CS 61A Lecture Notes Second Half of Week 5

Topic: Streams

Reading: Abelson & Sussman, Section 3.5.1-3, 3.5.5

Streams are an abstract data type, not so different from rational numbers, in that we have constructors and selectors for them. But we use a clever trick to achieve tremendously magical results. As we talk about the mechanics of streams, there are three big ideas to keep in mind:

- Efficiency: Decouple order of evaluation from the form of the program.
- Infinite data sets.
- Functional representation of time-varying information (versus OOP).

You'll understand what these all mean after we look at some examples.

How do we tell if a number n is prime? Never mind computers, how would you express this idea as a mathematician? Something like this: "N is prime if it has no factors in the range $2 \leq f < n$."

So, to implement this on a computer, we should

- Get all the numbers in the range $[2, n-1]$.
- See which of those are factors of n.
- See if the result is empty.

```
;;;;;                       In file cs61a/lectures/3.5/prime1.scm
(define (prime? n)
  (null? (filter (lambda (x) (= (remainder n x) 0))
                 (enumerate-interval 2 (- n 1)))))
```

But we don't usually program it that way. Instead, we write a *loop*:

```
;;;;;                       In file cs61a/lectures/3.5/prime0.scm
(define (prime? n)
  (define (iter factor)
    (cond ((= factor n) #t)
          ((= (remainder n factor) 0) #f)
          (else (iter (+ factor 1)))))
  (iter 2))
```

(Never mind that we can make small optimizations like only checking for factors up to $\sqrt{n}$. Let's keep it simple.)

Why don't we write it the way we expressed the problem in words? The problem is one of efficiency. Let's say we want to know if 1000 is prime. We end up constructing a list of 998 numbers and testing *all* of them as possible factors of 1000, when testing the first possible factor would have given us a false result quickly.

The idea of streams is to let us have our cake and eat it too. We'll write a program that *looks like* the first version, but *runs like* the second one. All we do is change the second version to use the stream ADT instead of the list ADT:

```
;;;;;                          In file cs61a/lectures/3.5/prime2.scm
(define (prime? n)
  (stream-null? (stream-filter (lambda (x) (= (remainder n x) 0))
                               (stream-enumerate-interval 2 (- n 1)))))
```

The only changes are `stream-enumerate-interval` instead of `enumerate-interval`, `stream-null?` instead of `null?`, and `stream-filter` instead of `filter`.

How does it work? A list is implemented as a pair whose `car` is the first element and whose `cdr` is the rest of the elements. A stream is almost the same: It's a pair whose `car` is the first element and whose `cdr` is a *promise* to compute the rest of the elements later.

For example, when we ask for the range of numbers $[2, 999]$ what we get is a single pair whose `car` is 2 and whose `cdr` is a promise to compute the range $[3, 999]$. The function `stream-enumerate-interval` returns that single pair. What does `stream-filter` do with it? Since the first number, 2, does satisfy the predicate, `stream-filter` returns a single pair whose `car` is 2 and whose `cdr` is a promise *to filter* the range $[3, 999]$. `Stream-filter` returns that pair. So far no promises have been "cashed in." What does `stream-null?` do? It sees that its argument stream contains the number 2, and maybe contains some more stuff, although maybe not. But at least it contains the number 2, so it's not empty. `Stream-null?` returns `#f` right away, without computing or testing any more numbers.

Sometimes (for example, if the number we're checking *is* prime) you do have to cash in the promises. If so, the stream program still follows the same order of events as the original loop program; it tries one number at a time until either a factor is found or there are no more numbers to try.

Summary: What we've accomplished is to decouple the form of a program—the order in which computations are presented—from the actual order of evaluation. This is one more step on the long march that this whole course is about, i.e., letting us write programs in language that reflects the problems we're trying to solve instead of reflecting the way computers work.

• Implementation. How does it work? The crucial point is that when we say something like

```
(cons-stream from (stream-enumerate-interval (+ from 1) to))
```

(inside `stream-enumerate-interval`) we can't actually evaluate the second argument to `cons-stream`. That would defeat the object, which is to defer that evaluation until later (or maybe never). Therefore, `cons-stream` has to be a special form. It has to `cons` its first argument onto a promise to compute the second argument. The expression

```
(cons-stream a b)
```

is equivalent to

```
(cons a (delay b))
```

`Delay` is itself a special form, the one that constructs a promise. Promises could be a primitive data type, but since this is Scheme, we can represent a promise as a function. So the expression

```
(delay b)
```

really just means

```
(lambda () b)
```

We use the promised expression as the body of a function with no arguments. (A function with no arguments is called a *thunk*.)

Once we have this mechanism, we can use ordinary functions to redeem our promises:

```
(define (force promise) (promise))
```

and now we can write the selectors for streams:

```
(define (stream-car stream) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

Notice that forcing a promise doesn't compute the entire rest of the job at once, necessarily. For example, if we take our range $[2, 999]$ and ask for its tail, we don't get a list of 997 values. All we get is a pair whose `car` is 3 and whose `cdr` is a new promise to compute $[4, 999]$ later.

The name for this whole technique is *lazy evaluation* or *call by need*.

• Reordering and functional programming. Suppose your program is written to include the following sequence of steps:

```
...
(set! x 2)
...
(set! y (+ x 3))
...
(set! x 7)
...
```

Now suppose that, because we're using some form of lazy evaluation, the actual sequence of events is reordered so that the third `set!` happens before the second one. We'll end up with the wrong value for `y`. This example shows that we can only get away with below-the-line reordering if the above-the-line computation is functional.

(Why isn't it a problem with `let`? Because `let` doesn't mutate the value of one variable in one environment. It sets up a local environment, and any expression within the body of the `let` has to be computed within that environment, even if things are reordered.)

• Infinite streams. Think about the plain old list function

```
(define (enumerate-interval from to)
  (if (> from to)
      '()
      (cons from (enumerate-interval (+ from 1) to)) ))
```

When we change this to a stream function, we change very little in the appearance of the program:

```
(define (stream-enumerate-interval from to)
  (if (> from to)
      THE-EMPTY-STREAM
      (cons-STREAM from (stream-enumerate-interval (+ from 1) to)) ))
```

but this tiny above-the-line change makes an enormous difference in the actual behavior of the program.

Now let's cross out the second argument and the end test:

```
(define (stream-enumerate-interval from)
  (cons-stream from (stream-enumerate-interval (+ from 1))) )
```

This is an *enormous* above-the-line change! We now have what looks like a recursive function with no base case—an infinite loop. And yet there is hardly any difference at all in the actual behavior of the program. The old version computed a range such as $[2, 999]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, 999]$ later. The new version computes a range such as $[2, \infty]$ by constructing a single pair whose `car` is 2 and whose `cdr` is a promise to compute $[3, \infty]$ later!

This amazing idea lets us construct even some pretty complicated infinite sets, such as the set of all the prime numbers. (Explain the sieve of Eratosthenes. The program is in the book so it's not reproduced here.)

• Time-varying information. Functional programming works great for situations in which we are looking for a timeless answer to some question. That is, the same question always has the same answer regardless of events in the world. We invented OOP because functional programming didn't let us model changing state. But with streams we *can* model state functionally. We can say

```
(define (user-stream)
  (cons-stream (read) (user-stream)) )
```

and this gives us *the stream of everything the user is going to type* from now on. Instead of using local state variables to remember the effect of each thing the user types, one at a time, we can write a program that computes the result of the (possibly infinite) collection of user requests all at once! This feels really bizarre, but it does mean that purely functional programming languages can handle user interaction. We don't *need* OOP.