CS 360: Programming Languages
Lecture 4: Streams

Geoffrey Mainland

Drexel University

Section 1

Programming With Streams

Issues with lists

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
           (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))

(define (sum-primes a b)
  (reduce + 0 (filter prime? (enumerate-interval a b))))
```

- How much memory does the first program need?
- How about the second?

The cost of the list abstraction

Tools like `map`, `filter`, and `reduce` are great for abstraction, but they can be expensive—composing these constructs requires creating potentially huge intermediate lists at each step of the computation.

Even worse...

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

Streams

- Streams are sequences that are constructed partially.
- If a stream consumer tries to access part of the stream that is not yet constructed, the stream will automatically construct just enough of itself to produce the required part.
- This should remind you of normal order evaluation—only what is needed is evaluated.

Introduction to Streams

- Streams are like lists but with different names for the functions that manipulate them.
- We have `stream-cons`, `stream-first`, and `stream-rest` for constructing and destructing (tearing apart) streams.
- There is a constant `empty-stream`, the stream analogue to `null`, and a predicate `stream-empty?` for testing for the empty stream.
- The stream analogue of the `list` function is `stream`.
- Let's write the stream analogues of higher-order functions we've already seen in the context of lists.

Summary: streams

- Streams are delayed lists.
- The stream abstraction is built on lower-level abstractions `delay` and `force`.
- These abstractions allow us to only perform as much computation as is necessary to get our desired result.
- Streams recover efficiency while allowing us to compose abstractions like `map`, `filter`, and `reduce`.

Lab 1: More Scheme

- You may work with 1 partner.
- You **must acknowledge your partner**. Let me know who you worked with in your READEME.md.
- You must both turn in the assignment by pushing to GitHub. You may turn in the same or identical code.
- **For labs you may work with a partner. For homeworks you must work alone.**