Class - CSc 335

Date - Feb 23, 2023 (Thursday)

Advice for quiz

- Read the questions carefully

Iterative VS Recursive

- For an iterative procedure, it will always (?) have an accumulator, which I tend to call somethig like `result-so-far`.
- For recursive procedure, it never (?) has accumulative.

Recursive solution for the sum-of-digit problem

Proposed Divide & Conquer:

- linear splitting of the `input n` into `(quotient n 10)` and `(modulo n 10)`, then adding → This may suffice as background for the coding effort
  - Basis case is that which admits no further decomposition.
    - i.e. For which futther decomposition is unnecessary.
  - What is the stopping step? Is it `(< n 10)` or is it `(zero? n)`
    - In both cases, we would return `n`.

```
(define (digit-sum n)
  (cond ((< n 10) n)
        (else (+ (digit-sum (quotient n 10)) (modulo n 10)))))
```

- pre-condition: `n >= 0` is an integer

Checking/Testing

- Does the precondition hold the ahead of the recursive call?
- We only need to check that

  ```
  n >= 0 an integer -> (quotient n 10) >= 0 an integer
  ```

  so - by the IH, the recursive call returns the sum of all the digits except the right most digit of n and now the job of the IS is to show that the program does "the right thing with the result".
  - where `IH = Induction hypothesis` and `IS = Induction steps`

Comparision with Iterative solution

- Compare the simplicity of this argument to the one developed for an iterative solution to the same problem.

- Where does the simplicity come from or what is the source of this simplicity?
  - Primarily, it stems for the recursive program not needing to worry about the "intermediate points in the execution of the program".
  - There is no invariant in recursive!!!

From class notes

Recursive Solution to b^n problem

- `n >= 0` an integer and `b` a number
  - Recursive call to compute b^{n-1}
  - multiply this result by b

Done??? or given our recent experience, perhaps we should worry about the basis before going to code...

- We plan to induct on `n` (non-negative integer), so presumably `n = 0` is the stopping case. But now, we notice that `b = 0` is a possibility, and also that $\mathbf{0^0}$ (zero over zero) is undefined.

So: The specification is broken and how to repair??

- We might
  1. insist that `b != 0`
  2. insist that `n != 0`
  3. insist that b and n are not both **ZERO**

Aside on pre-conditions, which helps us see that option 3 is the best choice among these?

- Option 3 is the weakest among these.

In thise case, a model is a (b, n) pair.

- Every model of 1 is a model of 3 but there are models of 3 which are not models of 1.

Solution where `b != 0`

- what can we do with this to solve the entire problem? What about using a wrapper? (or a `cond`?)
- Assuming b and n are not both zero
  - when b is zero, we can just return zero.
  - when b is not zero, compute b^n by recursion on n
    - in these two cases, `n` might be zero, but since `b is not equal` this is not a problem.

What might be a functional decomposition?

- Check of b is equal to zero
  - if yes, compute zero^n for `n > 0`
  - if no, compute b^n by recursion on n

Back to the problem of adding two numbers `a` and `b`, where both `a and b >= 0` are integers

- One design idea for an iterative solution might be to decrement `a` and increment `b` until `a = 0`, at which point returns `b`
    - So: we will try to use `b` as the accumulator
- Roughly:

```
(define (add a b)
  (cond ((zero? a) b)
        (else (add (- a 1) (+ b 1)))))
```

What is a useful invariant?

- The idea is that `a + b` is maintained equal to the sum of the original `a` and original `b`, when `a = 0`, this implies `b = sum of original a and original b`.

Rather than write out `original a` and `original b`, let's introduce the convention of `ghost variables` (concept due to John Reynolds)

- we use `A` to abbreviate `original a`
- we use `B` to denote `original b`

With this convention, our GI is just `A + B = a + b`

- where a and b are computed variables

A and **B** never change and do not occure in the code (hence `ghost`)

- So far our invariants have had the form

```
Total Work = Work Done + Work Remaining
```

but this one appears to be different (?)

```
A + B = b + a
```

so........no, not different

```
Total Work is A + B
Work Done is b
Work Remaining is a
```

- Notice that the ghost variable idea is similar to the `ghost function` (`sum-digits-in`) function from office hour on Feb 22 Page 6