CS403: Programming Languages

More on Streams

The Sieve of Eratosthenes

The Sieve of Eratosthenes is easily implemented as repeated filtering of a stream. The assumption is that the first number in the stream is prime.

```
    (define (sieve s)
        (cons-stream
            (stream-car s)
            (sieve
                (stream-remove
                    (lambda (x) (integer? (/ x (stream-car s))))
                    (stream-cdr s))
                )
            )
        )
```

where *stream-remove* removes those elements for which the given predicate is true (in the same vein as *stream-filter*, but I can never remember whether *filter* is supposed to keep or remove). An infinite stream of primes is now defined as

```
    (define primes (seive (stream-cdr integers)))
```

Interleaving streams

Consider combining the stream odds with the streams evens to produce a stream of integers (we will assume that odds begins with 1 and evens begins with 2). Suppose we placed all the evens after all the odds. This works for finite streams (though the integers aren't not in there normal order) but this cannot work for infinite streams. In this case, we will never see an even integer (even though, technically, they are still there). Here's a routine which *interleaves* two infinite streams so that any particular item in the stream can be accessed in a finite amount of time.

```
    (define (interleave s1 s2)
        (cons-stream
            (stream-car s1)
            (interleave s2 (stream-cdr s1))
            )
        )
```

Note the the arguments in the recursive call are swapped so that the stream whose item was not chosen is in line to have an item selected the next time. Now we can generate our integers.

```
    (define integers (interleave odds evens))
```

What happens if we define our integers this way?

```
    (define integers (interleave evens odds))
```

The evens and odds are well distributed, although 2 comes before 1. Is there a way to fix interleave, in a generic sort of way, so that we can force the interleaved streams out in the proper order, no matter which stream comes first in the argument list? Yes, by adding an ordering predicate.

```
    (define (interleave s1 s2 order)
        (if (order (stream-car s1) (stream-car s2))
            (cons-stream
                (stream-car s1)
                (interleave (stream-cdr s1) s2)
                )
            (cons-stream
                (stream-car s2)
                (interleave s1 (stream-cdr s2))
                )
            )
        )
```

Now, the following definitions are equivalent

```
    (define integers (interleave evens odds <))
    (define integers (interleave odds evens <))
```

Still more on streams

The textbook gives a nice example of working with streams. The idea is to approximate π by generating a stream of the form:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{5} + \frac{1}{7} - \frac{1}{9} \ldots$$

It turns out that the summation of all the terms of this stream is equal to $\frac{\pi}{4}$. First let's generate the initial stream. Here's a function which generates such a stream.

```
    (define (pi-term-generator a b)
        (cons-stream
            (/ a b)
            (pi-term-generator (- a) (+ b 2.0))
            )
        )
```

We define terms to be

```
    (define terms (pi-term-generator 1 1))
```

Now it would be nice to sum the stream, but we can't since it is infinite. However, we can sum the first n terms. It will be convenient to express all of the possible partial sums as a stream, so that the n^{th} item in the resulting stream is the sum of items 1 through n in the original stream. Here is one such definition...

```
    (define (partial-sum s)
        (cons-stream
            (stream-car s)
            (add-stream (partial-sum s) (stream-cdr s)))
            )
        )
```

Given a definition of scale-stream...

```
    (define (scale-stream s x)
        (cons-stream
            (* (stream-car s) x)
            (scale-stream (stream-cdr s) x)
            )
        )
```

We can define π as

```
    (define pi (scale-stream (partial-sum (pi-terms 1 1)) 4))
```

As we discovered before, this series for determining π converges very slowly. However, Euler came up with a neat trick for accelerating the convergence. Given a stream s, whose partial sum converges to some value, the following routine produces an accelerated stream which converges to the same value, only more rapidly.

```
    (define (accelerate s)
        (define Sn-1 (stream-ref s 0))
        (define Sn (stream-ref s 1))
        (define Sn+1 (stream-ref s 2))
        (cons-stream
            (- Sn+1 (/ (square (- Sn+1 Sn)) (+ Sn-1 (* -2 Sn) Sn+1)))
            (accelerate (stream-cdr s))
            )
        )
```

Of course, what's to stop us from accelerating the accelerated stream? Nothing, but why stop there? Why don't we accelerate infinitely many times? Here is a function which makes an infinite stream of infinite streams. The first stream is the original, the next is the accelerated stream, the third stream is the acceleration of the accelerated stream and so on. The book calls this structure a tableau.

```
    (define (tableau s xcl)
        (cons-stream
            s
            (tableau (xcl s) xcl)
            )
        )
```

Now consider the first term of each stream in the tableau. Each of these terms successively approaches the convergence value at an astonishing rate. Here is the final stream for approximations to π.

```
    (define api (stream-map stream-car (tableau pi accelerate)))
```

The name *api* is mnemonic for accelerated π stream. After eight terms, the change in the convergence value is too small and outstrips the precision of the interpreter. In contrast, the original π stream converges to an equivalent value after 10^{13} terms!