CS61A Notes – Week 6a: Concurrency, Streams

A Concurrent March Through Programming Hell

On your computer, you often have multiple programs running at the same time – you might have your internet browser open browsing questionable pictures, your P2P software downloading non-pirated software, and your instant messaging client lying to a clueless middle-schooler across the country. But you have only one computer, and one CPU! How can you do so many things at once?

What actually happens is, the CPU switches between the different processes very quickly, doing work for each for a little while before moving on to the next, creating the illusion that the programs are running concurrently. The benefits are obvious for users like us so used to multitasking.

Unfortunately, parallelism is one of the biggest headaches you'll encounter. We'll attempt to give you a tiny migraine, but in CS162, you'll be swimming in a large ocean of pain with no shore in sight and a leaking life jacket.
A bit of syntax. To run things concurrently, we use a Scheme primitive called `parallel-execute`, a procedure that takes in any number of "thunks" – procedures that take no arguments – and executes the thunks in parallel. For example,

```
(define x 5)
(parallel-execute (lambda () (set! x (+ x 10)))
                  (lambda () (set! x (+ x 20))))
```

will attempt to set `x` to `(+ x 10)` and set `x` to `(+ x 20)` at the same time. Again, the computer "cheats" by interleaving operation between the different thunks. The answer that we want, of course, is `35` – we want the two thunks executed at the same time, but we still want the result to be *as if they executed consecutively*.

Now, consider this simple Scheme expression:

```
(set! x (+ x 10))
```

What looks like one Scheme operation is actually *three* operations:

1. lookup the value of `x`
2. add the value of `x` to `10`
3. store the result into `x`

Thus, consider the above call to `parallel-execute`, and keep in mind that the two thunks can be interleaved arbitrarily:

• lookup value of `x`	
• add `10` to the value of `x`	
• set `x` to the result	
	• lookup value of `x`
	• add `20` to the value of `x`
	• set `x` to the result

If the operations were interleaved in the above manner (not interleaved at all), then the value of `x` at the end is `35`.

• lookup value of `x`	
	• lookup value of `x`
• add `10` to the value of `x`	
	• add `20` to the value of `x`
	• set `x` to the result
• set `x` to the result	

In the above interleaving, the value of `x` ends up being `15`. This is not what we wanted!

QUESTION: What are the possible values of x after the below?

```
(define x 5)
(parallel-execute (lambda () (set! x (* x 2)))
                  (lambda () (if (even? x)
                                 (set! x (+ x 1))
                                 (set! x (+ x 100)))))
```

Concurrency: The Series

We use something called "serializers" to make sure that certain chunks of code are executed *together*. First, we need a way to create a serializer:

```
(define x-protector (make-serializer))
```

That was easy. A serializer takes in a procedure, and creates a *serialized* version of that procedure. So,

```
(define protected-plus-10 (x-protector (lambda () (set! x (+ x 10)))))
(define protected-plus-20 (x-protector (lambda () (set! x (+ x 20)))))
```

`protected-plus-10` still does the same thing as the original thunk – take in no arguments, and add `10` to `x`. However, because `protected-plus-10` and `protected-plus-20` are created *with the same serializer*, their instructions *will not be interleaved*. Therefore, in doing,

```
(parallel-execute protected-plus-10 protected-plus-20)
```

you can always be sure that `x` will be set to `35` at the end.

There's also a primitive object called a "mutex" that's even lower level than serializers (in fact, serializers are implemented with mutexes). You can interact with a mutex this way:

```
(define m (make-mutex))
(m 'acquire) ;; "reserves" the mutex
(m 'release) ;; "releases" the mutex
```

Once one program has acquired a mutex, if another wants to acquire the same mutex, it must wait until the mutex is released. So we can do this to obtain the same result:

```
(define x-mutex (make-mutex))
(parallel-execute
   (lambda () (x-mutex 'acquire) (set! x (+ x 10)) (x-mutex 'release))
   (lambda () (x-mutex 'acquire) (set! x (+ x 20)) (x-mutex 'release)))
```

The calls to acquire and release a mutex marks the *critical sections* of the code – sections that should *not* be interleaved with other processes also needing the same mutex.

When working with concurrency, there are four potential kinds of problems:

1. **incorrectness –** like the second interleaving example above, the answer you get might just be wrong
2. **inefficiency** – you could lock up the whole computer and always run only one program at a time, but that's horribly inefficient
3. **deadlocks** – if two programs are competing for the same two resources, there can be deadlocks
4. **unfairness** – one program may be unfairly favored to do more work than another

QUESTION: The Dining Politicians Problem. Politicians like to congregate once in a while, eat and spew nonsense. One slow Saturday afternoon, three politicians meet to have such wild fun. They sit around a circular table; however, due to the federal deficit (funny that these notes are timeless), they are provided with only three chopsticks, each lying in between two people. A politician will be able to eat only when both chopsticks next to him are not being used. If he cannot eat, he will just spew nonsense.

1. **Here is an attempt to simulate this behavior:**

```
(define (eat-talk i)
   (define (loop)
      (cond ((can-eat? i)
             (take-chopsticks i)
             (eat-a-while)
                (release-chopsticks i))
               (else (spew-nonsense)))
         (loop)
   (loop))

(parallel-execute (lambda () (eat-talk 0))
                  (lambda () (eat-talk 1))
                  (lambda () (eat-talk 2)))

;; a list of chopstick status, #t if usable, #f if taken
(define chopsticks '(#t #t #t))

;; does person i have both chopsticks?
(define (can-eat? i)
      (and (list-ref chopsticks (right-chopstick i))
        (list-ref chopsticks (left-chopstick i))))

;; let person i take both chopsticks
;; assume (list-set! ls i val) destructively sets the ith element of
;; ls to val
(define (take-chopsticks i)
   (list-set! chopsticks (right-chopstick i) #f)
   (list-set! chopsticks (left-chopstick i) #f))

;; let person i release both chopsticks
(define (release-chopsticks i)
   (list-set! chopsticks (right-chopstick i) #t)
   (list-set! chopsticks (left-chopstick i) #t))

;; some helper procedures
(define (left-chopstick i) (if (= i 2) 0 (+ i 1)))
(define (right-chopstick i) i)
```

Is this correct? If not, what kind of hazard does this create?

2. **Here's a proposed fix:**
```
(define protector (make-serializer))
(parallel-execute (protector (lambda () (eat-talk 0)))
(protector (lambda () (eat-talk 1)))
(protector (lambda () (eat-talk 2))))
```
Does this work?

3. **Here's another proposed fix: use one mutex per chopstick, and acquire both before doing anything:**
```
(define protectors
   (list (make-mutex) (make-mutex) (make-mutex)))

(define (eat-talk i)
   (define (loop)
      ((list-ref protectors (right-chopstick i)) 'acquire)
      ((list-ref protectors (left-chopstick i)) 'acquire)
      (cond ... ;; as before)
      ((list-ref protectors (right-chopstick i)) 'release)
      ((list-ref protectors (left-chopstick i)) 'release)
      (loop))
   (loop))
```
Does that work?

4. **What about this:**
```
(define m (make-mutex))
(define (eat-talk i)
   (define (loop)
      (m 'acquire)
      (cond ... ;; as before)
      (m 'release)
      (loop))
   (loop))
```

5. **So what would be a good solution?**

(Note: This problem is commonly referred to as "The Dining Philosophers" problem. However, here at Berkeley, we prefer to look down on politicians rather than philosophers.)

Streaming Along

A stream is an element and a "promise" to evaluate the rest of the stream. *You've already seen multiple examples of this and its syntax in lecture and in the book, so I will not dwell on that. Suffice it to say, streams is one of the most mysterious topics in CS61A, but it's also one of the coolest; mysterious, because defining a stream often seems like black magic (and requires MUCH more trust than whatever trust you worked up for recursion); cool, because things like infinite streams allows you to store an INFINITE amount of data in a FINITE amount of space/time!*

How is that possible? We're not going to be too concerned with the below-the-line implementations of streams, but it's good to have an intuition. Recall that the body of a `lambda` is NOT executed until it is called. For example, typing into `STk`:

```
(define death (lambda () (/ 5 0)))
```

Scheme says "okay", happily binding `death` to the `lambda`. But if you try to run it:

```
(death) ;; Scheme blows up
```

The crucial thing to notice is that, when you type the `define` statement, Scheme did NOT try to evaluate `(/ 5 0)` – otherwise, it would've died right on the spot. Instead, the evaluation of `(/ 5 0)` is *delayed* until the actual procedure call. Similarly, if we want to represent an infinite amount of information, we don't have to calculate all of it at once; instead, we can simply calculate ONE piece of information (the `stream-car`), and leave instructions on how to calculate the NEXT piece of information (the "promise" to evaluate the rest).

It's important to note, however, that Scheme doesn't quite use plain `lambda` for streams. Instead, Scheme memorizes results of evaluating streams to maximize efficiency. This introduces some complications that we'll visit later. The `delay` and `force` operators, therefore, are special operators with side effects.

QUESTIONS

1. Define a procedure `(ones)` that, when run with no arguments, returns a cons pair whose `car` is `1`, and whose `cdr` is a procedure that, when run, does the same thing.

2. Define a procedure `(integers-starting n)` that takes in a number `n` and, when run, returns a `cons` pair whose `car` is `n`, and whose `cdr` is a procedure that, when run with no arguments, does the same thing for `n+1`.

Using Stream Operators

Here are some that we'll be using quite a bit:
`(stream-map proc s ...)` – works just like list `map`; can take in any number of streams
`(stream-filter proc s)` – works just like list `filter`
`(stream-append s1 s2)` – appends two finite streams together (why not infinite streams?)
`(interleave s1 s2)` – interleave two streams into one, with alternating elements from `s1` and `s2`

Constructing Streams

This is the trickiest part of streams. I said that the topic of streams is a black art, and you'll soon see why. The construction of streams is counter-intuitive with a heavy dose of that-can't-possibly-work. So here are some rough guidelines:

1. **`cons-stream` is a special form!** `cons-stream` will NOT evaluate its second argument (the `stream-cdr`); obviously, this is desirable, since we'd like to delay that evaluation.

2. **Trust the, err, stream.** From the first day, we've been chanting "trust the recursion". Well now that you're (slightly more) comfortable with that idea, we need you to do something harder. When you're defining a stream, you **have to think as if that stream is already defined**. It's often very difficult to trace through how a stream is evaluated as you `stream-cdr` down it, so you have to work at the logical level. Therefore, the above definition of `integers` works. However, be careful that you don't trust the stream too much. For example, this won't work:

```
(define integers integers)
```

3. **Learn how to think about `stream-map`.** Consider this definition of `integers`, given the stream `ones`, a stream of ones, defined in SICP:

```
(define integers (cons-stream 1 (stream-map + ones integers)))
```

If the above definition of `integers` puzzles you a bit, here's how to think about it:

```
   1                          <= your stream-car
       1   2   3   4   5   6   ...   <= integers (as taken from the last line)
+      1   1   1   1   1   1   ...   <= ones
===========================
   1   2   3   4   5   6   7   ...   <= integers
```

If you're ever confounded by a `stream-map` expression, write it out and all should be clear. For example, let's try a harder one – `partial-sum`, whose `i`th element is the sum of the first i integers. It is defined thus:

```
(define partial-sum (cons-stream 0 (stream-map + partial-sum integers)))
```

```
   0                          <= your stream-car
       0   1   3   6   10  15  ...   <= partial-sum
+      1   2   3   4   5   6   ...   <= ones
===========================
   0   1   3   6   10  15  20  ...   <= partial-sum
```

Now, if you find it odd to have `integers` or `partial-sum` as one of the things you're adding up, refer to guideline #2.

4. **Specify the first element(s).** Recall that a stream is one element and a promise to evaluate more. Well, often, you have to specify that one element so that there's a starting point. Therefore, unsurprisingly, when you define streams, it often looks like

```
(cons-stream [first element] [a stream of black magic]).
```

But there are many traps in this. In general, what you're avoiding is an infinite loop when you try to look at some element of a stream. `stream-cdr` is usually the dangerous one here, as it may force evaluations that you want to delay. Note that **Scheme stops evaluating a stream once it finds one element**. So simply make sure that it'll always find one element immediately. For example, consider this definition of `fibs` that produces a stream of Fibonacci numbers:

```
(define fibs (cons-stream 0 (stream-map + fibs (stream-cdr fibs))))
```

Its intentions are admirable enough; to construct the next `fib` number, we add the current one to the previous one. But let's take a look at how it logically stacks up:

```
      0                         <= your stream-car
         0  1  1  2  3  5  ...  <= fibs?
   +     1  1  2  3  5  8  ...  <= (stream-cdr fibs)
   ===========================
0  1  2  3  5  8  13 ...  <= not quite fibs...
```

Close, but no cigar (and by the definition of Fibonacci numbers you really can't just start with a single number). Actually, it's even worse than that; if you type in the above definition of `fibs`, and call `(stream-cdr fibs)`, you'll send `STk` into a most unfortunate infinite loop. Why? Well, `stream-cdr` forces the evaluation of `(stream-map + fibs (stream-cdr fibs))`. **`stream-map` is not a special form**, so it's going to evaluate both its arguments, `fibs` and `(stream-cdr fibs)`. What's `fibs`? Well, `fibs` is a stream starting with `0`, so that's fine. What's `(stream-cdr fibs)`? Well, `stream-cdr` forces the evaluation of `(stream-map + fibs (stream-cdr fibs))`. **`stream-map` is not a special form**, so it's going to evaluate both its arguments, `fibs` and `(stream-cdr fibs)`. What's `fibs`? Well, `fibs` is a stream starting with `0`, so that's fine. What's `(stream-cdr fibs)`? You get the point.

How do we stop that horrid infinite loop? Well, it was asking for `(stream-cdr fibs)` that was giving us trouble – whenever we try to evaluate `(stream-cdr fibs)`, it goes into an infinite loop. Thus, why don't we just specify the `stream-cdr`?

```
(define fibs
   (cons-stream 0
      (cons-stream 1 (add-streams fibs (stream-cdr fibs)))))
```

So, then, let's try it again. What's `(stream-cdr fibs)`? Well, `(stream-cdr fibs)` is a stream starting with `1`. There! Done! See? Now, it's pretty magical that adding one more element fixes the `stream-cdr` problem for the whole stream. Convince yourself of this. As a general rule of thumb, **if in the body of your definition you use the `stream-cdr` of what you're defining, you probably need to specify two elements**. Let's check that it logically works out as well:

```
      0  1                         <= your stream-car
            0  1  1  2  3  5  ...  <= fibs
   +        1  1  2  3  5  8  ...  <= (stream-cdr fibs)
   ==============================
0  1  1  2  3  5  8  13 ...  <= win!
```

QUESTIONS: Describe what the following expressions define.

1.
```
(define s1 (add-stream (stream-map (lambda (x) (* x 2)) s1) s1))
```

2.
```
(define s2
  (cons-stream 1
    (add-stream (stream-map (lambda (x) (* x 2)) s2) s2)))
```

3.
```
(define s3
  (cons-stream 1
    (stream-filter (lambda (x) (not (= x 1))) s3)))
```

4.
```
(define s4
  (cons-stream 1
    (cons-stream 2
      (stream-filter (lambda (x) (not (= x 1))) s4))))
```

5.
```
(define s5
  (cons-stream 1
    (add-streams s5 integers)))
```

6. **Define `facts` without defining any procedures; the stream should be a stream of 1!, 2!, 3!, 4!, etc. More specifically, it returns a stream with elements `(1 2 6 24 ...)`**

```
(define facts
  (cons-stream
```

7. **(HARD!) Define powers; the stream should be 1^1, 2^2, 3^3, …, or, `(1 4 16 64 ...)`. Of course, you cannot use the exponents procedure. I've given you a start, but you don't have to use it.**

```
(define powers (helper integers integers))
(define (helper s t)
  (cons-stream
```

Constructing Streams Through Procedures

You'll find this the most natural way to construct streams, since it mirrors recursion so much. For example, to use a trite example,

```
(define (integers-starting n) (cons-stream n (integers-starting (+ n 1))))
```

So `(integers-starting 1)` is a stream whose first element is `1`, with a promise to evaluate `(integers-starting 2)`. The rules are similar to above; you still specify a first element, etc. Pretty simple? Let's try a few.

QUESTIONS

1. **Define a procedure, `(list->stream ls)` that takes in a list and converts it into a stream.**

```
(define (list->stream ls)
```

2. Define a procedure `(lists-starting n)` that takes in `n` and returns a stream containing `(n)`, `(n n+1)`, `(n n+1 n+2)`, etc. For example, `(lists-starting 1)` returns a stream containing `(1) (1 2) (1 2 3) (1 2 3 4)`…

```
(define (lists-starting n)
```

3. Define a procedure `(chocolate name)` that takes in a name and returns a stream like so:

```
(chocolate 'chung) =>
  (chung really likes chocolate chung really really likes chocolate chung really really
really likes chocolate ...)
```

You'll want to use helper procedures.

```
(define (chocolate name)
```

Stream Processing

Sometimes you'll be asked to write procedures that convert one given stream into another exerting a certain property.

QUESTIONS:

1. Define a procedure, `(stream-censor s replacements)` that takes in a stream `s` and a table `replacements` and returns a stream with all the instances of all the `car` of entries in `replacements` replaced with the `cadr` of entries in `replacements`:

```
(stream-censor (hello you weirdo ...) ((you I-am) (weirdo an-idiot))) =>
  (hello I-am an-idiot ...)
(define (stream-censor s replacements)
```

2. Define a procedure `(make-alternating s)` that takes in a stream of positive numbers and alternates their signs. So `(make-alternating ones) => (1 -1 1 -1 ...)` and `(make-alternating integers) => (1 -2 3 -4 ...)`. Assume `s` is an infinite stream.

```
(define (make-alternating s)
```

My Body's Floating Down the Muddy Stream

Now, more fun with streams!

MORE QUESTIONS

1. Given streams `ones`, `twos`, `threes`, and `fours`, write down the first ten elements of:

```
(interleave ones (interleave twos (interleave threes fours)))
```

2. Construct a stream `all-integers` that includes 0 and both the negative and positive integers.

```
(define all-integers
```

3. Suppose we were foolish enough to try to implement `stream-accumulate`:

```
(define (stream-accumulate combiner null-value s)
   (cond ((stream-null? s) null-value)
         (else (combiner
                  (stream-car s)
                  (stream-accumulate combiner null-value (stream-cdr s))))))
```

What happens when we do:

a. `(define foo (stream-accumulate + 0 integers))`

b. `(define bar (cons-stream 1 (stream-accumulate + 0 integers)))`

c.
```
(define baz (stream-accumulate
               (lambda (x y) (cons-stream x y))
               the-empty-stream integers))
```

4. ***SICP* ex. 3.68, page 341. If you understand this, you'll be fine.**