Computer Science 52

Induction, Recursion, and O-notation

Fall Semester, 2014

1 Proof by induction

You have seen (somewhere, no doubt) the principle of induction. It has many forms. One way to state it is as follows:

- [Base Case(s)] *Verify* that some property P holds for the simplest objects.
- [Induction Step(s)] *Assume* that the property P holds for all objects simpler than X, and from that assumption *prove* that P holds for X.
- [Conclusion] *Conclude* that the property P holds for all objects.

This principle holds for many classes of objects. Examples are

- the natural numbers, in which "simplest" means zero and "simpler" means less than;
- lists, in which "simplest" means empty and "simpler" means shorter, and
- finite trees, in which "simplest" again means empty and "simpler" means subtree.

There are also kinds of objects for which the principle of induction fails. One example is the set of subsets of natural numbers. Take "simplest" to mean the empty set and "simpler" to mean proper subset. One can prove that the empty set is finite, and if every proper subset of a set S is finite, then S is finite. But it would be incorrect to conclude that *every* set of natural numbers is finite.

Induction is possible when there is no infinite sequence of successively "simpler" objects. For the example in the previous paragraph, one can construct an infinite decreasing chain of proper subsets of the natural numbers, so inductive proofs are not possible. On the other hand, induction is possible over the collection of *finite* subsets of the natural numbers.

An easy example of a proof by induction is verifying that each natural number is even or odd. We use ordinary induction over the natural numbers.

- [Base case] Observe that 0 is even.

- [Inductive step] Given a natural number k, assume that every number less than k is either even or odd. (This is the *hypothesis of induction.*) Then $k-1$ is either even or odd. If $k-1$ is even, then k is odd. If $k-1$ is odd, then k is even. Therefore k is either even or odd.
- [Conclusion] By virtue of the above arguments, any natural number is either even or odd.

2 Recursive functions

Recursive functions can be defined in a pattern similar to inductive proofs:

- [Base case(s)] *Define* the value of a function F on each of the simplest objects.
- [Recursive step(s)] *Assume* that you know the value of F on every object that is simpler than X, and use (some of) those values to *define* the value of F on X.
- [Consequence] *Conclude* that F is a function defined on all objects.

The possible sets of candidate "objects" are the same as for proof by induction, and one uses induction to verify facts about recursively-defined functions.

Notice that our consequence can be justified by an appeal to the principle of induction. Recursion and induction are closely related, but they are used for different purposes. Often we *define* a concept by recursion and then *prove* something about it by induction. It is natural that the structure of the proof follow the definition, but it is important to keep the two ideas separate—otherwise the argument appears circular. One way to think about it is that recursive definitions are justified by the principle of induction.

When thinking about functions on natural numbers, the definition presented here is really for a special class called the *primitive* recursive functions. A later section of these notes describes the more general case.

3 Examples of Induction and Recursion

For our first example, consider the append operation on lists which is defined using list recursion:

`nil @ yt = yt`, and

`(x::xs) @ yt = x::(xs @ yt)`.

The recursion occurs on the first of the two arguments to the append operation. Notice how closely the definition above follows the formulation in SML.

Using the clauses of the definition, we can prove that `xt @ nil` is `xt`:

- [Base case] When `xt` is `nil`, then we know that `nil @ nil` is `nil`. This is just a special case of the first clause of the definition of the append operation.
- [Induction step] Given a list `xt` of the form `x::xs`, assume that `xs @ nil` is `xs`. Then compute as follows:

  ```
  (x::xs) @ nil = x::(xs @ nil) = x::xs
  ```

  The last step is the application of the hypothesis of induction.
- [Conclusion] These steps prove that `xt @ nil` is `xt` for all lists `xt`.

Most of the time, people write inductive proofs a bit more informally and omit the explicit conclusion.

One absolutely critical observation is to see where the hypothesis of induction is used. If you cannot find its use, then you probably do not have a correct proof.

Sometimes, beginners are uncomfortable with proofs by induction, because each part of the proof appears so obvious—but it is not at all clear exactly which obvious facts to write down or in what order they should be written. In the inductive step, remember that you do not have to *prove* the inductive hypothesis; it is an assumption. But do remember to *use* the inductive hypothesis, because it carries most of the information. In many proofs, the other reasoning is simply a minor maneuver to get to a place where one can use the inductive hypothesis.

In our second example, we can assign a height to every tree—and hence to every node in a tree, since a node is effectively a subtree.

- The height of an empty tree is -1.
- The height of a non-empty tree is one greater than the maximum of the heights of its subtrees.

Although we have not explicitly stated it, height is a recursively-defined function on trees. The principle of induction tells us that every tree has a height. With the notion of height, we can state and prove a bound on the number of leaves and nodes in a tree.

Proposition. A tree of height h has at most 2^h leaves and $2^{h+1} - 1$ nodes.

The proof is by induction on trees. Base case: The empty tree has height is -1. It has zero leaves and zero nodes, and $0 \leq 2^{-1}$ and $0 \leq 2^{-1+1} - 1$.

Induction step: Let T be a tree and assume that the proposition holds for all subtrees of T. Let L be the left subtree, whose height is h_L, whose leaves number m_L, and whose nodes number n_L. Our inductive hypothesis tells us that

$$m_L \leq 2^{h_L} \text{ and } n_L \leq 2h_L + 1 - 1.$$

We have an analogous collection of facts about the right subtree R of height h_R, with m_R and n_R leaves and nodes respectively.

Now, by the definition of height, the height of T is $h = 1 + \max(h_L, h_R)$. We can compute the number of leaves in T:

$$m = m_L + m_R \leq 2^{h_L} + 2^{h_R} \leq 2 \cdot 2^{\max(h_L,h_R)} = 2^{1+\max(h_L,h_R)} = 2^h$$

Take some time to verify each step as you read the equality and inequality symbols from left to right. Note the essential use of the induction hypotheses about m_L and m_R.

Next, compute the number of nodes. There is the root, all the nodes in the left subtree, and all the nodes in the right subtree.

$$n = 1 + n_L + n_R \leq 1 + \left(2^{h_L+1} - 1\right) + \left(2^{h_R+1} - 1\right)$$

Again, verify each step and identify the uses of the induction hypothesis. This completes the induction step, and hence the proof.

4 Complexity analyses

Consider the following natural list functions:

```
fun member e nil     = false
  | member e (x::xs) = e = x orelse member e xs;

fun uniquify2 nil     = nil
  | uniquify2 (x::xs) = if member x (uniquify2 xs)
                        then uniquify2 xs
                        else x::(uniquify2 xs);
```

Let $RU2(k)$ be the number of *recursive* calls (not counting the first) to `uniquify2` when the function is applied to a list of length k.

$$RU2(k) = \begin{cases} 0 & \text{if } k = 0, \text{ and} \\ 2 + 2 \cdot RU2(k-1) & \text{otherwise.} \end{cases}$$

As we have seen, we can prove facts about a recursively-defined function using induction. One example is that the domain of *RU2* is the set of all natural numbers. That exercise is left for the reader.

In another example, we can write the function in closed form and prove by induction that

$$RU2(k) = 2^{k+1} - 2.$$

Notice from the recursive definition that $RU2(0) = 0$ and also that $2^{0+1} - 2 = 0$. That establishes the base case. For the inductive step, assume the induction hypothesis:

For $j < k$, $RU2(j) = 2^{j+1} - 2$.

In particular, when $j = k - 1$, we have $RU2(k-1) = 2^k - 2$. Then we can compute using the definition.

$$\begin{aligned} RU2(k) &= 2 + 2 \cdot RU2(k-1) && \text{from the definition} \\ &= 2 + 2 \cdot \left(2^k - 2\right) && \text{by the induction hypothesis} \\ &= 2^{k+1} - 2 && \text{by algebra} \end{aligned}$$

We say that the function *RU2* is *exponential,* meaning that it has a growth rate very much like 2^k. Such functions get large very quickly as the argument k increases. For example, the value of $RU2(30)$ is over one billion. One consequence of the exponential growth rate of *RU2* is that the function `uniquify2` makes exponentially many recursive calls and will take a long time to compute. Shortly, we will examine a more efficient formulation of the same function.

5 O-notation

We want to clarify the phrase that "*RU2* has a growth rate very much like 2^k." Suppose that f and g are functions from natural numbers to natural numbers. We say that f is $O(g)$, read "f is big-oh of g" or "f is order-of g," if there are numbers c and k_0 satisfying

$$f(k) \leq c\, g(k) \text{ whenever } k_0 \leq k.$$

The idea is that f grows no faster than g.

We use the O-notation to compare functions that represent the running time of computer programs. We allow the multiplicative constant c because, among other reasons, you might get a faster computer. We ignore the first few values, those less than k_0, because they might unduly reflect the startup costs of a computation and be misleading. Of course, these simplifications can be misused: The constant c can be huge, and the bound k_0 could be larger than all the problems we really care about.

While it is true, the fact that $3k$ is $O(2^k)$ is not very interesting. We usually want our O-notation bounds to be "close." Further, it is important to remember that there is only one constant c and one bound k_0 in the definition. Those values cannot depend on k.

The number of recursive calls for `uniquify2` is $O(2^k)$, computed above, as can be verified by taking $c = 2$ and $k_0 = 0$. (Take a moment to write it out.)

A function for the actual running time for `uniquify2` must take into account the call to `member` as well as the recursive calls. Let A be the amount of time to return a value; B the amount of time for the `if` construction; Mk the amount of time, in the worst case, for the calls to `member`; and $EU2(k)$ be the total running time when `uniquify2` is applied to a list of length of length k. Then

$$EU2(k) = \begin{cases} A & \text{if } k = 0, \text{ and} \\ A + B + Mk + 2 \cdot EU2(k-1) & \text{otherwise.} \end{cases}$$

It is difficult to express the function in closed form, but it is easy to show that it is $O(3^k)$. Take c to be $A + B + M$ and n_0 to be 3, and prove by induction that

$$EU2(k) \leq (A + B + M)3^k \text{ whenever } 3 \leq k.$$

(Again, write it out for yourself.) One of the reasons for using the O-notation is to avoid unnecessary details of expressions like the one above involving A, B, and M.

Consider another function `uniquify1`.

```
fun uniquify1 nil     = nil
  | uniquify1 (x::xs) = if member x xs
                        then uniquify1 xs
                        else x::(uniquify1 xs);
```

A call to a function like `uniquify1` has three parts:

- a constant amount of overhead, involving the `if` and returning a value;
- an $O(n)$ amount of time for the call to `member`; and
- the time for the recursive call.

We can reason that there are n recursive calls, each taking $O(n)$ time, so the total running time is $O(n^2)$. (Here is a third opportunity to fill in the details!)

Still another version of *uniquify* appears below:

```
fun uniquify3 nil     = nil
  | uniquify3 (x::xs) =
    let
        val recResult = uniquify3 xs
    in
        if member x recResult
           then recResult
           else x::recResult
    end;
```

This function will *often* be faster than `uniquify1`, but in the worst case it the same as `uniquify1`.

From a mathematical point of view, there is only one `uniquify` function, but there are many ways to compute it. We have seen that some implementations are much faster than others.

6 Further examples with list induction

This section contains some facts about list functions, most of which are proved by induction. Let us begin by collecting a number of facts about the append operation.

1. `nil @ vt = vt`

   The first part of the recursive definition of append.

2. `(u::us) @ vt = u::(us @ vt)`

   The second part of the definition.

3. `ut @ nil = ut`

   Proved above.

4. `[u] @ vt = u::vt`

   No induction is needed for this result. Simply remember that `[u]` is an abbreviation for `u::nil` and calculate as follows, using the definition of append:

   ```
   [u] @ vt = (u::nil) @ vt
            = u::(nil @ vt)
            = u::vt
   ```

5. `(ut @ vt) @ wt = ut @ (vt @ wt)`

   This is proved by list induction on `ut`. When `ut` is `nil`, we have

   ```
   (nil @ vt) @ wt = vt @ wt = nil @ (vt @ wt)
   ```

   Both steps are applications of Fact 1. For the induction step, assume that `(us @ vt) @ wt = us @ (vt @ wt)` and calculate

   ```
   ((u::us) @ vt) @ wt = (u::(us @ vt)) @ wt
                       = u::((us @ vt) @ wt)
                       = u::(us @ (vt @ wt))
                       = (u::us) @ (vt @ wt)
   ```

   We must justify each step. The first, second, and last steps are justified by the definition of the append operation. The third step is an application of the induction hypothesis.

Now we can use the facts about the append operation to establish some results about our two versions of the reverse function. Here are the definitions.

```
nrev nil     = nil
nrev (u::us) = (nrev us) @ [u]

revApp nil     vt = vt
revApp (u::us) vt = revApp us (u::vt)

arev ut = revApp ut nil
```

6. `revApp ut vt = (nrev ut) @ vt`

   This key fact is proved by induction on `ut`. It's clear when `ut` is `nil`, because both sides evaluate to `vt`. Assume the result is true for `us` and prove for `u::us`.

   ```
   revApp (u::us) vt = revApp us (u::vt)
                     = (nrev us) @ (u::vt)
                     = (nrev us) @ ([u] @ vt)
                     = ((nrev us) @ [u]) @ vt
                     = (nrev (u::us)) @ vt
   ```

   Test yourself by justifying each step above. Remember to identify the use of the induction hypothesis.

7. `arev ut = nrev ut`

   This shows that both our versions of the reverse function do the same thing. (They may be both correct or both wrong, but at least we know they are the same.) The proof is a simple application of the previous fact; no induction is involved.

   ```
   arev ut = revApp ut nil
           = (nrev ut) @ nil
           = nrev ut
   ```

We can now prove some more interesting results about the reverse function. The first is a simple calculation.

8. `nrev [u] = (nrev nil) @ [u] = nil @ [u] = [u]`

9. `nrev (ut @ vt) = (nrev vt) @ (nrev ut)`

   The proof is by list induction on `ut`. It is easy when `ut` is `nil`. For the induction step, assume the result for `us` and calculate as follows.

   ```
   nrev ((u::us) @ vt)
        = nrev (u::(us @ vt))
   ```

```
= (nrev (us @ vt)) @ [u]
= ((nrev vt) @ (nrev us)) @ [u]
= (nrev vt) @ ((nrev us) @ [u])
= (nrev vt) @ (nrev (u::us))
```

10. `nrev (nrev ut) = ut`

    Again, use list induction. It is immediate for `nil`. Assume the result for `us` and compute:

    ```
    nrev (nrev (u::us))
         = nrev ((nrev us) @ [u])
         = (nrev [u]) @ (nrev (nrev us))
         = [u] @ us
         = u::us
    ```

    Be sure that you can identify the use of the induction hypothesis.

All these facts about list-reversal functions are, perhaps, obvious. The value of the examples is that they illustrate the fundamental principle of induction and the close relation between recursion and induction.

As a final example, we investigate the relationship between `foldr` and `foldl`. Use the definitions given in class for `foldr` and `foldl`. To simplify the notation, write `R` and `L`, respectively, for `foldr f` and `foldl f`.

11. `L (R b (ut @ vt)) wt = L (R b vt) ((rev ut) @ wt)`

    As usual, use list induction on `ut`. The result is immediate when `ut` is `nil`. Assume that the result holds for `us` and calculate.

    ```
    L (R b ((u::us) @ vt)) wt
        = L (f(u,R b (us @ vt))) wt
        = L (R b (us @ vt)) (u::wt)
        = L (R b vt) ((rev us) @ (u::wt))
        = L (R b vt) ((rev (u::us)) @ wt)
    ```

In the fall of 2003, CS 52 student Daniel Kleinman conjectured a relationship between `foldr` and `foldl`.[1] Namely, `foldr f b` applied to a list yields the same result as `foldl f b` applied to the reverse of the list. We can use item 11 to prove the conjecture.

12. `foldr f b ut = foldl f b (rev ut)`

    With both `vt` and `wt` being `nil` in 11, we get

    ```
    foldr f b ut = R b ut
                 = L (R b ut) nil
    ```

[1] It should be noted, however, that Daniel denies responsibility for his conjecture.

```
= L (R b nil) (rev ut)
= L b (rev ut)
= foldl f b (rev ut)
```

7 A curiosity

What is wrong with the following argument that shows all horses have the same color?

We show, by induction on finite sets, that every finite set of horses have the same color. For the base case, just observe that all the horses in the empty set have the same color. Alternatively, start the induction with sets of size one and note that all the horses in a set of size one have the same color.

Now assume that all sets of horses of size less than k have the same color. Consider a set of size k: $\{h_1, h_2, \ldots, h_k\}$. By the hypothesis of induction, all the horses in the smaller set $H_R = \{h_2, \ldots, h_k\}$ have the same color. Also, all the horses in another smaller set, $H_L = \{h_1, \ldots, h_{k-1}\}$ have the same color. Now, horse h_1 has the same color as all the horses in the intersection $\{h_2, \ldots, h_{k-1}\}$ of our two smaller sets, and horse h_k also has the same color as the horses in the intersection.

We conclude that horse h_1, horse h_k, and all the horses in the intersection have the same color. These are all the horses in our original set $\{h_1, h_2, \ldots, h_k\}$, and they all have the same color. Therefore, every finite set of horses has the same color.

8 General Recursion

When we write programs we have broad freedom in the kinds of recursive functions we can construct. Any self-referential definition is legitimate in the sense that it can be turned into executable code. Consider the function e on natural numbers defined by

$$e(n) = e(n) + 1.$$

One could easily declare its SML counterpart and (try to) compute the value $e(47)$. There is nothing wrong with our recursively-defined function e, it just happens that it is not defined on the whole set of natural numbers. In fact, it is defined nowhere.

Functions which are defined on a subset of their natural domain are called *partial functions.* As the example above shows, partial functions are inescapable.

The primitive recursive techniques discussed earlier yield functions which *are* defined everywhere, as one can prove using induction. Such functions are called *total functions.*

A less trivial example is provided by the Takeuchi function from Assignment 1. It is a total function, defined on all triples of natural numbers, as one can prove by induction. (Try it!)

Another example is a famous open problem. The "three-n plus one" function, also known as the Collatz function, is defined on natural numbers:

$$t(n) = \begin{cases} 1 & \text{if } n \leq 1, \\ t(n/2) & \text{if } 1 < n \text{ and } n \text{ is even, and} \\ t(3n+1) & \text{otherwise.} \end{cases}$$

Clearly, the only possible value for the function is 1. But does t have a value for *every* argument? No one has yet been able to prove that t is a total function.

9 Exercises

Here are a few problems for practice. Use the following definitions, and refer to them by number.

1a. `nil @ vl = vl`

1b. `(u::us) @ vl = u::(us @ vl)`

2a. `len nil = 0`

2b. `len (x::xs) = 1 + (len xs)`

3a. `map f nil = nil`

3b. `map f (x::xs) = (f x) :: (map f xs)`

4a. `cart nil vl = nil`

4b. `car (u::us) vl = (map (fn y=>(u,y)) vl) @ (car us vl)`

1. Prove that `len(ul @ vl)` = `len(ul)` + `len(vl)`.

2. Prove that `len(map f ul)` = `len(ul)`.

3. Prove that `map f (ul @ vl) = (map f ul) @ (map f vl)`.

4. Use the previous results to prove that `len(cart ul vl)` = `len(ul)` × `len(vl)`.

5. Prove that $2^0 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$ for $0 \leq n$.

6. We have been using this variant of the Fibonacci function.

$$f(n) = \begin{cases} 1 & \text{if } n \leq 1, \text{ and} \\ f(n-2) + f(n-1) & \text{otherwise} \end{cases}$$

Use it to prove, for $2 \leq n$, that $f(0) + f(1) + f(2) + \ldots + f(n-1) = f(n+1) - 1$.

7. Does the result of Exercise 6 still hold if we start the Fibonacci sequence at zero?

$$f(n) = \begin{cases} 0 & \text{if } n \leq 0, \\ 1 & \text{if } n = 1, \text{ and} \\ f(n-2) + f(n-1) & \text{otherwise} \end{cases}$$