CS403: Programming Languages

Notes on Streams

Consider a list of n factorials, where the i^{th} element of the list is $i!$. Here's one way to construct such a list.

```
(define (fs n)
    (define (iter count)
        (if (> count n)
            '()
            (cons (factorial count) (iter (+ count 1)))
            )
        )
    (iter 1)
    )
```

The function *fs*, which stands for *factorial stream*, generates a list of factorials from 1! to n!. The version generates n factorials and places them in the list. Now what happens if we end up wanting only the third factorial in that list, as in

```
(define p (fs 100))
(display (caddr p))
```

We will have generated n-3 factorials that we ended up not using. Here is a version which generates only the factorials we need. This seems like an impossible task. We are told to generate n numbers and at some future date we will be told how many of those numbers will actually be needed. To generate just the number we need seems to require the ability to tell the future! Since functions are first class objects, we don't need to tell the future at all (although we will require users of our list of factorials to use special access functions). Here's a new definition accomplishes our goal.

```
(define (dfs n)
    (define (iter count)
        (if (&gt; count n)
            '()
            (cons (factorial count) (lambda ()(iter (+ count 1))))
            )
        )
    (iter 1)
    )
```

The name *dfs* is mnemonic for *delayed factorial stream* because the *cdr* of the stream is delayed. Note that we have wrapped the recursive call to *iter* in a lambda function thus preventing evaluation of the recursive call. Thus `(dfs n)` returns a single cons object (the *car* points to the result of the call to `(factorial 1)` and the *cdr* points to the lambda function), regardless of how large n gets. Now we define our special access functions.

```
(define (h s) (car s))
(define (t s) ((cdr s)))
```

The function *h*, which is mnemonic for head, retrieves the *car* of the cons object. The function *t*, which is mnemonic for tail, evaluates the *cdr* of the cons object. It can't simply return the *cdr* since the *cdr* has been wrapped in a lambda. Evaluating the *cdr* strips the lambda. Now to display the first, second, and third numbers in the stream, we evaluate these expressions

```
(define p (dfs 100))
(display (h t))
(display (h (t p)))
(display (h (t (t p))))
```

Every time we ask for the tail of the stream, the lambda is evaluated which generates a new cons object, whose *car* points to the next number in the stream and whose *cdr* points to a wrapped recursive call to generate the remainder of the stream. Now that we have a delayed stream, if we ask for the third element, only the first three elements are ever generated. Note that now the upper limit on the stream is now moot. The function *dfs* can be rewritten as:

```
(define (dfs)
    (define (iter count)
        (cons (factorial count)
            (lambda () (iter (+ count 1))))
        )
    (iter 1)
    )
```

It turns out that there is a built-in special form called *delay* which uses this wrapping technique. To use *delay*, the above code would be simply rewritten as

```
(define (dfs)
    (define (iter count)
        (cons (factorial count) (delay (iter (+ count 1))))
        )
    (iter 1)
    )
```

Furthermore, there is a special version of *cons* which delays its second argument. Using this special form yields

```
(define (dfs)
    (define (iter count)
        (cons-stream (factorial count) (iter (+ count 1)))
        )
    (iter 1)
    )
```

The system versions of *h* and *t* are *stream-car* and *stream-cdr* respectively.

Consider now an infinite stream of ones. Here is such a definition

```
(define ones
    (cons-stream 1 ones))
```

and here is a definition of the positive integers

```
(define positive-integers
    (cons-stream 1 (add-stream ones integers)))
```

The function *add-stream* takes two stream arguments and generates a new stream where the i^{th} item in the new stream is the sum of the i^{th} items in the original stream. Here is a definition of *add-stream*.

```
(define (add-stream s1 s2)
    (cons-stream
        (+ (stream-car s1) (stream-car s2))
        (add-stream (stream-cdr s1) (stream-cdr s2))
        )
    )
```

There is a system definition which makes add-stream much simpler:

```
(define (add-stream s1 s2)
    (stream-map + s1 s2)
    )
```

assuming *stream-map* is implemented on your system.