Lecture 9

-Can define a ‘stream’ in lambda calculus

-Does not calculate value until needed (is like a lazy list) eg. a pipe

-Stream is a function, if never evoked, it’s never evaluated

**s-car**: gives first value

**s-cdr**: gives rest of stream

**s-cons**: needs to take in uncomputed values

**Definition:** “Thunk” - (lambda () EXPR) is a function that returns the value EXPR

**Example 1: (List-to-stream)**

Takes list and converts to stream.

If null, return empty

else take the first element of the list, call s-cons, then call list-to-stream on rest of list

**Example 2: (s-take)**

Takes integer n and stream s

returns first n elements in stream

if n=0 return empty list or null

**Example 3: (s-nat-from-n)**

Creates stream of all natural numbers starting at n

*[program crashed] - Need to add laziness to second element*

*[fix] - wrap both values in a thunk (n and n+1 functions)*

**Example 4: (s-map)**

Maps f over all the values in stream

*[edit1] wrap first element in cons\* in thunk so not always evaluated*

*[edit2] wrap all of the ‘if’ in thunk so it returns a stream*

The example maps 1/x across the stream. Error occurs on s-car of s-map.Why?

**Example 5: (s-zip-with)**

Can use map on two streams

ie (s-take 10 (s-zip-with \* s-nat s-nat)) - returns first 10 elements of (s-nat)\*(s-nat)

**Example 6: (s-const)**

*[New Idea : putting a (define) inside a lambda expression]*

Makes a stream of constant (input)

e.g. (s-take 10 (s-const 7)) = (7 7 7 7 7 7 7 7 7 7)

**Example 7: Fibonacci Stream**

Idea: summing 2 streams

[0 1 1 …]

+[1 1 2 …]

-----------------

[1 2 3 5 …]

*[Slow] This function is slow to evaluate because no caching, always re-evaluating.*

*[Fix] Use delay/force procedures*