CS2135: Applications of Streams

Kathi Fisler, WPI

February 18, 2003

1 A Streams Recap

At this point, we've motivated streams (infinite sequences computed on demand) and implemented a series of operators for working with streams. As a reminder, we have:

- *stream-cons : value stream -> stream*
- *stream-first : stream -> value*
- *stream-rest : stream -> stream*

We've intentionally designed these operators to resemble lists in their names and their contracts (i.e., that *stream-rest* returns the rest of the stream, not the function that makes the rest of the stream).

Can Streams be Finite?

If we continue our analogy with lists, we find one aspect of lists missing: we don't have a notion of *empty* for lists. On the one hand, this makes sense, because we said streams were designed to support infinite sequences of values. Recall, however, that streams satisfy *two* criteria: infinite length, and data on demand. We could imagine a situation in which we want data on demand, but still allow that data to be finite in length (for example: we might want to generate the traffic light inputs as the light changes—the on-demand part—while allowing ourselves to only test the light for a limited period of time).

As a result, we really should add a notion of an empty stream. To do this, we'll add the following definitions to our language of stream operators:

```
(define empty-stream 'end-of-stream)

;; stream-empty? : value → boolean
;; determine whether given value is the empty stream
(define (stream-empty? v)
  (eq? v empty-stream))
```

These definitions declare *empty-stream* as a constant for marking the end of a stream (like *empty* does for lists), and provide an operator for testing whether a value is the empty stream. Notice that I use an operator called *eq?* instead of *symbol=?* in the definition of *stream-empty?*. Recall that *symbol=?* expects both arguments to be symbols. The *stream-empty?* operator, however, could be used to test whether some arbitrary value is an empty stream, so *symbol=?* won't work. The *eq?* operator tests any two values for equality, regardless of their types. If given two symbols, it behaves as does *symbol=?*.

Now that we have our extended stream operators, let's look at some applications of streams.

2 Streams in Software

Data-on-demand is an increasingly common pattern in software, especially in the context of the Internet. Many of you have heard of "streaming video", "streaming media", or other terms that include a notion of data-on-demand. Applications based on streaming data must be able to handle data coming in piecemeal, rather than all at one time. (Note that this raises some interesting questions, since many operations we'd like to perform on data assume that you have all of the data available. For example, how would you get the largest number in a streaming list of numbers? Or compute the size of the data?).

Let's consider a simple example. Stock market traders routinely watch a stream of data on current stock prices. As this data goes by, the traders want to watch for noticable changes in prices (perhaps to buy a favorite stock that has suddenly dropped in price, or to sell something that seems to be dropping unexpectedly). If we want to provide software support for stock traders, we need to provide operators that help them watch for certain trends in stock tickers.

Let's assume we have a structure for stock-ticker information. The structure holds the stock name (a symbol) and the change in price since the last report on this datum:

```
;; A ticker is a (make-ticker symbol number)
(define-struct ticker (stock change))
```

Also assume that we have an input stream of tickers. The code file that goes with this lecture contains a function for generating a random input stream of tickers; in this example, we are going to assume we have such a stream already, and just worry about processing it.

Now, let's write a function *large-changes* that consumes a stream of tickers and produces a stream of tickers for that show changes of at least one dollar in either direction. For example, if the input stream started with the tickers (*make-ticker* 'intel 1), (*make-ticker* 'disney −.4), and (*make-ticker* 'amazon −2), then the tickers (*make-ticker* 'intel 1) and (*make-ticker* 'amazon −2) should be in the output stream:

```
;; large-changes : stream[ticker] → stream[ticker]
;; produce stream of all tickers containing changes of more than one
;; dollar in either direction
(define (large-changes tstream)
  (cond [(stream-empty? tstream) empty-stream]
        [(stream? tstream)
         (cond [(<= 1 (abs (ticker-change (stream-first tstream))))
                (stream-cons (stream-first tstream)
                             (large-changes (stream-rest tstream)))]
               [else (large-changes (stream-rest tstream))])]))
```

Wait – doesn't this look a *lot* like *filter*? Sure does, and that's not surprising given the similarity between our stream and list operators. Can we write a generic *stream-filter* the same way we did for lists? Certainly:

```
;; stream-filter : (alpha→bool) stream[alpha] → stream[alpha]
;; produce stream of all elements satisfying given predicate
(define (stream-filter keep? astream)
  (cond [(stream-empty? astream) empty-stream]
        [(stream? astream)
         (cond [(keep? (stream-first astream))
                (stream-cons (stream-first astream)
                             (stream-filter keep? (stream-rest astream)))]
               [else (stream-filter keep? (stream-rest astream))])]))
```

You could write a *stream-map* in similar spirit. Notice that both *stream-filter* and *stream-map* look just like their corresponding list versions, only using different operators to build and access the stream versus list data: for example, *cons* becomes *stream-cons* in the streams version (and so on for *first* and *rest*).

This example shows how you can think about streams similarly to lists, at least for some programs. At least with regards to operations like *map* and *filter*, stream programs look just like list functions. Using these operators, you could write various streaming-data programs for filtering and transforming data on demand.

3 Streams in Hardware

Streams were a useful data structure long before the streaming media of the Internet age. The idea of computing data on demand underlies operating systems, computer hardware, and networking. In this section, we look at using streams to write programs to simulate hardware circuits.

3.1 A Crash Course in Hardware

Hardware is built from a series of small devices called *gates* and *delays*. At a basic level, gates perform simple boolean operations on data, such as and, or, and not. Delays serve as a basic form of memory; computationally, they delay their inputs by one computation step. Hardware systems consist of connections of gates and delays. Given a sequence of inputs, they produce a corresponding sequence of outputs.

We want to write functions to simulate a hardware system. A hardware simulation will take some streams of boolean values as inputs and will produce streams of boolean values as outputs. The outputs let us inspect whether the hardware system is producing the correct function; we could write monitors over the output streams to test the hardware just as we monitored the traffic light. In this lecture, we won't add monitors to the hardware systems; we just want to implement the systems themselves.

First, we need to implement the gates and the delays. The gates are easier, so let's start with them. Each runs over one or two streams and produces a stream that combines the data from the input streams. For example:

```
(stream-and (stream-cons true (stream-cons true (stream-cons false stream-empty)))
            (stream-cons false (stream-cons true (stream-cons false stream-empty))))
= (stream-cons false (stream-cons true (stream-cons false stream-empty)))
```

With this intuition in hand, the gate implementations are as follows:

```
;; stream-and : stream[bool] stream[bool] → stream[bool]
;; produces a stream of the and of two input streams
(define (stream-and s1 s2)
  (cond [(or (stream-empty? s1) (stream-empty? s2)) empty-stream]
        [else (stream-cons (and (stream-first s1) (stream-first s2))
                           (stream-and (stream-rest s1) (stream-rest s2)))]))

;; stream-or : stream[bool] stream[bool] → stream[bool]
;; produces a stream of the or of two input streams
(define (stream-or s1 s2)
  (cond [(or (stream-empty? s1) (stream-empty? s2)) empty-stream]
        [else (stream-cons (or (stream-first s1) (stream-first s2))
                           (stream-and (stream-rest s1) (stream-rest s2)))]))

;; stream-not : stream[bool] → stream[bool]
;; produces a stream of the logical negation of an input streams
(define (stream-not s1)
  (stream-map not s1))
```

Let's develop *stream-delay* by working through an example.

```
(stream-delay (stream-cons true (stream-cons false (stream-cons false stream-empty))))
```

What should this return? By definition, *stream-delay* delays the stream contents by one step. This suggests the return value of

```
(stream-cons ??? (stream-cons true (stream-cons false (stream-cons false stream-empty))))
```

What goes at the front of the stream? The value is somewhat undefined, since there is no previous value from the original stream at the first step of the computation. We'll therefore leave a parameter in *stream-delay* for this undefined value:

```
;; stream-delay : stream[bool] → stream[bool]
;; produces a stream containing the same data as the input stream, but
;; delayed by one computation
(define (stream-delay s1 base)
  (stream-cons base s1))
```

With these definitions in hand, let's use them to implement a simple hardware device.

3.2 A Simple Hardware Device: A Single Pulser

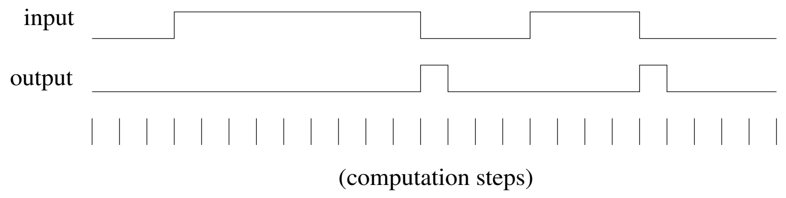
A single pulser is a simple device that converts each sequence of true values in a stream into a single true value (one might use this to convert a long push of a button into one push from the system's perspective). For example, given the input sequence

```
(stream-cons false
  (stream-cons true
    (stream-cons true
      (stream-cons true
        (stream-cons false
          (stream-cons false
            (stream-cons true
              (stream-cons false stream-empty))))))))
```

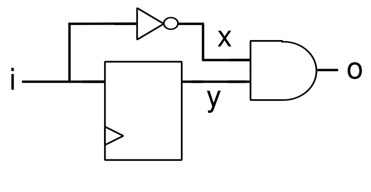
The pulser would output

```
(stream-cons false
  (stream-cons false
    (stream-cons false
      (stream-cons false
        (stream-cons true
          (stream-cons false
            (stream-cons false
              (stream-cons true stream-empty))))))))
```

The following diagram gives a visual picture of what the output should look like relative to the input.



The following diagram shows one implementation of a pulser. The rectangular device with output labeled y is a delay; the triangle-with-circle represents not, and the squared-off-semi-circle on the right represents and.



We want to turn this picture into a function *single-pulser* that takes a stream of inputs and produces the stream of outputs for this device. Write the body of *single-pulser* using our stream operator functions for gates and delays. We'll assume the delay initializes to false in the first step.

;; single-pulser : stream[bool] $\to$ stream[bool]

```
;; produces the output of a single pulser on the input
(define (single-pulser input)
  (stream-and (stream-not input)
              (stream-delay input false)))
```

If we test this on the sample input given above, we'd see that we were implementing the single pulser as expected.

Using similar ideas, we could code up simulators for a wide range of hardware devices. Streams are a natural data structure for implementing hardware devices because these devices run over an unspecified period of time, reading new data on demand at each tick of the system clock. The on-demand nature lets us delay the computation until the next clock tick, as happens in a real hardware system.

4 Summary

This concludes our coverage of streams. This example helps illustrate the impact that language design has on programming: by defining our stream operators to resemble list operators (in name, but more importantly in contracts), you were able to write stream programs easily, almost pretending that you were writing regular list functions (though note this would break down in some cases, for example if I asked you to do something like write *sum* on an infinite stream). Had we chosen very different names for the operators, you'd have found stream programming a bit more difficult. Never underestimate the value of well-chosen operators when you create a language!

For the material on streams, I expect that you

- know the difference between streams and lists, including when to use each in an implementation,
- can write programs with streams, and
- can explain the definitions of the stream operators from these lectures (i.e., why does *stream-cons* use **lambda**, etc).