitions in PVS

Suppose we want to define a function to sum the first  $n$  natural numbers:

$$\text{sum}(n) = \sum_{i=0}^n i.$$

In PVS:

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

## Recursive Definitions in PVS

Suppose we want to define a function to sum the first  $n$  natural numbers:

$$\text{sum}(n) = \sum_{i=0}^n i.$$

In PVS:

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

# Functions in PVS are Total

Two Type Correctness Conditions(TCCs):

- ▶ The argument for the recursive call is a natural number.

```
% Subtype TCC generated for n - 1
```

```
  % expected type  nat
```

```
sum_TCC1: OBLIGATION FORALL (n: nat):
```

```
  NOT n = 0 IMPLIES n - 1 >= 0;
```

- ▶ The recursion terminates.

```
% Termination TCC generated for sum(n - 1)
```

```
sum_TCC2: OBLIGATION FORALL (n: nat):
```

```
  NOT n = 0 IMPLIES n - 1 < n;
```

# Functions in PVS are Total

Two Type Correctness Conditions(TCCs):

- ▶ The argument for the recursive call is a natural number.

```
% Subtype TCC generated for n - 1
```

```
  % expected type  nat
```

```
sum_TCC1: OBLIGATION FORALL (n: nat):
```

```
  NOT n = 0 IMPLIES n - 1 >= 0;
```

- ▶ The recursion terminates.

```
% Termination TCC generated for sum(n - 1)
```

```
sum_TCC2: OBLIGATION FORALL (n: nat):
```

```
  NOT n = 0 IMPLIES n - 1 < n;
```

## A Simple Property of Sum

We would like to prove the following closed form solution to sum:

$$\sum_{i=0}^n i = \frac{n(n+1)}{2}.$$

In PVS:

```
closed_form: THEOREM
  sum(n) = (n * (n + 1)) / 2
```

# Induction Proofs

`(induct/$ var &optional (fnum 1) name) :`

Selects an induction scheme according to the type of VAR in FNUM and uses formula FNUM to formulate an induction predicate, then simplifies yielding base and induction cases. The induction scheme can be explicitly supplied as the optional NAME argument.



## Induction Schemes from the Prelude

```
% Weak induction on naturals.
```

```
nat_induction: LEMMA
```

```
  (p(0) AND (FORALL j: p(j) IMPLIES p(j+1)))  
    IMPLIES (FORALL i: p(i))
```

```
% Strong induction on naturals.
```

```
NAT_induction: LEMMA
```

```
  (FORALL j: (FORALL k: k < j IMPLIES p(k)) IMPLIES p(j))  
    IMPLIES (FORALL i: p(i))
```

# Proof by Induction

closed\_form :

|-----

{1}  FORALL (n: nat): sum(n) = (n \* (n + 1)) / 2

Rule? (induct "n")

Inducting on n on formula 1,  
this yields 2 subgoals:

# Proof by Induction

closed\_form :

|-----

{1}  FORALL (n: nat): sum(n) = (n \* (n + 1)) / 2

Rule? (induct "n")

Inducting on n on formula 1,  
this yields 2 subgoals:

## Base Case

closed\_form.1 :

|-----  
{1} sum(0) = (0 \* (0 + 1)) / 2

Rule? (grind)

Rewriting with sum

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of closed\_form.1.

## Base Case

closed\_form.1 :

|-----  
{1} sum(0) = (0 \* (0 + 1)) / 2

Rule? (grind)

Rewriting with sum

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of closed\_form.1.

closed\_form.2 :

```
|-----  
{1}  FORALL j:  
      sum(j) = (j * (j + 1)) / 2 IMPLIES  
      sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
```

Rule? (skeep)

Skolemizing with the names of the bound variables,  
this simplifies to:

closed\_form.2 :

```
{-1}  sum(j) = (j * (j + 1)) / 2  
      |-----  
{1}  sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
```

closed\_form.2 :

|-----  
{1}    FORALL j:  
        sum(j) = (j \* (j + 1)) / 2 IMPLIES  
        sum(j + 1) = ((j + 1) \* (j + 1 + 1)) / 2

Rule? (skeep)

Skolemizing with the names of the bound variables,  
this simplifies to:

closed\_form.2 :

{-1}   sum(j) = (j \* (j + 1)) / 2  
      |-----  
{1}    sum(j + 1) = ((j + 1) \* (j + 1 + 1)) / 2

$$\{-1\} \quad \text{sum}(j) = (j * (j + 1)) / 2$$

|-----

$$\{1\} \quad \text{sum}(j + 1) = ((j + 1) * (j + 1 + 1)) / 2$$

Rule? (expand "sum" +)

Expanding the definition of sum,

this simplifies to:

closed\_form.2 :

$$\{-1\} \quad \text{sum}(j) = (j * (j + 1)) / 2$$

|-----

$$\{1\} \quad 1 + \text{sum}(j) + j = (2 + j + (j * j + 2 * j)) / 2$$



$$\{-1\} \quad \text{sum}(j) = (j * (j + 1)) / 2$$

|-----

$$\{1\} \quad \text{sum}(j + 1) = ((j + 1) * (j + 1 + 1)) / 2$$

Rule? (expand "sum" +)

Expanding the definition of sum,

this simplifies to:

closed\_form.2 :

$$\{-1\} \quad \text{sum}(j) = (j * (j + 1)) / 2$$

|-----

$$\{1\} \quad 1 + \text{sum}(j) + j = (2 + j + (j * j + 2 * j)) / 2$$

$$[-1] \quad \text{sum}(j) = (j * (j + 1)) / 2$$

$$\begin{array}{l} |----- \\ \{1\} \quad 1 + \text{sum}(j) + j = (2 + j + (j * j + 2 * j)) / 2 \end{array}$$

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed\_form.2.

Q.E.D.

$$[-1] \quad \text{sum}(j) = (j * (j + 1)) / 2$$

|-----

$$\{1\} \quad 1 + \text{sum}(j) + j = (2 + j + (j * j + 2 * j)) / 2$$

Rule? (assert)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed\_form.2.

Q.E.D.

# Automated Simple Induction Proofs

|-----  
{1}    `FORALL (n: nat): sum(n) = (n * (n + 1)) / 2`

Rule? `(induct-and-simplify "n")`

Rewriting with `sum`

Rewriting with `sum`

By induction on `n`, and by repeatedly rewriting and simplifying,

Q.E.D.

# Automated Simple Induction Proofs

|-----  
{1}    `FORALL (n: nat): sum(n) = (n * (n + 1)) / 2`

Rule? `(induct-and-simplify "n")`

Rewriting with `sum`

Rewriting with `sum`

By induction on `n`, and by repeatedly rewriting and simplifying,

Q.E.D.

## Limitations of automation

Consider the  $n$ th factorial:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n(n-1)!, & \text{otherwise.} \end{cases}$$

In the NASA PVS theory `ints@factorial`:

```
factorial(n : nat): RECURSIVE posnat =  
  IF n = 0 THEN 1 ELSE n * factorial(n - 1) ENDIF  
MEASURE n
```

# A Simple Property of Factorial

$$\forall n : n! \geq n$$

In PVS:

```
factorial_ge : LEMMA
  FORALL (n:nat): factorial(n) >= n
```

## A Series of Unfortunate Events ...

Rule? (induct-and-simplify "n")

Rewriting with factorial

Rewriting with factorial

Rewriting with factorial

Warning: Rewriting depth = 50; Rewriting with factorial

Warning: Rewriting depth = 100; Rewriting with factorial

...



Whenever the theorem prover falls into an infinite loop, the **Emacs command C-c C-c** will force PVS to break into Lisp. The Lisp command **(restore)** will return to the PVS state prior to the last proof command.

...

```
Error: Received signal number 2 (Interrupt)
```

```
  [condition type: interrupt-signal]
```

```
Restart actions (select using :continue):
```

```
  0: continue computation
```

```
  1: Return to Top Level (an "abort" restart).
```

```
  2: Abort entirely from this (lisp) process.
```

```
[1c] pvs(137): (restore)
```

```
factorial_ge :
```

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

```
{1}  FORALL (n: nat): factorial(n) >= n
```

```
Rule?
```

## Factorial in C

Consider a common implementation of the  $n$ -th factorial in an imperative programming language:

```
/* Pre: n >= 0 */  
  
int a = 1;  
for (int i=0;i < n;i++) {  
    /* Inv: a = i! */  
    a = a*(i+1);  
}  
  
/* Post: a = n! */
```

## Factorial in C

Consider a common implementation of the  $n$ -th factorial in an imperative programming language:

```
/* Pre: n >= 0 */  
  
int a = 1;  
for (int i=0;i < n;i++) {  
    /* Inv: a = i! */  
    a = a*(i+1);  
}  
  
/* Post: a = n! */
```

## In PVS ...

```
fact_it(n:nat,i:upto(n),a:posnat) : RECURSIVE posnat =  
  IF  i = n THEN a  
  ELSE fact_it(n,i+1,a*(i+1))  
  ENDIF  
MEASURE n-i
```

```
fact_it_correctness : THEOREM  
  fact_it(n,0,1) = factorial(n)
```

## Proving fact\_it\_correctness

```
|-----  
{1}  FORALL (n: nat): fact_it(n, 0, 1) = factorial(n)
```

Rule? (**induct-and-simplify "n"**)

this simplifies to:

```
fact_it_correctness :
```

```
{-1}  fact_it(j!1, 0, 1) = factorial(j!1)
```

```
|-----  
{1}  fact_it(1 + j!1, 1, 1) =  
      factorial(j!1) + factorial(j!1) * j!1
```

The proof by (explicit) induction requires an inductive proof of an auxiliary lemma.

## Proving fact\_it\_correctness

|-----  
{1}   FORALL (n: nat): fact\_it(n, 0, 1) = factorial(n)

Rule? (~~induct-and-simplify "n"~~)

this simplifies to:

fact\_it\_correctness :

{-1}   fact\_it(j!1, 0, 1) = factorial(j!1)

|-----  
{1}   fact\_it(1 + j!1, 1, 1) =  
      factorial(j!1) + factorial(j!1) \* j!1

The proof by (explicit) induction requires an inductive proof of an auxiliary lemma.

# Induction-Free Induction By Predicate Subtyping

```
fact_it(n:nat,i:upto(n),(a:posnat|a=factorial(i))) :  
  RECURSIVE {b:posnat | b=factorial(n)} =  
  IF    i = n THEN a  
  ELSE fact_it(n,i+1,a*(i+1))  
  ENDIF  
MEASURE n-i
```

```
n : VAR nat
```

```
fact_it_correctness : LEMMA  
  fact_it(n,0,1) = factorial(n)  
%|- fact_t_correctness : PROOF (skip) (assert) QED
```

# Induction-Free Induction By Predicate Subtyping

```
fact_it(n:nat,i:upto(n),(a:posnat|a=factorial(i))) :  
  RECURSIVE {b:posnat | b=factorial(n)} =  
  IF    i = n THEN a  
  ELSE fact_it(n,i+1,a*(i+1))  
  ENDIF  
MEASURE n-i
```

```
n : VAR nat
```

```
fact_it_correctness : LEMMA  
  fact_it(n,0,1) = factorial(n)  
%|- fact_t_correctness : PROOF (skip) (assert) QED
```



# Induction-Free Induction By Predicate Subtyping

```
fact_it(n:nat,i:upto(n),(a:posnat|a=factorial(i))) :  
  RECURSIVE {b:posnat | b=factorial(n)} =  
  IF   i = n THEN a  
  ELSE fact_it(n,i+1,a*(i+1))  
  ENDIF  
MEASURE n-i
```

```
n : VAR nat
```

```
fact_it_correctness : LEMMA  
  fact_it(n,0,1) = factorial(n)  
%|- fact_t_correctness : PROOF (skip) (assert) QED
```

# Induction-Free Induction By Predicate Subtyping

```
fact_it(n:nat,i:upto(n),(a:posnat|a=factorial(i))) :  
  RECURSIVE {b:posnat | b=factorial(n)} =  
  IF    i = n THEN a  
  ELSE fact_it(n,i+1,a*(i+1))  
  ENDIF  
MEASURE n-i
```

```
n : VAR nat
```

```
fact_it_correctness : LEMMA  
  fact_it(n,0,1) = factorial(n)  
%|- fact_t_correctness : PROOF (skip) (assert) QED
```

# There is No Free Lunch

```
fact_it_TCC4 :  
  |-----  
{1}  FORALL (n: nat, i: upto(n),  
            (a: nat | a = factorial(i))):  
      NOT i = n IMPLIES a * (i + 1) = factorial(1 + i)
```

Rule? (skeep :preds? t)

```
fact_it_TCC4 :  
{-1}  n >= 0  
{-2}  i <= n  
{-3}  a = factorial(i)  
  |-----  
{1}   i = n  
{2}   a * (i + 1) = factorial(1 + i)
```

# There is No Free Lunch

```
fact_it_TCC4 :  
  |-----  
{1}  FORALL (n: nat, i: upto(n),  
            (a: nat | a = factorial(i))):  
      NOT i = n IMPLIES a * (i + 1) = factorial(1 + i)
```

Rule? (skeep :preds? t)

```
fact_it_TCC4 :  
{-1}  n >= 0  
{-2}  i <= n  
{-3}  a = factorial(i)  
  |-----  
{1}   i = n  
{2}   a * (i + 1) = factorial(1 + i)
```

Rule? (expand "factorial" 2)

fact\_it\_TCC4 :

[-1]  $n \geq 0$

[-2]  $i \leq n$

[-3]  $a = \text{factorial}(i)$

|-----

[1]  $i = n$

{2}  $a * i + a = \text{factorial}(i) + \text{factorial}(i) * i$

Rule? (assert)

Q.E.D.

Rule? (expand "factorial" 2)

fact\_it\_TCC4 :

[-1] n >= 0

[-2] i <= n

[-3] a = factorial(i)

|-----

[1] i = n

{2} a \* i + a = factorial(i) + factorial(i) \* i

Rule? (assert)

Q.E.D.

Rule? (expand "factorial" 2)

fact\_it\_TCC4 :

[-1] n >= 0

[-2] i <= n

[-3] a = factorial(i)

|-----

[1] i = n

{2} a \* i + a = factorial(i) + factorial(i) \* i

Rule? (assert)

Q.E.D.

## You Can Also Pay at the Exit

```
fact_it2(n:nat,i:upto(n),a:posnat) : RECURSIVE
  {b:posnat | b = a*factorial(n)/factorial(i)} =
  IF   i = n THEN a
  ELSE fact_it2(n,i+1,a*(i+1))
  ENDIF
MEASURE n-i
```

```
fact_it2_correctness : LEMMA
  fact_it2(n,0,1) = factorial(n)
```



## You Can Also Pay at the Exit

```
fact_it2(n:nat,i:upto(n),a:posnat) : RECURSIVE
  {b:posnat | b = a*factorial(n)/factorial(i)} =
  IF   i = n THEN a
  ELSE fact_it2(n,i+1,a*(i+1))
  ENDIF
MEASURE n-i
```

```
fact_it2_correctness : LEMMA
  fact_it2(n,0,1) = factorial(n)
```

```
|-----  
{1}  FORALL (n: nat): fact_it2(n, 0, 1) = factorial(n)
```

Rule? (skip)

```
|-----  
{1}  fact_it2(n, 0, 1) = factorial(n)
```

Rule? (typepred "fact\_it2(n,0,1)")

```
{-1}  fact_it2(n, 0, 1) > 0  
{-2}  fact_it2(n, 0, 1) = 1 * factorial(n) / factorial(0)  
|-----  
[1]  fact_it2(n, 0, 1) = factorial(n)
```

```
|-----  
{1}  FORALL (n: nat): fact_it2(n, 0, 1) = factorial(n)
```

Rule? (skip)

```
|-----  
{1}  fact_it2(n, 0, 1) = factorial(n)
```

Rule? (typepred "fact\_it2(n,0,1)")

```
{-1}  fact_it2(n, 0, 1) > 0  
{-2}  fact_it2(n, 0, 1) = 1 * factorial(n) / factorial(0)  
|-----  
[1]  fact_it2(n, 0, 1) = factorial(n)
```

```
|-----  
{1}  FORALL (n: nat): fact_it2(n, 0, 1) = factorial(n)
```

Rule? (skeep)

```
|-----  
{1}  fact_it2(n, 0, 1) = factorial(n)
```

Rule? (typepred "fact\_it2(n,0,1)")

```
{-1}  fact_it2(n, 0, 1) > 0  
{-2}  fact_it2(n, 0, 1) = 1 * factorial(n) / factorial(0)  
|-----  
[1]  fact_it2(n, 0, 1) = factorial(n)
```

Rule? (expand "factorial" -2 2)

[-1]  $\text{fact\_it2}(n, 0, 1) > 0$

{-2}  $\text{fact\_it2}(n, 0, 1) = 1 * \text{factorial}(n) / 1$

|-----

[1]  $\text{fact\_it2}(n, 0, 1) = \text{factorial}(n)$

Rule? (assert)

Q.E.D.

Rule? (expand "factorial" -2 2)

[-1] fact\_it2(n, 0, 1) > 0

{-2} fact\_it2(n, 0, 1) = 1 \* factorial(n) / 1

|-----

[1] fact\_it2(n, 0, 1) = factorial(n)

Rule? (assert)

Q.E.D.

## But The Price is Higher

```
fact_it2_TCC5: OBLIGATION
  FORALL (n: nat, i: upto(n),
    v:
      [d1: z: [n: nat, upto(n), posnat] |
        z'1 - z'2 < n - i ->
        b: posnat | b = d1'3 * factorial(d1'1) /
          factorial(d1'2)],
    a: posnat):
  NOT i = n IMPLIES
  v(n, i + 1, a * (i + 1)) =
  a * factorial(n) / factorial(i);
```

Rule? (skeep :preds? t)

Skolemizing with the names of the bound variables,  
this simplifies to:

fact\_it2\_TCC5 :

{-1} n >= 0

{-2} i <= n

{-3} a > 0

|-----

{1} i = n

{2}  $v(n, i + 1, a * (i + 1)) = a * \text{factorial}(n) /$   
 $\text{factorial}(i)$



Rule? (skeep :preds? t)

Skolemizing with the names of the bound variables,  
this simplifies to:

fact\_it2\_TCC5 :

{-1} n >= 0

{-2} i <= n

{-3} a > 0

|-----

{1} i = n

{2}  $v(n, i + 1, a * (i + 1)) = a * \text{factorial}(n) / \text{factorial}(i)$

Rule? (name-replace "HI" "v(n, i + 1, a \* (i + 1))")  
Using HI to name and replace  $v(n, i + 1, a * (i + 1))$ ,  
this yields 2 subgoals:  
fact\_it2\_TCC5.1 :

[-1]  $n \geq 0$

[-2]  $i \leq n$

[-3]  $a > 0$

|-----

[1]  $i = n$

{2}  $HI = a * \text{factorial}(n) / \text{factorial}(i)$

Rule? (name-replace "HI" "v(n, i + 1, a \* (i + 1))")  
Using HI to name and replace v(n, i + 1, a \* (i + 1)),  
this yields 2 subgoals:  
fact\_it2\_TCC5.1 :

[-1] n >= 0

[-2] i <= n

[-3] a > 0

|-----

[1] i = n

{2} HI = a \* factorial(n) / factorial(i)

Rule? (typepred "HI")

Adding type constraints for HI,  
this simplifies to:

fact\_it2\_TCC5.1 :

{-1} HI > 0

{-2}  $HI = \frac{\text{factorial}(n) * a + \text{factorial}(n) * a * i}{\text{factorial}(1 + i)}$

[-3] n >= 0

[-4] i <= n

[-5] a > 0

|-----

[1] i = n

[2]  $HI = a * \text{factorial}(n) / \text{factorial}(i)$

Rule? (typepred "HI")

Adding type constraints for HI,  
this simplifies to:

fact\_it2\_TCC5.1 :

{-1} HI > 0

{-2}  $HI = \frac{\text{factorial}(n) * a + \text{factorial}(n) * a * i}{\text{factorial}(1 + i)}$

[-3] n >= 0

[-4] i <= n

[-5] a > 0

|-----

[1] i = n

[2]  $HI = a * \text{factorial}(n) / \text{factorial}(i)$

Rule? (expand "factorial" -2 3)

Expanding the definition of factorial,  
this simplifies to:

fact\_it2\_TCC5.1 :

[-1] HI > 0

{-2} HI =

$$\frac{(\text{factorial}(n) * a + \text{factorial}(n) * a * i)}{(\text{factorial}(i) + \text{factorial}(i) * i)}$$

[-3] n >= 0

[-4] i <= n

[-5] a > 0

|-----

[1] i = n

[2] HI = a \* factorial(n) / factorial(i)

Rule? (expand "factorial" -2 3)  
 Expanding the definition of factorial,  
 this simplifies to:  
 fact\_it2\_TCC5.1 :

[-1] HI > 0

{-2} HI =

$$\frac{(\text{factorial}(n) * a + \text{factorial}(n) * a * i) / (\text{factorial}(i) + \text{factorial}(i) * i)}$$

[-3] n >= 0

[-4] i <= n

[-5] a > 0

|-----

[1] i = n

[2] HI = a \* factorial(n) / factorial(i)

Rule? (replaces -2)

Iterating REPLACE,  
this simplifies to:  
fact\_it2\_TCC5.1 :

{-1}  $(\text{factorial}(n) * a + \text{factorial}(n) * a * i) /$   
       $(\text{factorial}(i) + \text{factorial}(i) * i)$   
       $> 0$

{-2}  $n \geq 0$

{-3}  $i \leq n$

{-4}  $a > 0$

|-----

{1}  $i = n$

{2}  $(\text{factorial}(n) * a + \text{factorial}(n) * a * i) /$   
       $(\text{factorial}(i) + \text{factorial}(i) * i)$   
       $= a * \text{factorial}(n) / \text{factorial}(i)$



Rule? (replaces -2)

Iterating REPLACE,  
this simplifies to:  
fact\_it2\_TCC5.1 :

$$\{-1\} \quad \frac{\text{factorial}(n) * a + \text{factorial}(n) * a * i}{\text{factorial}(i) + \text{factorial}(i) * i} > 0$$

$$\{-2\} \quad n \geq 0$$

$$\{-3\} \quad i \leq n$$

$$\{-4\} \quad a > 0$$

|-----

$$\{1\} \quad i = n$$

$$\{2\} \quad \frac{\text{factorial}(n) * a + \text{factorial}(n) * a * i}{\text{factorial}(i) + \text{factorial}(i) * i} = a * \text{factorial}(n) / \text{factorial}(i)$$

Rule? (grind-reals)

Rewriting with `pos_div_gt`

Rewriting with `cross_mult`

Applying GRIND-REALS,

This completes the proof of `fact_it2_TCC5.1`.

- ▶ All the other subgoals are discharged by `(assert)`.

# Induction-Free Induction

- + Induction scheme based the recursive definition of the function not on the measure function!.
- + Proofs exploit type-checker power.
  - Some TCCs look scary (but they are easy to tame)
  - If you modify the definitions, the TCCs get re-arranged (be careful or you can lose your proof)
- ? Can this method be used when the recursive function was not originally typed that way?

## Recursive Judgments

Consider the Ackermann function:

$$A(m,n) = \begin{cases} n + 1, & \text{if } m = 0 \\ A(m - 1, 1), & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)), & \text{otherwise.} \end{cases}$$

In PVS:

```
ack(m,n) : RECURSIVE nat =  
  IF      m = 0 THEN n+1  
  ELSIF  n = 0 THEN ack(m-1,1)  
  ELSE   ack(m-1,ack(m,n-1))  
  ENDIF  
MEASURE ?
```

## Recursive Judgments

Consider the Ackermann function:

$$A(m,n) = \begin{cases} n + 1, & \text{if } m = 0 \\ A(m - 1, 1), & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)), & \text{otherwise.} \end{cases}$$

In PVS:

```
ack(m,n) : RECURSIVE nat =  
  IF      m = 0 THEN n+1  
  ELSIF  n = 0 THEN ack(m-1,1)  
  ELSE   ack(m-1,ack(m,n-1))  
  ENDIF  
MEASURE lex2(m,n)
```

# Ackermann

Proving this fact:

$$\forall m, n : A(m, n) > m + n$$

by regular induction is not trivial: you may need two nested inductions!

# Recursive Judgements

```
ack_gt_m_n : RECURSIVE JUDGEMENT  
  ack(m,n) HAS_TYPE above(m+n)
```

The type checker generates TCCs corresponding to the recursive definition of the type-restricted version of ack, e.g.,

```
ack_gt_m_n_TCC1: OBLIGATION FORALL (m, n: nat): m=0 IMPLIES  
  n+1 > m+n;
```

```
ack_gt_m_n_TCC3: OBLIGATION  
  FORALL (v: [d: [nat, nat] -> above(d'1+d'2)], m, n: nat):  
    n=0 AND NOT m=0 IMPLIES v(m-1, 1) > m+n;
```

```
ack_gt_m_n_TCC7: OBLIGATION  
  FORALL (v: [d: [nat, nat] -> above(d'1+d'2)], m, n: nat):  
    NOT n=0 AND NOT m=0 IMPLIES v(m-1, v(m, n-1)) > m+n;
```

# Recursive Judgements

```
ack_gt_m_n : RECURSIVE JUDGEMENT
  ack(m,n) HAS_TYPE above(m+n)
```

The type checker generates TCCs corresponding to the recursive definition of the type-restricted version of ack, e.g.,

```
ack_gt_m_n_TCC1: OBLIGATION FORALL (m, n: nat): m=0 IMPLIES
  n+1 > m+n;
```

```
ack_gt_m_n_TCC3: OBLIGATION
  FORALL (v: [d: [nat, nat] -> above(d'1+d'2)], m, n: nat):
    n=0 AND NOT m=0 IMPLIES v(m-1, 1) > m+n;
```

```
ack_gt_m_n_TCC7: OBLIGATION
  FORALL (v: [d: [nat, nat] -> above(d'1+d'2)], m, n: nat):
    NOT n=0 AND NOT m=0 IMPLIES v(m-1, v(m, n-1)) > m+n;
```



## PVS Automatically Uses Judgements

Most of these TCCs are automatically discharged by the type checker (in this case, all of them). Furthermore, the theorem prover automatically uses judgements:

```
ack_simple_property :
```

```
  |-----  
{1}  FORALL (m, n): ack(m, n) > max(m, n)
```

Rule? (grind)

Rewriting with max

Trying repeated skolemization, instantiation, and if-lifting,  
Q.E.D.

## PVS Automatically Uses Judgements

Most of these TCCs are automatically discharged by the type checker (in this case, all of them). Furthermore, the theorem prover automatically uses judgements:

```
ack_simple_property :
```

```
  |-----  
{1}  FORALL (m, n): ack(m, n) > max(m, n)
```

Rule? (grind)

Rewriting with max

Trying repeated skolemization, instantiation, and if-lifting,  
Q.E.D.

# Iterations

```
/* Pre: n >= 0 */
int a = 1;
for (int i=0;i < n;i++) {
  /* Inv: a = i! */
  a = a*(i+1);
}
/* Post: a = n! */
```

In PVS:

```
IMPORTING structures@for_iterate
```

```
fact_for(n:nat) : real =
  for[real](0,n-1,1,LAMBDA(i:below(n),a:real):
    a*(i+1))
```

# Iterations

```
/* Pre: n >= 0 */
int a = 1;
for (int i=0;i < n;i++) {
  /* Inv: a = i! */
  a = a*(i+1);
}
/* Post: a = n! */
```

In PVS:

```
IMPORTING structures@for_iterate
```

```
fact_for(n:nat) : real =
  for[real](0,n-1,1,LAMBDA(i:below(n),a:real):
    a*(i+1))
```

# Iterations

```
/* Pre: n >= 0 */
int a = 1;
for (int i=0;i < n;i++) {
  /* Inv: a = i! */
  a = a*(i+1);
}
/* Post: a = n! */
```

In PVS:

```
IMPORTING structures@for_iterate
```

```
fact_for(n:nat) : real =
  for[real](0,n-1,1,LAMBDA(i:below(n),a:real):
    a*(i+1))
```

# Iterations

```
/* Pre: n >= 0 */
int a = 1;
for (int i=0;i < n;i++) {
  /* Inv: a = i! */
  a = a*(i+1);
}
/* Post: a = n! */
```

In PVS:

```
IMPORTING structures@for_iterate
```

```
fact_for(n:nat) : real =
  for[real](0,n-1,1,LAMBDA(i:below(n),a:real):
    a*(i+1))
```

# Iterations

```
/* Pre: n >= 0 */
int a = 1;
for (int i=0;i < n;i++) {
  /* Inv: a = i! */
  a = a*(i+1);
}
/* Post: a = n! */
```

In PVS:

```
IMPORTING structures@for_iterate
```

```
fact_for(n:nat) : real =
  for[real](0,n-1,1,LAMBDA(i:below(n),a:real):
    a*(i+1))
```

# Proving Correctness of Iterations

Consider the following implementation of factorial:

```
fact_for : THEOREM
  fact_for(n) = factorial(n)
```

```
fact_for :
```

```
  |-----
{1}  FORALL (n: nat): fact_for(n) = factorial(n)
```

Rule? (skeep)(expand "fact\_for")

```
fact_for :
```

```
  |-----
{1}  for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =
      factorial(n)
```



# Proving Correctness of Iterations

Consider the following implementation of factorial:

```
fact_for : THEOREM
  fact_for(n) = factorial(n)
```

```
fact_for :
```

```
  |-----
{1}  FORALL (n: nat): fact_for(n) = factorial(n)
```

Rule? (skeep)(expand "fact\_for")

```
fact_for :
```

```
  |-----
{1}  for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =
      factorial(n)
```

Rule? (lemma "for\_induction[real]")

Applying for\_induction[real]

this simplifies to:

fact\_for :

```
{-1}  FORALL (i, j: int, a: real, f: ForBody[real](i, j),
           inv: PRED[[UpTo[real](1 + j - i), real]]):
  (inv(0, a) AND
   (FORALL (k: subrange(0, j - i), ak: real):
    inv(k, ak) IMPLIES inv(k + 1, f(i + k, ak))))
  IMPLIES inv(j - i + 1, for(i, j, a, f))
```

|-----

```
[1]  for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =
      factorial(n)
```

Rule? (lemma "for\_induction[real]")

Applying for\_induction[real]

this simplifies to:

fact\_for :

```
{-1}  FORALL (i, j: int, a: real, f: ForBody[real](i, j),
           inv: PRED[[UpTo[real](1 + j - i), real]]):
  (inv(0, a) AND
   (FORALL (k: subrange(0, j - i), ak: real):
    inv(k, ak) IMPLIES inv(k + 1, f(i + k, ak))))
  IMPLIES inv(j - i + 1, for(i, j, a, f))
```

|-----

```
[1]  for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =
      factorial(n)
```

Rule? (inst?)

Instantiating quantified variables,  
this yields 2 subgoals:

fact\_for.1 :

```
{-1}  FORALL (inv:PRED[[UpTo[real](n)real]]):  
      (inv(0,1) AND  
       (FORALL (k:subrange(0,n-1),ak:real):  
         inv(k,ak) IMPLIES inv(k+1,ak+ak*(0+k))))  
      IMPLIES  
      inv(n,  
         for(0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i))  
      |-----  
[1]   for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =  
      factorial(n)
```

Rule? (inst?)

Instantiating quantified variables,  
this yields 2 subgoals:

fact\_for.1 :

```
{-1}  FORALL (inv:PRED[[UpTo[real](n)real]]):  
      (inv(0,1) AND  
        (FORALL (k:subrange(0,n-1),ak:real):  
          inv(k,ak) IMPLIES inv(k+1,ak+ak*(0+k))))  
      IMPLIES  
      inv(n,  
        for(0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i))  
      |-----  
[1]   for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =  
      factorial(n)
```

```
Rule? (inst -1 "LAMBDA(i:upto(n),a:real) : a = factorial(i)")
fact_for.1.1 :
```

```
{-1} ...
```

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

```
[1]   for[real](0,n-1,1,LAMBDA (i:below(n),a:real):a+a*i) =
      factorial(n)
```

- ▶ The variable  $i$  in the invariant refers to the  $i$ th iteration.
- ▶ Remaining subgoals are discharged with (grind). See `Examples/Lecture-2.pvs`.

# Inductive Definitions

- ▶ An inductive definition gives rules for generating members of a set.
- ▶ An object is in the set, only if it has been generated according to the rules.
- ▶ An inductively defined set is the smallest set closed under the rules.
- ▶ PVS automatically generates weak and strong induction schemes that are used by command `(rule-induct "<name>")` command .

## Even and Odd

```
even(n:nat): INDUCTIVE bool =  
  n = 0 OR (n > 1 AND even(n - 2))
```

```
odd(n:nat): INDUCTIVE bool =  
  n = 1 OR (n > 1 AND odd(n - 2))
```



## Induction Schemes

The definition of `even` generates the following induction schemes (use the Emacs command `M-x ppe`):

```
even_weak_induction: AXIOM
  FORALL (P: [nat -> boolean]):
    (FORALL (n: nat): n = 0 OR (n > 1 AND P(n - 2))
      IMPLIES P(n))
  IMPLIES
    (FORALL (n: nat): even(n) IMPLIES P(n));
```

```
even_induction: AXIOM
  FORALL (P: [nat -> boolean]):
    (FORALL (n: nat):
      n = 0 OR (n > 1 AND even(n - 2) AND P(n - 2))
      IMPLIES P(n))
  IMPLIES (FORALL (n: nat): even(n) IMPLIES P(n));
```

# Inductive Proof

```
even_odd :
```

```
  |-----  
{1}  FORALL (n: nat): even(n) => odd(n + 1)
```

```
Rule? (rule-induct "even")
```

Applying rule induction over even, this simplifies to:

```
even_odd :
```

```
  |-----  
{1}  FORALL (n: nat):  
      n = 0 OR (n > 1 AND odd(n - 2 + 1)) IMPLIES odd(n + 1)
```

The proof can then be completed using

```
(skosimp*)(rewrite "odd" +)(ground)
```

# Inductive Proof

```
even_odd :
```

```
  |-----  
{1}  FORALL (n: nat): even(n) => odd(n + 1)
```

```
Rule? (rule-induct "even")
```

Applying rule induction over even, this simplifies to:

```
even_odd :
```

```
  |-----  
{1}  FORALL (n: nat):  
      n = 0 OR (n > 1 AND odd(n - 2 + 1)) IMPLIES odd(n + 1)
```

The proof can then be completed using

```
(skosimp*)(rewrite "odd" +)(ground)
```

## Mutual Recursion and Higher-Order Recursion

The predicates `odd` and `even` can be defined using a mutual-recursion:

```
even?(0) = true
odd?(0)  = false
odd?(1)  = true
even?(n + 1) = odd?(n)
odd?(n + 1) = even?(n)
```

## In PVS ...

```
my_even?(n) : INDUCTIVE bool =  
  n = 0 OR n > 0 AND my_odd?(n-1)
```

```
my_odd?(n) : INDUCTIVE bool =  
  n = 1 OR n > 1 AND my_even?(n-1)
```

- ▶ These definitions don't type-check. What is wrong with them?
- ▶ PVS does not (directly) support mutual recursion.

## In PVS ...

```
my_even?(n) : INDUCTIVE bool =  
  n = 0 OR n > 0 AND my_odd?(n-1)
```

```
my_odd?(n) : INDUCTIVE bool =  
  n = 1 OR n > 1 AND my_even?(n-1)
```

- ▶ These definitions don't type-check. What is wrong with them?
- ▶ PVS does not (directly) support mutual recursion.

## Mutual Recursion via Higher-Order Recursion

```
even_f?(fodd:[nat->bool],n) : bool =  
  n = 0 OR  
  n > 0 AND fodd(n-1)
```

```
my_odd?(n) : INDUCTIVE bool =  
  n = 1 OR  
  n > 1 AND even_f?(my_odd?,n-1)
```

```
my_even?(n) : bool =  
  even_f?(my_odd?,n)
```

## Mutual Recursion via Higher-Order Recursion

```
even_f?(fodd:[nat->bool],n) : bool =  
  n = 0 OR  
  n > 0 AND fodd(n-1)
```

```
my_odd?(n) : INDUCTIVE bool =  
  n = 1 OR  
  n > 1 AND even_f?(my_odd?,n-1)
```

```
my_even?(n) : bool =  
  even_f?(my_odd?,n)
```



## Mutual Recursion via Higher-Order Recursion

```
even_f?(fodd:[nat->bool],n) : bool =  
  n = 0 OR  
  n > 0 AND fodd(n-1)
```

```
my_odd?(n) : INDUCTIVE bool =  
  n = 1 OR  
  n > 1 AND even_f?(my_odd?,n-1)
```

```
my_even?(n) : bool =  
  even_f?(my_odd?,n)
```