ב״ה

Reasoning about Programs using Specifications and Induction

R.B. Yehezkael (formerly Haskell) December 2004 - כסלו תשס״ה

These notes are intended to explain inductive methods of reasoning to computer scientists and engineeers. The concepts presented are not novel; our aim is to facilitate their understanding and use.

What does it mean for a program to be correct? What is correctness?

Partial correctness: Halting is not guaranteed for all values of the inputs, but whenever the program halts normally, the results are correct according to the specifications.

Total Correctness: The program always halts and gives values which are correct according to the specifications.

Specifications: What the program should do and not how it does it, that is, what are the inputs and the outputs. While it is desirable to add to the specifications what happens when there are errors in the inputs, we shall not require this. We take the view that failure because of illegal inputs is a case of misuse of the program and not incorrectness. (Analogy: A light bulb which burns out immediately because it is connected to a very high voltage supply is not considered defective.) In writing specifications, one should give a clear and direct description of the final result only, and not describe intermediate results. The description should avoid the use of imperatives.

Computational induction: This is a technique for reasoning about programs. Specifically this technique can be used to prove or to justify that a program is partially correct. Proof of halting or total correctness will not be discussed here. For now, run the program on test data to gain confidence that it will halt.

Now for some examples.

Example 1

```
FUNCTION even_test(IN n:integer) RETURN boolean;
-- SPECIFICATION
-- the function even_test tests if n is even

FUNCTION odd_test(IN n:integer) RETURN boolean;
-- SPECIFICATION
-- the function odd_test tests if n is odd

--THE BODIES OF THE FUNCTIONS
```

```
FUNCTION even_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN true;
        ELSIF n>0
        THEN RETURN odd_test(n-1);
        ELSE RETURN odd_test(n+1);
        END IF;
END even_test;

FUNCTION odd_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN false;
        ELSIF n>0
        THEN RETURN even_test(n-1);
        ELSE RETURN even_test(n+1);
        END IF;
END odd_test;
```

How can we prove or justify that even_test successfully tests that n is even and that odd_test indeed tests that n is odd. If we try to read the program the way it is executed we get into a mess as even_test calls odd_test and odd_test calls even_test and we can go around endlessly trying to understand the program. The inductive way of reading the program in order to understand it is to read each function from beginning to end and try to justify each line in the program. When we reach a function or procedure call we do not jump anywhere in our reading of the program but instead simply assume that the call works to specification. If we do this and every line can be justified then theory shows that the program is partially correct.

Actually, computational induction is based on a form of simple induction on the length of the computation. (Induction on the number of function/procedure calls executed, may also be used.)

Let us now present two parallel proofs that even_test is partially correct using both these techniques. We use two columns where there are differences between the proofs and write the common parts once only across the width of the page.

if n=0 then even_test(n) = true and so even_test correctly tests if n is even.

if n>0 then even_test(n) = odd_test(n-1).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call odd_test(n-1) works correctly.	Assuming the computation halts normally, the length of computation of odd_test(n-1) is shorter than the length of computation of even_test(n). Therefore by simple induction we may assume that odd_test(n-1) works correctly.

So, odd_test(n-1) will test if n-1 is odd. But n-1 is odd means that n is even. So in this case even_test(n) tests correctly if n is even.

if n<0 then even_test(n) = odd_test(n+1).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call odd_test(n+1) works correctly.	Assuming the computation halts normally, the length of computation of odd_test(n+1) is shorter than the length of computation of even_test(n). Therefore by simple induction we may assume that odd_test(n+1) works correctly.

So, odd_test(n+1) will test if n+1 is odd. But n+1 is odd means that n is even. So in this case as well, even_test(n) tests correctly that n is even.

Similarly we can check the definition of odd_test in the same way.

So assuming halting, correct results will be obtained. That is, even_test and odd_test are partially correct.

Regarding halting, it has been shown by Turing and Godel that this can not be determined by a program or algorithm. There are however, systematic but not fully general methods, for proving halting. In this specific case, we see that the absolute value of n is being reduced (or simplified) on each function call so we can see that halting will occur as zero must be reached. This idea of "simplification" can be made mathematically precise - see Zohar Manna's book "Mathematical Theory of Computation" in particular the material on well founded orderings. We shall not discuss these concepts here.

Class discussion: How can a proof by computational induction be translated into a proof by induction on the length of computation? Does the existence of such a translation justify computational induction?

Exercise: Check the definition of odd_test by using computational induction and simple induction on the length of the computation.

Class Discussion: What would happen is we used computational induction to check the following definition of even_test. (Here, execution is non terminating when n≠0.)

```
FUNCTION even_test(IN n:integer) RETURN boolean IS
BEGIN
        IF n = 0
        THEN RETURN true;
        ELSIF n>0
        THEN RETURN odd_test(n+1);    -- error, should be n-1
        ELSE RETURN odd_test(n-1);    -- error, should be n+1
        END IF;
END even_test;
```

Example 2

```
size: CONSTANT integer := 100;

TYPE vector IS ARRAY (1..size) OF integer;

PROCEDURE swap (IN OUT v:vector; IN low,high: integer) IS
-- SPECIFICATION
-- exchange the values of the v(low) and v(high).

        .....
BEGIN
        .....
END swap;

PROCEDURE reverse(IN OUT v:vector; IN low,high: integer) IS
-- SPECIFICATION
-- When  low<high,
-- reverse the order of the elements of the vector v between "low" and "high".
-- When high≤low, no change is made to v.

BEGIN
        IF high > low
        THEN swap(v,low,high);
                reverse(v,low+1,high-1);
        ENDIF;
END reverse;
```

Again let us use computational induction and simple induction on the length of the computation to justify that the procedure reverse works.

Originally the elements are in the order :-
.....,v(low),v(low+1),.....,v(high-1),v(high),.....

If high ≤ low then the procedure does nothing which is in agreement with the specification.

If high > low then we execute swap(v,low,high).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call swap(v,low,high) works correctly.	Assuming the computation halts normally, the length of computation of swap(v,low,high) is shorter than the length of computation of reverse(v,low,high). Therefore by simple induction we may assume that swap(v,low,high) works correctly.

This means that the elements are in the order :-
.....,v(high),v(low+1),.....,v(high-1),v(low),.....

Then reverse(v,low+1,high-1) is called.

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call reverse(v,low+1,high-1) works correctly.	Assuming the computation halts normally, the length of computation of reverse(v,low+1,high-1) is shorter than the length of computation of reverse(v,low,high). Therefore by simple induction we may assume that reverse(v,low+1,high-1) works correctly.

So this means that the elements of v between low+1 and high-1 will be reversed. This means that the elements will now be in the order:-

.....,v(high),v(high-1),.....,v(low+1),v(low),.....

Which means that all the elements of the vector v between low and high have been reversed.

So assuming halting, correct results will be obtained. That is, the procedure reverse is partially correct.

Class Discussion

Can computational induction be used to check correctness of a single execution, assuming halting?
For example, to check the execution of reverse (c, 1, 5)
where c = (1, 2, 3, 4, 7) we have to perform swap (c, 1, 5)
so c is now (7, 2, 3, 4, 1) assuming swap (c, 1, 5) works.
Now we execute reverse (c, 2, 4)
and so c is now (7, 4, 3, 2, 1) assuming reverse (c, 2, 4) works.
So we see that in this single execution, assuming halting, that c is reversed.
The above is not a valid argument and c may not actually get this value. Why?

Example 3

In this example the programmer has made an error as indicated.

```
PROCEDURE reverseb(INOUT v:vector; IN low,high: integer) IS
-- SPECIFICATION AS BEFORE.

BEGIN
        IF high > low
        THEN swap(v,low,high);
                reverseb(v,low+2,high-2);   -- programmer error here
        ENDIF;
END reverseb;
```

Again let us use computational induction and simple induction on the length of the computation to find out that there is an error.

Originally the elements are in the order :-
.....,v(low),v(low+1),v(low+2),..... ,v(high-2),v(high-1),v(high),.....

If high $\leq$ low then the procedure does nothing which is in agreement with the specification.

If high > low then we execute swap(v,low,high).

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call swap(v,low,high) works correctly.	Assuming the computation halts normally, the length of computation of swap(v,low,high) is shorter than the length of computation of reverseb(v,low,high). Therefore by simple induction we may assume that swap(v,low,high) works correctly.

This means that the elements are in the order :-
.....,v(high),v(low+1),v(low+2),..... ,v(high-2),v(high-1),v(low),.....

Then reverseb(v,low+2,high-2) is called.

Computational Induction.	Induction on length of computation.
So by computational induction we may assume that the internal call reverseb(v,low+2,high-2) works correctly.	Assuming the computation halts normally, the length of computation of reverseb(v,low+2,high-2) is shorter than the length of computation of reverseb(v,low,high). Therefore by simple induction we may assume that reverseb(v,low+2,high-2) works correctly.

So this means that the elements of v between low+2 and high-2 will be reversed. This means that the elements will now be in the order:-

.....,v(high),v(low+1),v(high-2),..... ,v(low+2),v(high-1),v(low),.....

Which means there is an error.

Class Discussion

Can computational induction be used to check that a single execution is incorrect, assuming halting?
For example, to check the execution of reverseb (c, 1, 5)
where c = (1, 2, 3, 4, 7) we have to perform swap (c, 1, 5)
so c is now (7, 2, 3, 4, 1) assuming swap (c, 1, 5) works.
Now we execute reverseb (c, 3, 3)
and so c is now (7, 2, 3, 4, 1) assuming reverseb (c, 3, 3) works.
So we see that in this single execution, assuming halting, that there is an error and c is not reversed.
The above is not a valid argument and c may not actually get this value. Why?
Can we conclude there is an error in the function?

A subtle point

Let us assume that the program halts and that we use the method of computational induction to check it.

If it is correct the results obtained from the program, the specification, and from the computational induction proof itself, will all be identical.

Here is what happens when it is not correct. There will be different results from the specification and program. There will be different results from the specification and the computational induction proof. There may or may not be different results from the program and the computational induction proof.

The following example shows what can happen when execution does not halt.

Example 4

With this example, execution does not always halt.

```
FUNCTION ll (IN n:integer) RETURN integer IS
BEGIN
        IF n < 0
        THEN RETURN 0;
        ELSE RETURN ll (n+1);
        END IF;
END ll;
```

Also, it can be shown by computational induction that the above definition is partially correct with respect to both specifications below.

Specification 1: ll (n) = 0 when n < 0 and
ll (n) = -5 when n ≥ 0.

Specification 2: ll (n) = 0 when n < 0 and
ll (n) = +7 when n ≥ 0.

There seems to be a contradiction here but it is not a real contradiction. The apparent contradiction occurs for those values of n where the execution does not halt normally and the method of computational induction by definition says nothing about this situation. Indeed, if these apparent contradictions occur, this effectively shows that execution does not halt.

Class Discussion: How does the above example of not halting differ from the non halting execution presented in the class discussion following example 1?

Class Discussion: If there are operations to stop a (sub) computation, you can not assume that the length of a subcomputation is shorter that the computation itself. Under what circumstances can computation induction be used even in such a case?

NOTES

1) Computational induction does not use an explicit basis as the technique only guarantees that no false results are produced.

2) It is similar to the "step over" feature in debuggers for avoiding the displaying of detailed steps when executing functions or procedures. We recommend the use of the "step over" feature when debugging programs, and the use of computational induction when reasoning about programs. (They both prevent being swamped by too much information.)

3) Computational induction can be used for procedures and functions whether recursive or not, the use being identical in all cases.

Programming Large Systems

We have demonstrated how specifications and computational induction can be used to prove correctness of programs. For expository reasons, the examples we have given have been simple ones. Without automatic methods, large programs can not be proved correct because quite simply the proof is usually longer than the program itself. So if one has doubts about the correctness of the program, one will have even greater doubts about the validity of the proof! However the technique of assuming that internal function and procedure calls work when reading programs is a very powerful technique for understanding and checking large programs and systems.

For example, suppose we have a very large system in which P calls Q1,Q2,Q3. To check P using these methods, read the specifications of P,Q1,Q2,Q3 and assume that calls to them will work. Then check that the statements of P make

sense with respect to these assumptions. No other specifications and no other statements need to be looked at when checking or writing P.

This ability of looking at only one item (i.e. a function or procedure) in detail and items directly connected to it in summary (i.e. specifications) makes it easier to develop programs and systems within teams. A team member needs to know the specifications of the procedures and functions that he calls. He does not need to know the detailed statements in the procedures and functions written by others.

Finally, this is a fundamental technique of thinking and enables one to decide what needs to be known in summary, what needs to be known in detail, and what may be ignored. It enables the application of mental effort where it is needed and eases problem solving.

Class Discussion: A technique used for checking plans is to be optimistic about the outcome of all sublans tasks etc. How is this technique similar to Computational Induction?

Inductive Assertions or Loop Invariants

The principle of computational induction is an excellent aid for reasoning about procedure and function calls. Inductive assertions or loop invariants enable us to reason about loops. Again this technique assures partial correctness only - termination is not guaranteed. We illustrate this technique with an example

```
-- SPECIFICATION:
-- sq and n are non negative integers. sq = n^2 after executing the statements below.

i:=0; sq := 0;
LOOP

  -- Inductive Assertion or Loop Invariant:-  we claim that  sq = i^2
  WHEN i = n EXIT;
  sq := sq + 2*i +1;   -- the reason for this form will become clear below
  i := i + 1;

END LOOP;
```

Proof:

We have to guess some property that remains invariant when executing the loop. The property $sq = i^2$ is similar to the specification but uses the loop index i instead of n its final value. This is a good guess. We write this property before the exit conditions of the loop. i.e the WHEN statements in the loop.

Now we check if that this property is true when we first reach the assertion. Clearly this is so because sq = i = 0 and so $sq = i^2$ on loop entry.

Assume this property holds at the head of the loop and then prove that it will continue to hold after executing one more iteration of the loop. Let us say that

the value of i and sq on loop entry are i = k and sq = i^2 = k^2 . Now if we execute the loop once more the value of sq and i will now be :-
sq = k^2+ 2k + 1 =$(k+1)^2$
i=k+1

After executing the statements of the loop once more we have sq = i^2 just as before. So no matter how many times we go around the loop, sq = i^2 remains correct. We can only leave the loop when i = n and so when we exit the loop sq = n^2.

This demonstrates partial correctness of the statements above.

NOTE - to reason about the loop we had to introduce symbols for the values of i and sq at the head of the loop. This is common when reasoning about loops.

Handling Loops by Transforming them into Procedures

Another way of handling the proof of correctness of a loop, is to convert it to procedure(s), and in fact this can always be done systematically. Then use computational induction to prove the partial correctness of the procedures. For example, the previous loop and initialization code can be written as two procedures as follows.

```
PROCEDURE square(IN n:integer; OUT result:integer) IS
-- SPECIFICATION
-- result = n^2.

PROCEDURE looping(IN n,i,sq:integer; resullt:OUT integer) IS
-- SPECIFICATION
-- result = sq+n^2-i^2

--THE BODIES OF THE FUNCTIONS

PROCEDURE square(IN n:integer; OUT result:integer) IS

BEGIN -- square
  looping(n,0,0,result);
  -- this effectively initialize i and sq to zero for procedure looping.
END square;

PROCEDURE looping(IN n,i,sq:integer; resullt:OUT integer) IS

BEGIN -- looping
 IF i = n
 THEN result:= sq;
 ELSE looping(n, i + 1, sq + 2*i +1, result);
 ENDIF;
END looping;
```

Exercise: Prove by computational induction that the previous two procedures are partially correct.

SUMMARY

Specifications - What has to be done NOT how to do it.

Computational Induction - proof method for procedures and functions based on simple induction on the length of computation. Enables you to decide what can be ignored, what needs to be studied in summary (i.e. specifications), and what must be looked at in every detail (i.e. the procedure or function being checked).

Inductive Assertions or Loop Invariants - Proof technique for loops.

Handling Loops by Transforming them into Procedures - Proof technique for loops.

Other Inductive Techniques - Structural Induction can be used for proving halting or total correctness. It is a generalization of the principle of simple induction. We do not discuss this here and the interested reader is referred to Zohar Manna's book "Mathematical Theory of Computation".

Exercises:

1) Prove by computational induction that the following functions are partially correct

```
FUNCTION f(IN n:integer) RETURN integer IS
-- SPECIFICATION
-- f(n) = n!

  FUNCTION ff(IN m,n:integer) RETURN integer IS
  -- SPECIFICATION
  -- ff(n) = n! * m

  BEGIN -- ff
        IF n = 0
        THEN RETURN m;
        ELSE  RETURN ff(m*n,n-1);
        END IF;
  END ff;

BEGIN -- f
  RETURN ff(1,n);
END f;
```

2) Complete the following function using recursion. Use computational induction to check your procedure. DO NOT RUN the procedure.

```
size: CONSTANT integer :=100;

TYPE vector IS ARRAY (1..size) OF integer;

FUNCTION palindrome (IN a:vector; IN low,high:integer) RETURN boolean;
-- SPECIFICATION
-- the elements of the vector a(low),a(low+1), .... ,a(high-1),a(high) are not
-- affected by reversing them
```

3) The towers of Hanoi problem. You are given a pile of n disks of decreasing size where disk 1 is the largest and disk n the smallest. There are three pegs (A,B,C) on which disks may be placed. You are allowed to move one disk from one peg to another provided it goes on a larger disk. Initially all the disks are on peg A with the disk 1 at the bottom and the disk n on the top. The following program will move the disks to peg C in such a way that at no stage will you have a large disk on a small disk. Use computational induction to check this claim.

```
TYPE peg IS ('A','B','C');

PROCEDURE hanoi(IN n: integer; IN start,finish,extra:peg) IS

-- SPECIFICATION
-- Prints the moves needed for
-- moving disks 1,2,3, ... ,n  from peg "start" to peg "finish" using peg "extra"
-- as an auxiliary peg.  Never put a small disk on a large disk.

BEGIN -- hanoi
        IF n=1
        THEN put('MOVE'); put(start); put('TO'); put(finish); new_line;
        ELSE  hanoi(n-1,start,extra,finish);
                put('MOVE'); put(start); put('TO'); put(finish); new_line;
                hanoi(n-1,extra,finish,start);
        ENDIF;
END hanoi;

hanoi(4,'A','B','C');
```

Acknowledgment

My sincere thanks to E. Gensburger, E.Dashtt, A.E. Naiman, M. Reif, for their constructive comments and help.