CSc 335 Class 12 March 7 2024

Quiz 2

(full period)

CSc 335 Class 13 March 12 2024

Towards Higher Order Procedures

; pre: a and b are integers

; post: returns $\sum_{i=a}^{b} i$ (sum of all i, $a \le i \le b$)

recall: by convention, this is 0 if $a > b$.

```
; Consider the summation procedure sigma:

(define (sigma a b)
  (cond ((> a b) 0)
        (else (+ a (sigma (+ a 1) b)))))
```

; this is a recursive program, but the induction is a bit different from what we have seen so far
; in that we will induct on the size of the gap between a and b:

```
;  ---------------------------------------------------
;  |          a                              b       |
;  ---------------------------------------------------
;                | <--------gap(a,b)--------->|
```

so if a=3 and b=5, the sum is 0. to

why? suppose we're to say "proof by induction on a". Problem is that a is increasing and so the IH would be for a+1

And this does not fit the customary inductive framework.

provided a<b, the gap between a+1 and b is smaller than the gap between a and b. So this METRIC is suitable for an induction.

2-case basis step? to make the gap induction work?

Your documentation should include the definition you have in mind for gap(a,b)

Suggested HW

would ((>= a b) 0) be better?

Well, NO: when a=b, ans. needs to be a.

Code for this design is easily seen to be

```
(define (sigma a b)
  (cond ((> a b) 0)
        (else (+ a (sigma (+ a 1) b)))))
```

Might we instead keep a fixed and induct on b?

Work out the details!

Work out the program!

Higher-order functions — some motivation

We've discussed:

Computes $\sum_{i=a}^{b} i$

```
(define (sigma a b)
  (cond ((> a b) 0)
        (else (+ a (sigma (+ a 1) b)))))
```

and one notices that this pattern occurs frequently. For example, we can compute $\sum_{i=a}^{b} i^2$

using

```
(define (sum-of-squares a b)
  (cond ((> a b) 0)
        (else (+ (square a) (sum-of-squares (+ a 1) b)))))
```

Also -

```
(define (sum-of-cubes a b)
  (cond ((> a b) 0)
        (else (+ (cube a) (sum-of-cubes (+ a 1) b)))))
```

When a functional programmer detects a pattern like this, she/he asks whether one program might be made to do the work of all three.

In fact - this is easy to do: one introduces a function parameter and calls the new function with values (lambda (x) x), square, and cube for that parameter

```
(define (generalized-sigma a term b)
  (cond ((> a b) 0)
        (else (+ (term a)
                 (generalized-sigma (+ a 1)
                                    term
                                    b)))))
```

Now we obtain sigma using generalized-sigma

```
(define (sigma a b)
  (generalized-sigma a (lambda (x) x) b))
```

and sum-of-squares

```
(define (sum-of-squares a b)
  (generalized-sigma a square b))
```

and sum-of-cubes

```
(define (sum-of-cubes a b)
  (generalized-sigma a cube b))
```

This is one nice way of achieving software reuse — here, for example, only generalized-sigma would need to be stored in a function library, rather than sigma, sum-of-squares and sum-of-cubes

separately.

generalized-sigma is said to abstract the functions sigma, sum-of-squares, and sum-of-cubes.

Is this all there is to do? Does introducing this new variable have any impact on the proofs we write? Looking at the code, you can see that it does - we need to check the interface between term and generalized-sigma.

```
(define (generalized-sigma a term b)
  (cond ((> a b) 0)
        (else (+ (term a)
                 (generalized-sigma (+ a 1)
                                    term
                                    b)))))
```

You can see that term needs to be a function from $[a, b] \longrightarrow$ Numbers - otherwise makes no sense.

Moreover, you can see that (term i) needs to terminate for each i in $[a, b]$ - for otherwise we cannot make any promises about the termination of generalized-sigma.

So we should perhaps look at how one would write a proof for a function with a function parameter.

Just as we want our function library to contain just the most general version of sigma, so also would we like to write just one proof. In particular, we do not want to be forced to write a proof for each possible value of term.

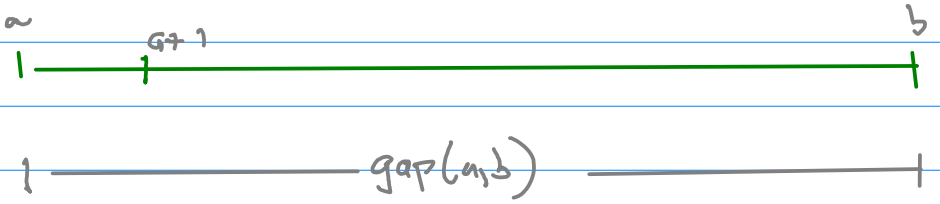
(Can you see other values of term which would be acceptable? fourth-power? Maybe sqrt? Maybe---- our catching that pattern is very much more powerful than just generalizing the functions we started with)

Can you see what the qualifications for term values must be? These qualifications are precisely the specification for term: if the proof we file is to be as useful as the function, then the pf should not refer at all to particular choices for term — eg - not to square or to cube — but just to the specification of term. In other words, we want the proof to be abstract in the same sense that the code is abstract.

Let's sketch a proof for generalized-sigma

```
(define (generalized-sigma a term b)
  (cond ((> a b) 0)
        (else (+ (term a)
                 (generalized-sigma (+ a 1)
                                    term
                                    b)))))
```

As for sigma (see earlier notes), we can organize the proof as an induction on gap(a,b)



where $gap(a,b) = \begin{cases} 0 & \text{if } a > b \\ b-a & \text{otherwise} \end{cases}$

We focus just on the IH:

'The recursive call works correctly'

ie

(given the pre-conditions)

(generalized-sigma (+ a 1) term b)

correctly computes

$$\sum_{i=a+1}^{b} term(i)$$

For this to make sense, term must be compatible with + (ie, Σ) and the values term(i) must all be defined (ie, (term i) must terminate.

These are the ONLY requirements we impose on term.

The spec for generalized-sigma has — in addition to the information contained in the spec for sigma — requirements for term

pre: $a \le b$ and a, b are integers
and
for each $a \le i \le b$, term(i) is compatible with +

post: returns $\sum_{i=a}^{b} \text{term}(i)$

Partial Correctness

Termination argument also requires that (term i) returns a value for each $i \in [a, b]$

To put this another way- the user who wishes to use generalized-sigma must guarantee $a \leq b$, a and b integers AND that term has the properties required by the precondition.

All of square, cube, fourth, sqrt ... have these properties.

Notice that our proofs – both of partial correctness and termination – rely on the fact the evaluating (term a) does NOT change a or b. We know this because our functions have no side-effects. In C, you'd need to add this requirement to the precondition.

We can go even further with this usage of function parameters. Suppose, for example, that we wanted to compute the sum of the odds between a and b. We'd need of course to worry about the parity of a and b, but surely there is no need to clock a up by 1 at each step — why not have a step size of 2? This suggests

```
(define (even-more-general-sigma
          a term b next)
  (cond ((> a b) 0)
        (else (+ (term a)
                 (even-more-generalized-sigma
                   (next a) term
                   b next)))))
```

where a possible value for next is

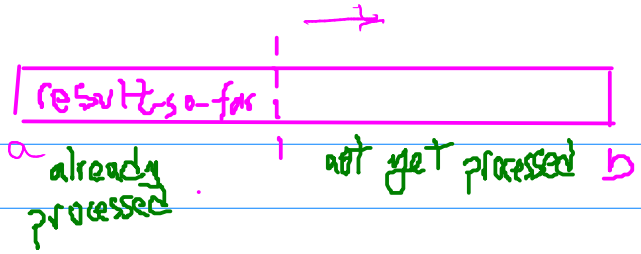
```
(lambda (x) (+ x 2))
```

You should develop a specification for the function next - and indeed, for this entire procedure.

Pay particular attention to the requirements imposed by termination!

For example, you might want to require that gap(next(a), b) < gap(a, b)

But in fact - though the reduction would need to be adjusted - all you would need is that next reduces the gap after a bounded number of calls...



Now consider an iterative version

```
(define (sigma a b)

  (define (iter a result-so-far)
    (cond ((= a b) result-so-far
          (else (iter (+ a 1) (+ (+ a 1) result-so-far))))))

  (cond ((> a b) 0)
        (else (iter a a))))


; with guess-invariant for iter as follows:

; a <= b AND result-so-far = sum of all integers i
; from A up to a, A <= i <= a, where A is the
; original value of the parameter a
```

Let's check the GI:

strong-enough? When $a = b$, this says result-so-far is the sum from A up to and including b - which is correct.

weak-enough? We need to check $\sum_{i=a}^{a} i = a$, and this is true

is it preserved? Yes, given the code and the fact that $\sum_{i=a}^{b} i = a + \sum_{i=a+1}^{b} i$.

Suggested Exercise

What changes are needed if result-so-far is initialized to 0, rather than a?

We can go even further with this usage of function parameters. Suppose, for example, that we wanted to compute the sum of the odds between a and b. We'd need of course to worry about the parity of a and b, but surely there is no need to clock a up by 1 at each step — why not have a step size of 2? This suggests

```
(define (even-more-general-sigma
          a term b next)
  (cond ((> a b) 0)
        (else (+ (term a)
                 (even-more-generalized-sigma
                   (next a) term
                   b next)))))
```

where a possible value for next is

```
(lambda (x) (+ x 2))
```

You should develop a specification for the function next - and indeed, for this entire procedure.

Pay particular attention to the requirements imposed by termination!

For example, you might want to require that gap(next(a), b) < gap(a, b)

But in fact - though the induction would need to be adjusted - all you would need is that next reduces the gap after a bounded number of calls...