Recursion, Induction, and Iteration

César A. Muñoz

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

PVS Class 2012

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:

```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;
A Simple Property of Sum

We would like to prove the following closed form solution to \( \sum_{i=0}^{n} i \):

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

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.
\[
\text{nat\_induction: LEMMA} \quad \begin{align*}
    & (p(0) \text{ AND } (\text{FORALL } j: p(j) \text{ IMPLIES } p(j+1))) \\
    & \text{IMPLIES } (\text{FORALL } i: p(i))
\end{align*}
\]

% Strong induction on naturals.
\[
\text{NAT\_induction: LEMMA} \quad \begin{align*}
    & (\text{FORALL } j: (\text{FORALL } k: k < j \text{ IMPLIES } p(k)) \text{ IMPLIES } p(j)) \\
    & \text{IMPLIES } (\text{FORALL } i: p(i))
\end{align*}
\]

### Proof by Induction

\[
\text{closed\_form :}
\]
\[
\begin{align*}
& |----- \\
{1} & \text{FORALL } (n: \text{nat}): \text{sum}(n) = (n \ast (n + 1)) / 2
\end{align*}
\]

\text{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.

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

Rule? \textbf{(expand } \textit{"sum" +)}
Expanding the definition of \textit{sum},
this simplifies to:
closed_form.2 :

[-1] \sum(j) = (j \times (j + 1)) / 2
\ |
|--------
{1} 1 + \sum(j) + j = (2 + j + (j \times j + 2 \times j)) / 2

Rule? \textbf{(assert)}
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of closed_form.2.

Q.E.D.
Automated Simple Induction Proofs

\[
\text{|------}
\{1\} \quad \text{FORALL} \ (n: \text{nat}): \text{sum}(n) = (n \ast (n + 1)) / 2
\]

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:

\[
\text{factorial(n : nat): RECURSIVE posnat =}
\]

\[
\text{IF n = 0 THEN 1 ELSE n * factorial(n - 1) ENDIF}
\]

MEASURE n
A Simple Property of Factorial

\[ \forall n : n! > n \]

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:

```c
/* 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 ...

\[
\begin{align*}
fact_it(n:nat, i: upto(n), a: posnat) : & \text{ RECURSIVE posnat} = \\
& \begin{cases} 
  a & \text{if } i = n \\
  fact_it(n, i+1, a*(i+1)) & \text{else }
\end{cases} \\
& \text{MEASURE } n-i
\end{align*}
\]

\[
\text{fact_it}_\text{correctness} : \text{ THEOREM} \\
fact_it(n, 0, 1) = \text{factorial}(n)
\]

Proving \text{fact_it}_\text{correctness}

\[
\begin{align*}
{1} & \quad \text{FORALL } (n: \text{nat}): \text{fact_it}(n, 0, 1) = \text{factorial}(n) \\
\hline
\text{Rule? } (\text{induct-and-simplify } "n") \\
\text{this simplifies to:} \\
\text{fact_it}_\text{correctness} : \\
{-1} & \quad \text{fact_it}(j!1, 0, 1) = \text{factorial}(j!1) \\
\hline
{1} & \quad \text{fact_it}(1 + j!1, 1, 1) = \\
& \quad \text{factorial}(j!1) + \text{factorial}(j!1) \times j!1
\end{align*}
\]

The proof by (explicit) induction requires an inductive proof of an auxiliary lemma.
Induction-Free Induction By Predicate Subtyping

```verbatim
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 (skeep) (assert) QED
```

There is No Free Lunch

```verbatim
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)
```
Recursion, Induction, and Iteration

Induction-Free Induction

Rule? (expand "factorial" 2)

\[
\text{fact_it\_TCC4 :}
\]

\begin{align*}
[-1] & \quad n \geq 0 \\
[-2] & \quad i \leq n \\
[-3] & \quad a = \text{factorial}(i) \\
\end{align*}

|-------

\begin{align*}
[1] & \quad i = n \\
\{2\} & \quad a \times i + a = \text{factorial}(i) + \text{factorial}(i) \times i
\end{align*}

Rule? (assert)

Q.E.D.

You Can Also Pay at the Exit

\[
\text{fact_it2}(n:\text{nat},i:\text{upto}(n),a:\text{posnat}) : \text{RECURSIVE}
\]

\[
\{b:\text{posnat} \mid b = a \times \text{factorial}(n) / \text{factorial}(i)\}
\]

\[
\begin{align*}
\text{IF} & \quad i = n \text{ THEN } a \\
\text{ELSE} & \quad \text{fact_it2}(n,i+1,a \times (i+1)) \\
\text{ENDIF}
\end{align*}
\]

\[
\text{MEASURE } n-i
\]

\[
\text{fact_it2\_correctness : LEMMA}
\]

\[
\text{fact_it2}(n,0,1) = \text{factorial}(n)
\]
Recursion, Induction, and Iteration

Induction-Free Induction

|-------
{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]  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

\[
\text{fact\_it2\_TCC5: OBLIGATION} \quad \text{FORALL } (n: \text{nat}, i: \text{upto}(n), v:\n\begin{array}{l}
[d1: z: [n: \text{nat}, \text{upto}(n), \text{posnat}] | \\
\quad z'1 - z'2 < n - i -> \\
\quad b: \text{posnat} | b = d1'3 \times \text{factorial}(d1'1) / \\
\quad \text{factorial}(d1'2)], \\
\quad a: \text{posnat}):
\end{array}
\]
\begin{array}{l}
\text{NOT } i = n \text{ IMPLIES } \\
v(n, i + 1, a \times (i + 1)) = \\
a \times \text{factorial}(n) / \text{factorial}(i);
\end{array}
\]

Rule? (skeep :preds? t)

Skolemizing with the names of the bound variables, this simplifies to:
\[
\text{fact\_it2\_TCC5 :}
\]
\[
\begin{array}{l}
{-1} \quad n \geq 0 \\
{-2} \quad i \leq n \\
{-3} \quad a > 0 \\
\quad |-------- \\
{1} \quad i = n \\
{2} \quad v(n, i + 1, a \times (i + 1)) = a \times \text{factorial}(n) / \\
\quad \text{factorial}(i)
\end{array}
\]
Rule? (name-replace "HI" "v(n, i + 1, a \cdot (i + 1))")
Using HI to name and replace v(n, i + 1, a \cdot (i + 1)), this yields 2 subgoals:

```
|-------
[1] i = n
{2} HI = a * factorial(n) / factorial(i)
```

Adding type constraints for HI, this simplifies to:

```
|-------
[1] i = n
{2} HI = a * factorial(n) / factorial(i)
```
Rule? (expand "factorial" -2 3)
Expanding the definition of factorial, this simplifies to:

\[
\text{fact_it2_TCC5.1 :}
\]

\[
\begin{align*}
\{-1\} & \quad \text{HI} > 0 \\
\{-2\} & \quad \text{HI} = \\
& \quad \frac{\text{factorial}(n) \times a + \text{factorial}(n) \times a \times i}{\text{factorial}(i) + \text{factorial}(i) \times i} \\
\{-3\} & \quad n \geq 0 \\
\{-4\} & \quad i \leq n \\
\{-5\} & \quad a > 0 \\
\mid & \quad \text{-------} \\
\{1\} & \quad i = n \\
\{2\} & \quad \text{HI} = a \times \text{factorial}(n) / \text{factorial}(i)
\end{align*}
\]

Rule? (replaces -2)

Iterating REPLACE, this simplifies to:

\[
\text{fact_it2_TCC5.1 :}
\]

\[
\begin{align*}
\{-1\} & \quad \text{factorial}(n) \times a + \text{factorial}(n) \times a \times i \\
& \quad \text{factorial}(i) + \text{factorial}(i) \times i \\
& \quad > 0 \\
\{-2\} & \quad n \geq 0 \\
\{-3\} & \quad i \leq n \\
\{-4\} & \quad a > 0 \\
\mid & \quad \text{-------} \\
\{1\} & \quad i = n \\
\{2\} & \quad \text{factorial}(n) \times a + \text{factorial}(n) \times a \times i \\
& \quad \text{factorial}(i) + \text{factorial}(i) \times i \\
& \quad = a \times \text{factorial}(n) / \text{factorial}(i)
\end{align*}
\]
Recursion, Induction, and Iteration

Induction-Free Induction

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:

\[
\text{ack}(m,n) : \text{RECURSIVE nat} = \\
\quad \text{IF } m = 0 \text{ THEN } n+1 \\
\quad \text{ELSIF } n = 0 \text{ THEN } \text{ack}(m-1,1) \\
\quad \text{ELSE } \text{ack}(m-1,\text{ack}(m,n-1)) \\
\quad \text{ENDIF} \\
\text{MEASURE } ?\text{lex2}(m,n)
\]

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

\( \text{ack}_{\text{gt}}_{\text{m}_n} : \text{RECURSIVE JUDGEMENT} \)
\( \text{ack}(m,n) \text{ HAS_TYPE above}(m+n) \)

The type checker generates TCCs corresponding to the recursive
definition of the type-restricted version of \( \text{ack} \), e.g.,

\( \text{ack}_{\text{gt}}_{\text{m}_n}_{\text{TCC1}}: \text{OBLIGATION} \) \( \text{FORALL} (m, n: \text{nat}): m=0 \text{ IMPLIES } n+1 > m+n; \)

\( \text{ack}_{\text{gt}}_{\text{m}_n}_{\text{TCC3}}: \text{OBLIGATION} \)
\( \text{FORALL} (v: [d: [\text{nat, nat}] \rightarrow \text{above}(\text{d'1+d'2})], m, n: \text{nat}): \)
\( \text{n=0 AND NOT m}=0 \text{ IMPLIES } v(m-1, 1) > m+n; \)

\( \text{ack}_{\text{gt}}_{\text{m}_n}_{\text{TCC7}}: \text{OBLIGATION} \)
\( \text{FORALL} (v: [d: [\text{nat, nat}] \rightarrow \text{above}(\text{d'1+d'2})], m, n: \text{nat}): \)
\( \text{NOT n}=0 \text{ AND NOT m}=0 \text{ IMPLIES } v(m-1, v(m, n-1)) > m+n; \)

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:

\( \text{ack\_simple\_property} : \)
\n\n|--------
\{1\} \text{ 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))

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:

\[
\text{fact}_\text{for}:
\]

\[
\begin{align*}
\{-1\} & \quad \forall i, j: \text{int}, a: \text{real}, f: \text{ForBody}[\text{real}](i, j), \\
& \quad \quad \quad \quad \quad \quad \quad \text{inv}: \text{PRED}[[\text{UpTo}[\text{real}](1 + j - i), \text{real}]]: \\
& \quad \quad \quad \quad \quad \quad \quad (\text{inv}(0, a) \land \\
& \quad \quad \quad \quad \quad \quad \quad (\forall k: \text{subrange}(0, j - i), ak: \text{real}: \\
& \quad \quad \quad \quad \quad \quad \quad \text{inv}(k, ak) \Rightarrow \text{inv}(k + 1, f(i + k, ak)))) \\
& \quad \quad \quad \quad \text{IMPLIES} \quad \text{inv}(j - i + 1, \text{for}(i, j, a, f))
\end{align*}
\]

|-------
[1] \text{for[real]}(0, n - 1, 1, \lambda (i: \text{below}(n), a: \text{real}): a + a \times i) = \text{factorial}(n)

Rule? (inst?)
Instantiating quantified variables,
this yields 2 subgoals:

\[
\text{fact}_\text{for}.1:
\]

\[
\begin{align*}
\{-1\} & \quad \forall \text{inv}: \text{PRED}[[\text{UpTo}[\text{real}](n)\text{real}]]: \\
& \quad \quad \quad (\text{inv}(0,1) \land \\
& \quad \quad \quad (\forall k: \text{subrange}(0, n - 1), ak: \text{real}: \\
& \quad \quad \quad \text{inv}(k, ak) \Rightarrow \text{inv}(k+1, ak+ak\times(0+k)))) \\
& \quad \quad \text{IMPLIES} \\
& \quad \quad \text{inv}(n, \\
& \quad \quad \quad \text{for}(0, n - 1, 1, \lambda (i: \text{below}(n), a: \text{real}): a + a \times i))
\end{align*}
\]

|-------
[1] \text{for[real]}(0, n - 1, 1, \lambda (i: \text{below}(n), a: \text{real}): a + a \times i) = \text{factorial}(n)
Rule? (inst -1 "LAMBDA(i:upto(n),a:real) : a = factorial(i)"

\texttt{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.

\section*{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 (\texttt{rule-induct "<name>"}) command.
Even and Odd

\[
even(n:\text{nat}): \text{INDUCTIVE bool} = \\
n = 0 \ \text{OR} \ (n > 1 \ \text{AND} \ even(n - 2))
\]

\[
odd(n:\text{nat}): \text{INDUCTIVE bool} = \\
n = 1 \ \text{OR} \ (n > 1 \ \text{AND} \ odd(n - 2))
\]

Induction Schemes

The definition of \text{even} generates the following induction schemes (use the Emacs command \texttt{M-x ppe)}:

\[
even\_weak\_induction: \text{AXIOM} \\
\text{FORALL} \ (P: [\text{nat -> boolean}]): \\
\text{(FORALL} \ (n: \text{nat}): \ n = 0 \ \text{OR} \ (n > 1 \ \text{AND} \ P(n - 2)) \\
\implies \ P(n)) \\
\implies \ (\text{FORALL} \ (n: \text{nat}): even(n) \implies P(n));
\]

\[
even\_induction: \text{AXIOM} \\
\text{FORALL} \ (P: [\text{nat -> boolean}]): \\
\text{(FORALL} \ (n: \text{nat}): \ n = 0 \ \text{OR} \ (n > 1 \ \text{AND} \ even(n - 2) \ \text{AND} \ P(n - 2)) \\
\implies \ P(n)) \\
\implies \ (\text{FORALL} \ (n: \text{nat}): even(n) \implies P(n));
\]
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)

Mutual Recursion and Higher-Order Recursion

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

\[
\begin{align*}
even?(0) &= true \\
odd?(0) &= false \\
odd?(1) &= true \\
even?(n+1) &= odd?(n) \\
odd?(n+1) &= even?(n)
\end{align*}
\]
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)
```

▶ Theses 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)
```

The only recursive definition is my_odd?.