CS2135: Introduction to Streams

Kathi Fisler, WPI

February 14, 2003

1 The Never-Ending Traffic Light

Our monitor application is coming along well. We have a fast implementation and a nice syntax that can interface to either data definition. What else might we want to do with this application?

It does have one drawback at the moment. What are the inputs to *interp-monitor*? We give it the monitor and a list of samples. How many samples can be in the list? An arbitrary number, but it must be finite. When can we run the monitor? Only *after* we've collected the samples. Wouldn't it be better to run the monitor *while* the traffic light is running, so we can check for errors on the spot?

These questions suggest two problems with our current approach: we need to collect all the samples before we run the monitor, and we need to decide up-front how many samples to give. Can we wait until the traffic light stops running to stop collecting samples? Ideally, traffic lights would never malfunction, so deciding when to stop sampling seems a bit difficult.[1]

We'd really like a way to generate the samples as we need them, rather than have to gather them up-front. This would address both problems, because we can keep generating samples until we're satisfied in our testing of the traffic light, and the traffic light can request the next sample each time it's ready to take a transition. The goal of today's lecture is to figure out how to produce an arbitrary-length sequence of samples (or any other data) on demand.

2 Streams

Technically, the data structure we're looking to define is called a *stream*. Streams are like lists in that they can have an arbitrary number of elements, they could be empty or not, and they have both first elements and the rest of the stream. Streams are unlike lists in that they can have an infinite number of elements, and we only get the elements as we ask for them (necessary if you have an infinite-length data structure, unless you plan to wait a while—and then some). Given that there's *some* resemblence, let's try to implement streams with lists and see where we get.

As an initial example, let's try to write code to generate an infinite stream of natural numbers (the numbers starting from 0). Since we're starting from lists, let's first write a similar function on lists: one that generates the list of all numbers from up to a given number.

```
;; make-num-list : number number → list[number]
;; generate a list of all numbers from the first up to (but not including) the second
(define (make-num-list start-from stop-at)
  (cond [(= start-from stop-at) empty]
        [else (cons start-from (make-num-list (+ 1 start-from) stop-at))]))

> (make-num-list 0 5)
(cons 0 (cons 1 (cons 2 (cons 3 (cons 4 empty)))))
```

What's the difference between this code and the version for streams? In the streams case, we don't want to stop at a particular point–we just want to generate an infinite stream. Since we don't want to stop, let's try writing *make-num-stream* by copying the code for *make-num-list* and deleting all the code that pertains to *stop-at*:

[1] Admittedly, there are only so many sample sequences in a traffic light, so this may seem like a silly question. On a real system, though, we might have more sequences than a computer could generate in your lifetime, so this becomes an issue.

```
;; make-num-stream : number → stream[number]
;; generate a stream of all numbers starting from the given input
(define (make-num-stream start-from)
  (cond
        [else (cons start-from (make-num-stream (+ 1 start-from)))]))
```

We can eliminate the **cond** with just the **else** clause by just returning the answer in the **else** case:

```
;; make-num-stream : number → stream[number]
;; generate a stream of all numbers starting from the given input
(define (make-num-stream start-from)
  (cons start-from (make-num-stream (+ 1 start-from))))
```

What happens if we run (*make-num-stream* 0)? We'll get an infinite loop (which is why we needed the *stop-at* num in the first place). We don't want to put a limit on the maximum number, but we are willing to extract just one number at a time. Under those constraints, what might we do?

Remember our now standard technique of delaying evaluation: add a **lambda**. Where do we add the **lambda**? Around whatever expression we want to delay, which in this case, is the recursive call (since that causes the infinite loop):

```
;; make-num-stream : number → stream[number]
;; generate a stream of all numbers starting from the given input
(define (make-num-stream start-from)
  (cons start-from
        (lambda () (make-num-stream (+ 1 start-from)))))
```

What would (*make-num-stream* 0) return now? It looks like it would return a *cons* with a number and a function. But we know that *cons* needs a list as the second argument, so we need to use some other data structure to glue together the number and the function.[2] We'll introduce a *stream* struct for this purpose. We'll call the fields *first* and *make-rest* to maintain the parallel to lists (acknowledging that the second argument is a function that makes a stream, but isn't a stream until we call the function). Before reading on, ask yourself what the right data definition should be for a stream.

```
;; A stream is a (make-stream value (→ stream))
(define-struct stream (first make-rest))

;; make-num-stream : number → stream[number]
;; generate a stream of all numbers starting from the given input
(define (make-num-stream start-from)
  (make-stream start-from
               (lambda () (make-num-stream (+ 1 start-from)))))
```

Running (*make-num-stream* 0) terminates and gives us a data structure, but does it do what we want? Does it generate a (seemingly) infinite sequence of the numbers (in order) and only on demand? How could we test this? Let's make a *num-stream* and try to extract its first element:

```
> (define s1 (make-num-stream 0))
> (stream-first s1)
0
```

Looks good. How do we get to the second stream and make sure that its first element is 1?

```
> (define s2 (stream-make-rest s1))
> (stream-first s2)
[BUG] stream-first: expects argument of type <struct:stream>; given <procedure:40:16>
```

[2] We could actually use *cons* here: once you get to the "Pretty Big" language level, the second argument to *cons* can be anything you want. We're won't avail of that feature here because I prefer to keep *cons* clean with a list in the second position—templates and all other sorts of design instincts break otherwise.

What happened? Remember, the value in the *make-rest* position of a stream is a function that returns a stream. If we want to extract the first element, we need to first call that function to get the stream, then call *stream-first*:

```
> (define s2 (stream-make-rest s1))
> (stream-first (s2))
1
> (define s3 (stream-make-rest (s2)))
> (stream-first (s3))
2
```

This seems to be working in terms of generating the numbers, but it does seem a bit clunky to wrap the extra parens around the result of *stream-make-rest* each time. Let's clean that up by defining our own *stream-rest* operator:

```
;; stream-rest : stream → stream
;; return stream for the rest of the given stream
(define (stream-rest astream)
  ((stream-make-rest astream)))

> (define s1 (make-num-stream 0))
> (stream-first s1)
0
> (define s2 (stream-rest s1))
> (stream-first s2)
1
> (define s3 (stream-rest s2))
> (stream-first s3)
2
```

Looking good. Just to be sure, let's wrap a little interface around this that lets a user keep asking for numbers from the stream as long as she wants:

```
(define (nums-until-stop nstream)
  (begin (printf "Another number?: ")
         (cond [(symbol=? (read) 'y)
                (begin (printf "Next num: ~a~n" (stream-first nstream))
                       (nums-until-stop (stream-rest nstream)))]
               [else (printf "No more numbers requested~n")])))

> (nums-until-stop (make-num-stream 0))
Another number?: y
Next num: 0
Another number?: y
Next num: 1
Another number?: y
Next num: 2
Another number?: y
Next num: 3
Another number?: y
Next num: 4
Another number?: n
No more numbers requested
```

Recap

Let's recap what we've seen so far:

- Streams are similar to lists, but differ in two key ways: they should only generate their elements on request, and they can have an infinite number of elements.
- Given these differences, we need slightly different operators to implement streams from *first*, *rest*, and *cons* as defined for lists.
- In order to achieve generation-on-demand, we must delay building the rest of a stream until a user has asked for the first element from the stream. To implement this feature, we use **lambda**, our standard technique for delaying computation.
- We introduced a new *stream* struct to bundle together the first element and the function for making the rest of the stream. We also wrote a *stream-rest* function to clean up the operator for accessing the rest of a stream as a stream.

As this point, our operators for streams are as follows:

- *make-stream : value* (-> *stream*) -> *stream*
- *stream-first : stream -> value*
- *stream-rest : stream -> stream*

The *make-stream* operator is a bit annoying, because it requires the programmer to include the **lambda**. A cleaner operator for creating streams would suppress that detail, while providing the same functionality. Let's create such an operator, and call it *stream-cons*. Given such an operator, we'd write *make-num-stream* as:

```
;; make-num-stream : number → stream[number]
;; generate a stream of all numbers starting from the given input
(define (make-num-stream start-from)
  (stream-cons start-from
               (make-num-stream (+ 1 start-from))))
```

Should *stream-cons* be a function or a macro? Since the goal of a stream is to avoid evaluating the rest of the stream until later, we'll have to use a macro.

```
(define-syntax stream-cons
  (syntax-rules ()
    [(stream-cons f r)
     (make-stream f (lambda () r))]))
```

This finishes our introduction to streams. Next class, we'll take a deeper look at how and where to use them (they're good for more than just traffic lights!).