Computational Induction

Abdelrahman Kamel
Cornell University
Formal Methods

December 3, 2011

Abstract

Computational Induction is used to prove the correctness of numerous problems in Computer Science. In this paper, we will observe the various types of Induction processes that can be used to solve simple and complex recursive functions. By exploring various types of Induction methods, we can show that one induction principle supersedes another. In determining the best induction method to use, computational experiments will also be preformed to verify our hypotheses.

1 Introduction

Induction can be very advantageous when verifying the execution of computer programs. Given an input predicate, that meets a restriction, we can prove that a program is correct with respect to such input and output predicates. With such restriction we can also show that the program terminates and at the completion of execution the output predicate is satisfied.

Induction can also be powerful when proving termination of primitive recursive functions. A very popular method to prove is the execution of Euclid's greatest common divisor algorithm. In later section we show the proof of this primitively recursive function.

While computational induction is applied to various areas of Computer Science, it is important to note proving correctness of Distributed Systems and language Compilers. Proving these two subfields of Computer Science, we rely heavily on the use and understanding of induction. In Distributed systems we use induction to verify many things such as protocols and logic of events. While language compilers have proven much harder to verify, proving compiler correctness would show that all programs compiled would be sound and correct. It also becomes increasingly more difficult to prove compilers that have optimization enabled.

2 A proposed problem

Before I describe the various types of Induction methods, I will show a proof of the proposed problem of equality

First we note that

$$\neg E(x, y) \equiv E(x, y) \Rightarrow false$$

$$\begin{aligned}
\forall x, y.(E(x, y) \vee \neg E(x, y)) & \\
\forall x, y.(E(x, y) \vee E(x, y) \Rightarrow false) & \\
\vdash \forall x, y.(E(x, y) \vee E(x, y) \Rightarrow false) \textbf{ by } \lambda \text{ (x. } \lambda \text{ (y. \_\_))} & \qquad (1) \\
x : D, y : D \vdash (E(x, y) \vee E(x, y) \Rightarrow false) \textbf{ by } \text{ ind(x; \_\_; u,i.\_\_)} & \qquad (2)
\end{aligned}$$

== Base Case ==

$$\begin{aligned}
y : D \vdash (E(0, y) \vee E(0, y) \Rightarrow false) \textbf{ by } \text{ ind(y; \_\_; u,i.\_\_)} & \qquad (3) \\
\vdash (E(0, 0) \vee E(0, 0) \Rightarrow false) \textbf{ by } \text{ inl(*)} &
\end{aligned}$$

== Induction on Base Case ==

$$\begin{aligned}
u : D, i : (E(0, u) \vee E(0, u) \Rightarrow false) & \\
\vdash (E(0, s(u)) \vee E(0, s(u)) \Rightarrow false)) \textbf{ by } \text{ inr(\_\_)} & \qquad (4) \\
\vdash (E(0, s(u)) \Rightarrow false) \textbf{ by } \lambda \text{ (a.\_\_))} & \qquad (5) \\
a : E(0, s(u)) \vdash false & \qquad (6)
\end{aligned}$$

Axiom of symmetry

$$a = E(0, s(u)) \Rightarrow E(s(u), 0) by ap(Kleen Axiom 15;\ a) \qquad (7)$$

== Original Induction Step ==

$$\begin{aligned}
u : D, i : (E(u, y) \vee E(u, y) \Rightarrow false) & \\
\vdash (E(s(u), y) \vee E(s(u), y) \Rightarrow false) \textbf{ by } \text{ decide(i;l.\_\_;r.\_\_)} & \qquad (8) \\
& \qquad (9)
\end{aligned}$$

Case 1:

$$l : E(u, y) \vdash (E(s(u), y) \vee E(s(u), y) \Rightarrow false) \qquad (10)$$

Now we have evidence to show two cases from the decide branch. In the case we have evidence E(u,y) or x = y we can see that if we substituted that into the right side of the goal we can obtain 1 = 0. From this we can then transition to say that E(s(u),0) ⇒ false.

$$\begin{aligned}
l : E(u, y) \vdash (E(s(u), y) \vee E(s(u), y) \Rightarrow false) & \\
\textbf{by } \text{inr(ap(Kleen Axiom 15; } l)) & \qquad (11)
\end{aligned}$$

Case 2:

$$r : E(u, y) \Rightarrow \mathit{false} \vdash (E(s(u), y) \vee E(s(u), y) \Rightarrow \mathit{false}) \tag{12}$$

Here we have a clear example that x does not equal y. In the case that x and y are different we can show that E(s(u),u). Using this fact then we can kill the right branch of the decide.

$$\begin{aligned} r : E(u, y) \Rightarrow \mathit{false} \vdash (E(s(u), y) \vee E(s(u), y) \Rightarrow \mathit{false}) \\ \textbf{by } \text{inl(ap(Kleen Axiom 14; } r)) \end{aligned} \tag{13}$$

The following terms where used in this proof
**ind meaning induction*
**ap meaning apply*
**inl meaning in left*
**inr meaning in right*

3 Types of computational induction

There are many types of Computational induction that can be preformed on a simple or complex function. Deciding the type of induction can be a key in formulating a proof for verification or an implementation. By deciding the type of induction you can drastically reduce the computational time needed to calculate an answer. Below we show examples of how to implement the addition function written in the C language. The following functions demonstrate induction used in a recursive and iterative manner.

My hypothesis is that the tail-recursive and iterative functions will have very similar timing results. The double recursive addition function is expected to have the worst timing complexity out of all implementations.

```
int add_rec(int x, int y, int z){
    if (x == 0)
        return z;
    else
        return add_rec(x-1, y, z+1);
}
```

The above function uses Tail recursion to demonstrate addition. What we expect from this function is that the time complexity of this function would be similar to an iterative implementation.

```
int add_slow(int x, int y, int z){
    if(x == 0){
        if(y == 0)
            return z;
        else
            return add_slow(x,y-1,z+1);
    }else
        return add_slow(x-1, y, z+1);
}
```

The reason why this function is called slow is because it preforms induction on the x variable then induction on the y variable. As we decrement both variables we increment the z variable. This type of addition only works if the variable $z = 0$. The main point of this implementation is to show how computationally inefficient this function preforms.

```
int add_while(int x, int y, int z){
    while( x != 0 ) {
        z = z + 1;
        x = x - 1;
    }
    return z;
}
```

This function is the iterative version of the addition function which is similar to the tail recursive implication add_rec.

```
int add_for(int x, int y, int z){
    for(; x != 0; z++, x--);
    return z;
}
```

Above is another implementation of the iterative addition function. We can see using a different loop structure such as the for-loop, we simplify the implementation to two simple lines of code.

By setting the variables $x = 92847, y = 63820$ we can invoke the all the functions with the input (x,y,y). The only function we call differently is the add_slow, with the input (x,y,0). Blow we see computational timing results of the functions.

- Tail-Recursive Addition 92847 + 63820 = 156667
  Time: 2.774ms
- Double Recursive Addition 92847 + 63820 = 156667
  Time: 4.16ms
- While-loop Addition 92847 + 63820 = 156667
  Time: 0.356ms
- For-loop Addition 92847 + 63820 = 156667
  Time: 0.282ms

As we can see the worst preforming implementation is the add_slow function. While the function still uses tail recursion it still preforms slow. However, when we look at the tail revulsive function and compare the results with the for and while loop implementation we notice a huge difference in timing results. As we expect the recursive function preformed better then the double revulsive version but worst then the two iterative methods. In general we can conclude that recursion is slow except for when it is used properly. The above example shows the function being used properly but still costly in terms of time. Telling the compiler that we would like to use a loop rather then recursion helps in terms of speed up.

Another thing to notice from the results is that out of approximately forty trial runs, the for-loop preforms the best. By understanding how the compiler works, we can show that a simple on line loop helps the compiler in restructuring the assembly code for faster execution.

By using any type of iterative induction methods, we can early show that the loop terminates and therefor the program produces the output that we expected. In the for-loop example we can see the variable x being decremented until it eventually reaches zero. The variable only decrements and does not get set in the body of the function, therefore we can conclude that x will eventually equal zero. Using the same method we can show that the while-loop implementation also terminates.

It is also important to note that when we compile the code we do not use any type of compiler optimization. Therefore we do not expect the compiler to preform any extra work in understanding the function and restructuring the assembly code. If we compile the functions using the following

```
gcc -O3 add.c -o add
```

We obtain the following timing results.

- Recursive Addition 92847 + 63820 = 156667
  Time: 0.046ms
- While-loop Addition 92847 + 63820 = 156667
  Time: 0.001ms
- Double Recursive Addition 92847 + 63820 = 156667
  Time: 0.079ms
- For-loop Addition 92847 + 63820 = 156667
  Time: 0ms

We see that the loop functions take about the same exact time. The recursive functions only vary by one hundredth of a millisecond.

Formally we can prove by induction the tail-recursive method. The proof uses evidence semantics.

$$\forall x, y.\exists z.Add(x, y, z) \text{ such that z = x + y}$$

$\vdash \forall x, y.\exists z.Add(x, y, z)$by $\lambda(x.\lambda(y.$__)
$x : D, y : D, \vdash \forall x, y.\exists z.Add(x, y, z)$ by ind(x;__; u,i.__)
Base Case
$\vdash \exists z.Add(0, y, y)$ by ap(Kleene Axiom 18; y)
$u : D, i : \exists z.Add(u, y, z) \vdash \exists z.Add(s(u), y, z)$ by spread(i; z, a,__)
$z : D, a : Add(u, y, z) \vdash \exists z.Add(s(u), y, z)$ by pair (s(z), __)
$\vdash Add(s(u), y, s(z))$ by ap(Kleene Axiom 19; Add(s(u), y, s(z))

Q.E.D

When proving the tail-recursive addition function, weak induction was used. There are two types of induction that could have been used to prove this function, the first being weak induction, and the second is strong induction. Weak induction has the form

$$(A(0)\ \&\ \forall A(x) \Rightarrow A(s(x))) \Rightarrow A(x)$$

Strong induction, also known as complete induction, takes the form

$$\forall x.(\forall y.(y < x \Rightarrow P(y) \Rightarrow P(x))) \Rightarrow \forall x.P(x)$$

4 Fixpoint Induction and the Y combinator

To demonstrate a proof in fixed point theory, the McCarthy 91 function becomes very handy. The function simply states

$$F(n) = \begin{cases} n - 10 & \text{if } n > 100 \\ F(F(n + 11)) & \text{if } n \leq 100 \end{cases}$$

To demonstrate computational timing results, I implemented the function using the recursive definition and iterative form. The functions as implemented in C language.

```
int McCarthy(int n){
    if(n > 100)
        return n - 10;
    else
        return McCarthy(McCarthy (n+11));
}
}
```

The function uses tail-recursion for optimization. Below we show the iterative version of the function.

```
int itr_McCarthy_91(int n){
    int c;
    for (c = 1; c != 0; ) {
        if (n > 100) {
            n = n - 10;
            c--;
        } else {
            n = n + 11;
            c++;
        }
    }
    return n;
}
```

Below are the timing results of both functions.

- Recursive McCarthy 91 (n = -18760)
  Time: 0.309ms
- Iterative McCarthy 91 (n = -18760)
  Time: 0.106ms

As expected, the iterative function preformed tree times faster then the Tail-recursive function. As stated before, in general recursion is slow. It should be avoided if possible. The only need to use recursion is to reduce the complexity of the functions.

The termination of the iterative method is very simple. We can see that the "c" variable needs to equal zero to break the conditional statement of the for-loop. Depending on the input we can see that when n is above 100 the variable will keep decrementing until it equals zero. Therefore, the loop terminates.

The recursive function is a bit more difficult to prove. The example below uses the input 99 to demonstrate fix point F using the innermost-leftmost reduction rule.

- F(99) → F(F(110)) → F(F(100)) → F(F(111)) → 91

To show the power of fix point induction, consider the following recursive programs

$$P_1 : F(x) \leftarrow \text{ if } x > 10 \text{ then x - 10 else } F(F(x+13))$$

$$P_2 : F(x) \leftarrow \text{ if } x > 10 \text{ then x - 10 else } F(F(x+3))$$

We would like to show that

$$f_{P1}(x) \equiv f_{P2}(x) \text{ for all } x \in \mathbb{N}$$

By substituting F(x+13) in P_1 by its application we can obtain

$P_{new} : F(x) \rightarrow ifx > 10$ then $x{-}10$ else $F(ifx{+}13 > 10$ then $x{+}13$ else $F(F(x{+}13{+}13)))$

Since x $\geq$, its always the case that 13 ¿ 10; therefore P_{new} can be simplified to F(x+3).

$$\text{Thus}(\forall x \in \mathbb{N}) f_{p1}(x) \equiv f_{p2}(x)$$

Q.E.D

5 Program verification

In proving the correctness of programs we should worry about what type of termination we expect. In partial correctness we "don't care" about termination, but in total correctness termination is essential. To Verify a program for a given input predicate P(x) and an output predicate Q(x) means to show total correctness with respect to P(x) and Q(x). To establish this proof we show it in two steps; first we prove partial correctness, and then we prove termination of Q(x).

To formally verify a program we use Euclid's greatest common divisor algorithm.

$$\forall y_1 : \mathbb{N}.\forall y_2 : \mathbb{N}^+ . y_1 = y_2 * q + r \ \& \ r < y_2$$

This proof can show that the algorithm is partially correct. For the algorithm the computation method is based on the fact that

$$\text{If } y_1 > y_2 \text{ then } \gcd(y_1, y_2) = \gcd(y_1 - y_2, y_1)$$

$$\text{If } y_1 < y_2 \text{ then } \gcd(y_1, y_2) = \gcd(y_1, y_2 - y_1)$$

$$\text{If } y_1 = y_2 \text{ then } \gcd(y_1, y_2) = y_1 = y_2$$

```
while $y_1 \neq y_2$ do
    if $y_1 > y_2$ then $y_1 = y_1 - y_2$ else $y_2 = y_2 - y_1$;
```

Using the above computational rule allows us to build the following program that we can show is partially correct. The simplest way to show program termination is to substitute y_1 with y_2 or vise versa. Any order of substitution shows that y_1 is equal to y_2. The negation of the conditional statement to the while loop is $\neg(y_1 \neq y_2)$. Therefore we can show that the while loop will eventually terminate. The above code realize on the inductive expression principle that was introduced by Hoare. This principle is also known as Hoare's while rule.

$$\frac{\{P \wedge B\} S \{P\}}{\{P\} \textbf{ while } B \textbf{ do } S \textbf{ done } \{\neg B \wedge P\}}$$

Something to note in the above program is that we use a while loop rather then other forms of induction. Using the while loop allows us to adapt Hoare's while rule and formalize the proof. Deciding the type of induction to use helped guide the proof to an easier solution.

6 Conclusion

In general we can not conclude that there is a general induction method to use to solve all problems. The Induction method to use highly depends on the problem domain. Correctly analyzing the problem and determining the type of induction to use can increase readability and simplicity of the proof. In terms of computational results, deciding the right induction method to use can help improve the time complexity of the program. The right induction principle to use can also help verify the correctness of a given program. Based on the computational experiments with the addition and McCarthy 91 functions, we can see that in both cases the iterative implementation preformed better. We can conclude that recursion is slow and should be avoided, but that comes at the cost of simplicity. Recursion can help simplify the readability of algorithms at the cost of execution time while, iterative methods help improve execution time at the cost of readability.