CS403: Programming Languages

Yet More on Streams

Combining streams

An interesting example in the book concerns the generation of all possible pairings of the items in two streams, modulo commutativity. Below is a table which shows all the pairings up to element 5. The first number in the pair is the index of the first stream and the second number in the pair is the index of the item in the second stream.

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
		(2,2)	(2,3)	(2,4)	(2,5)
			(3,3)	(3,4)	(3,5)
				(4,4)	(4,5)
					(5,5)

Here is the pairing procedure, annotated as to which parts of the procedure generate which parts of the table:

```
(define (pair s1 s2)
    (cons-stream
        (list (stream-car s1) (stream-car s2))       ;top-left corner
        (interleave
            (stream-map                              ;the rest of the top row
                (lambda (x) (list (stream-car s1) x))
                (stream-cdr s2))
            (pairs (stream-cdr s1) (stream-cdr s2))  ;the rest of the table
            )
        )
    )
```

How the top-left corner and the rest of that row are generated should be readily understandable. Consider now the rest of the table. Note how it looks just like the original table; we can generated it with a simple recursive call, moving to the second elements in both streams.

Streams and the concept of time

Suppose Scheme has a function named rand-update which takes a number and returns its transmorgification. If there isn't much of an observed relationship between the original and the transmorgification, why you've got yourself a handy random number generator. Here's a constructor for a (repeatable) random number generator. Given the same seed, it produces the same sequence of random numbers. Given different streams it should give a radically different stream.

```
(define (random-generator seed)
    (define (dispatch msg . args)
        (cond
            ((eq? msg 'get)
                (set! seed (rand-update seed))
                seed
                )
            ((eq? msg 'set)
                (set! seed (car args))
                (dispatch 'get)
                )
            )
        )
    dispatch
    )
```

What happens if we try to generate a stream of random numbers. Here's such a generator.

```
(define (random-stream seed)
    (cons-stream
        (rand-update seed)
        (random-stream (rand-update seed))
        )
    )
```

To access successive random numbers, we pass successive indices to stream-ref, as in

```
(define rs (random-stream 123))
(stream-ref rs 0)
(stream-ref rs 1)
(stream-ref rs 2)
...
```

The remarkable thing is that streams work functionally, yet can accomplish that same things that are usually done imperatively. In fact, streams are often used in purely functional languages in lieu of state.