Recursion and Induction

Paul S. Miner

NASA Langley Formal Methods Group

p.s.miner@nasa.gov

28 November 2007

Outline

Recursive definitions in PVS

Simple inductive proofs

Automated proofs by induction

More complicated induction problems

Equivalence between recursive algorithms

Induction on lists

Inductive definitions

definition by recursion

Suppose we want to define a function to sum the first n natural numbers. The conventional notation for this is:

$$\sum_{i=0}^{n} i$$

We may define this in PVS using recursion:

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

As a result of this definition, PVS generates two proof obligations called *Type Correctness Conditions* or TCCs. Proofs involving this function will not be considered complete until all TCCs have been proven.

type correctness conditions for sum

- The first TCC is to ensure that the argument for the recursive call is a natural number. This is necessary because the natural numbers are not closed under subtraction.

```
% Subtype TCC generated (line 8) for  n - 1
sum_TCC1: OBLIGATION
       (FORALL (n): NOT n = 0 IMPLIES n - 1 >= 0);
```

- The second TCC is to ensure that the recursion terminates:

```
% Termination TCC generated (line 8) for  sum
sum_TCC2: OBLIGATION
    (FORALL (n): NOT n = 0 IMPLIES n - 1 < n);
```

  This goal is determined from the recursive call and the MEASURE on the function arguments.

a simple property of sum

We'd like to prove the following closed form solution to sum:

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

In PVS, we state this goal as follows:

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

The usual approach for proving properties of recursively defined functions is to use induction.

induction proofs in PVS

(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 of closed_form using induction

```
closed_form :

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

Rule? (induct "n")
Inducting on n,
this yields  2 subgoals:
```

base case

```
closed_form.1 :
  |-------
{1}   sum(0) = (0 * (0 + 1)) / 2

Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
closed_form.1 :

  |-------
{1}   0 = 0 / 2

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

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

induction step

```
closed_form.2 :

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

Rule? (skosimp*)
Repeatedly Skolemizing and flattening,
this simplifies to:
closed_form.2 :

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

induction step (continued)

```
Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
closed_form.2 :

[-1]  sum(j!1) = (j!1 * (j!1 + 1)) / 2
  |-------
{1}   1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2

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

This completes the proof of closed_form.2.

Q.E.D.
```

automated proofs by induction

Suppose we want to prove things about summations of the form

$$\sum_{i=0}^{n} f(i)$$

for some function f.
The following definition allows us to sum the first n values of some function f.

```
n: VAR nat
f: VAR [nat -> nat]

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

closed_form revisited

The PVS prelude includes a definition for the *identity* function, *id*. This unary function simply returns its argument. That is, for all x:

$$id(x) = x$$

This allows a restatement of the earlier closed form result using the new definition of `sum`.

```
closed_form2: THEOREM
    sum(n, id) = (n * (n + 1)) / 2
```

PVS has some built-in strategies that automate the steps in a proof by induction. The above result can be proven using `(induct-and-simplify "n")`
The more general definition of summation allows us to establish results for more complicated summation expressions.

sum of squares

For example, if we define

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

Then, the sum of the first n squares is

```
sum(n, square)
```

with closed form solution

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

In PVS notation:

```
sum(n, square) = (n * (n + 1) * (2 * n + 1)) / 6
```

automated proofs

```
  |-------
{1}    (FORALL (n: nat):
         sum(n, square) = (n * (n + 1) * (2 * n + 1)) / 6)

Rerunning step: (induct-and-simplify "n")
square rewrites square(0)
  to  0
sum rewrites sum(0, square)
  to  0
square rewrites square(1 + j!1)
  to  1 + j!1 + (j!1 * j!1 + j!1)
sum rewrites sum(1 + j!1, square)
  to  1 + sum(j!1, square) + j!1 * j!1 + j!1 + j!1
By induction on n, and by repeatedly rewriting and simplifying,
Q.E.D.
```

limitations of automation

The strategy `(induct-and-simplify ...)` is useful for automating relatively simple proofs by induction. However, automation can sometimes lead us astray.
Consider the nth factorial:

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

In PVS:

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

Suppose we wish to prove for $n > 3$:

$$n! > 2^n$$

a series of unfortunate events ...

```
  |-------
{1}   FORALL (n: nat): n > 3 IMPLIES factorial(n) > 2 ^ n

Rule? (induct-and-simplify "n")
factorial rewrites factorial(j!1 - 2)
  to factorial(j!1 - 3) * j!1 - 2 * factorial(j!1 - 3)
     [...]
^ rewrites 2 ^ j!1
  to 8 * expt(2, j!1 - 3)
Warning: Rewriting depth = 50; Rewriting with factorial
Warning: Rewriting depth = 100; Rewriting with factorial
Warning: Rewriting depth = 150; Rewriting with factorial
Warning: Rewriting depth = 200; Rewriting with factorial
Warning: Rewriting depth = 250; Rewriting with factorial
```

Whenever PVS falls into an infinite loop, C-c C-c will force PVS to break into lisp. The lisp command (restore) will return you to the PVS state just prior to the last proof command.

let's try to induct directly on this goal

```
|-------
{1}   FORALL (n: nat): n > 3 IMPLIES factorial(n) > 2 ^ n

Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
factorial_gt_expt2_nat.1 :

|-------
{1}   0 > 3 IMPLIES factorial(0) > 2 ^ 0
```

This is a vacuous base case. It does not provide any information for the induction step to build on.

subsequent induction step

```
factorial_gt_expt2_nat.2 :

{-1}  j!1 > 3 IMPLIES factorial(j!1) > 2 ^ j!1
{-2}  j!1 + 1 > 3
  |-------
{1}   factorial(j!1 + 1) > 2 ^ (j!1 + 1)

Rule? (split)
```

The resulting sequents are the actual induction step and actual base case for the induction.

actual induction cases

The induction step:

```
factorial_gt_expt2_nat.2.1 :

{-1}  factorial(j!1) > 2 ^ j!1
[-2]  j!1 + 1 > 3
  |-------
[1]   factorial(j!1 + 1) > 2 ^ (j!1 + 1)
```

and the base case, $4! > 2^4$:

```
factorial_gt_expt2_nat.2.2 :

[-1]  j!1 + 1 > 3
  |-------
{1}   j!1 > 3
[2]   factorial(j!1 + 1) > 2 ^ (j!1 + 1)
```

choosing an appropriate induction scheme

The PVS Prelude includes both weak and strong induction schemes for various integer subtypes. These include:

```
integers: THEORY
[...]
  upfrom(i): NONEMPTY_TYPE = {s: int | s >= i} CONTAINING i
  above(i):  NONEMPTY_TYPE = {s: int | s > i}  CONTAINING i + 1
[...]
  subrange(i, j): TYPE = {k: int | i <= k AND k <= j}
[...]

naturalnumbers: THEORY
[...]
  upto(i):   NONEMPTY_TYPE = {s: nat | s <= i} CONTAINING i
  below(i):  TYPE = {s: nat | s < i}
[...]
```

Note that both `subrange(i, j)` and `below(i)` may be empty.

induction using integer subtype "above(3)"

```
factorial_gt_expt2_above3 :

  |-------
{1}   FORALL (x: above(3)): factorial(x) > 2 ^ x

Rule? (induct "x")
Inducting on x on formula 1,
this yields  2 subgoals:
factorial_gt_expt2_above3.1 :

  |-------
{1}   factorial(3 + 1) > 2 ^ (3 + 1)

Rule?
```

Note: The proof strategy
`(then (auto-rewrite-defs)(assert))` is useful for discharging subgoals of this form.

functional equivalence between recursive algorithms

The ith Fibonacci number is recursively defined by:

```
fib(i): RECURSIVE nat =
  IF    i = 0 THEN 1
  ELSIF i = 1 THEN 1
              ELSE (fib(i - 1) + fib(i - 2))
  ENDIF
  MEASURE i
```

The above recursive definition is computationally inefficient ($\mathcal{O}(2^i)$). An alternative $\mathcal{O}(i)$ (linear time) recursive algorithm for computing the ith Fibonacci is:

```
tfib(i, j, k): RECURSIVE nat =
  IF i = 0 THEN j
           ELSE tfib(i - 1, k, j + k)
  ENDIF
  MEASURE i
```

our goal

We'd like to prove:

```
tfib_fib: THEOREM fib(i) = tfib(i, 1, 1)
```

That is, we want to establish that the linear time algorithm computes the same function as the exponential time specification.

a fibonacci invariant

First, we introduce a lemma demonstrating that `tfib` satisfies the defining recurrence equation for `fib`

```
fib_tfib: LEMMA
   tfib(i + 2, j, k)
      = tfib(i + 1, j, k) + tfib(i, j, k)
```

We try the automated strategy (induct-and-simplify "i")

```
  |-------
{1}  (FORALL (i: nat, j: nat, k: nat):
        tfib(i + 2, j, k) = tfib(i + 1, j, k) + tfib(i, j, k))

Rule? (induct-and-simplify "i")
By induction on i, and by repeatedly rewriting and simplifying,
this simplifies to:
```

an unfortunate sequent

```
fib_tfib :

{-1}    tfib(j!1, j!2 + k!1, j!2 + 2 * k!1)
          = tfib(j!1, j!2, k!1) + tfib(j!1, k!1, j!2 + k!1)
  |-------
{1}    tfib(j!1, j!2 + 2 * k!1, 2 * j!2 + 3 * k!1)
         = tfib(j!1, j!2 + k!1, j!2 + 2 * k!1)
             + tfib(j!1, k!1, j!2 + k!1)

Rule? (undo)y
```

And we get a mess. We need a sequent where the arguments j and k match for both the goal and the induction hypothesis. The automated strategy guessed an incorrect instantiation. We have to manually step through the proof.
Hint: A relatively simple proof by induction works.
(left as an exercise)

using strong induction

With this lemma, we can now prove the desired equivalence:

```
tfib_fib: THEOREM fib(i) = tfib(i, 1, 1)
```

In this case, the default induction scheme is not strong enough to establish the result. The proof requires strong induction. This induction scheme has to be explicitly invoked.

```
  |-------
{1}    (FORALL (i: nat): fib(i) = tfib(i, 1, 1))

Rule? (induct "i" :name "NAT_induction")
Inducting on i using induction scheme NAT_induction,
this simplifies to:
tfib_fib :
  |-------
{1}    (FORALL (j: nat):
         (FORALL (k: nat): k < j IMPLIES fib(k) = tfib(k, 1, 1))
             IMPLIES fib(j) = tfib(j, 1, 1))
```

completing the proof

After expanding the definition of "fib" and discharging the cases for `j!1 = 0` and `j!1 = 1`, we are left with the following goal:

```
[-1]    (FORALL (k: nat): k < j!1 IMPLIES fib(k) = tfib(k, 1, 1))
  |-------
{1}    j!1 = 1
{2}    (fib(j!1 - 1) + fib(j!1 - 2)) = tfib(j!1, 1, 1)
{3}    j!1 = 0

Rule?
```

If we instantiate the antecedent twice; once with `j!1-1` and once with `j!1-2`, and invoke lemma `fib_tfib`, we can complete the proof. The PVS commands for the instantiations are:

```
(inst-cp - "j!1 - 1")
(inst - "j!1 - 2")
```

Command `(inst-cp ...)` preserves a copy of the instantiated formula in the resulting sequent.

induction on lists

The PVS prelude includes a declaration of lists as an abstract data type (more on how PVS does abstract datatypes in a later lecture). For each datatype declaration, PVS automatically generates an induction scheme.

```
list [T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons (car: T, cdr:list):cons?
 END list
```

list append

```
append(l1, l2): RECURSIVE list[T] =
  CASES l1 OF
    null: l2,
    cons(x, y): cons(x, append(y, l2))
  ENDCASES
  MEASURE length(l1)
```

associativity of append

```
append_assoc: LEMMA
  append(l1, append(l2, l3)) = append(append(l1, l2), l3)
```

This is easily proven using (induct-and-simplify “l1”)

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 schemas that are used by the (rule-induct ...) command

sample inductive definitions

```
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))
```

The definition of even generates the following induction schemas:

```
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));
```

sample proof using (rule-induct ...)

```
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)
```

exercise 6

If you get stuck, you may want to refer back to these slides.
All proofs begin with some form of induction. Do not begin any proofs in this exercise with any variation of (skolem ...).