UMassAmherst | Manning College of Information & Computer Sciences

Programming Methodology

Lab 7: Streams

Wednesday, October 16, 2024

Weekly Lab Agenda

- Go over reminders/goals
- Review past material
- Work in groups of 2-3 to solve a few exercises
    - Please sit with your group from last week.
- Discussion leaders will walk around and answer questions
- Solutions to exercises will be reviewed as a class
- Attendance taken at the end

Reminders

- Download and open the starter code
- Homework 4b is due tonight at 11:59pm
    - Come to office hours for help!
- Homework 5 will be released

Today's Goals

- Practice working with streams

Memoization

The idea behind memoization is that when a function is making too many repetitive computations, store the results and look at them later.

- Create a memo object by passing the computation (closure f, not executed)
- When needed, use the memo object: only the first call to the get() method will run the function f
- The result is stored for later in val
- Later calls use the stored result
- toString(): let's us view the object without evaluating it

```
interface Memoized<T> {
  get: () => T;
  toString: () => string;
}

/**
 * Memoizes a provided function
 * @param f A zero-parameter function (thunk)
 * @returns A memoized version of @param f
 */
function memo<T>(f: () => T): Memoized<T> {
  let evaluated = false;
  let value: T;
  return {
    get: () => {
      if (!evaluated) {
        value = f();
        evaluated = true;
      }

      return value;
    },
    toString: () => (evaluated ? String(value): "<unevaluated>"),
  };
}
```

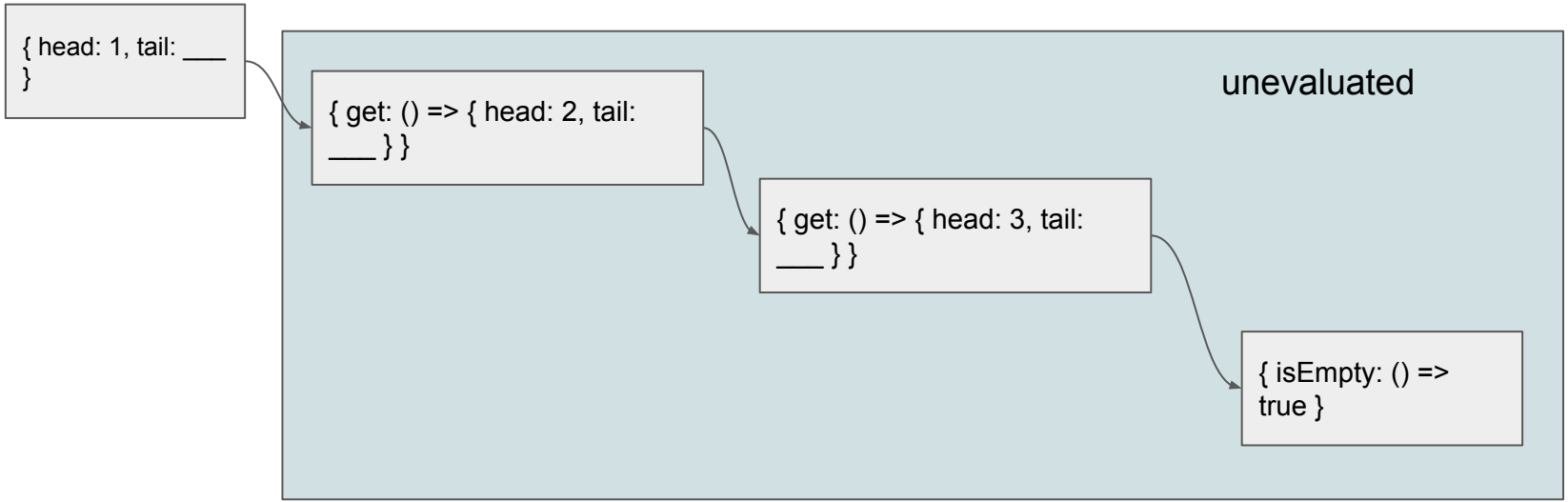
Memoization in Streams

UMassAmherst | Manning College of Information & Computer Sciences

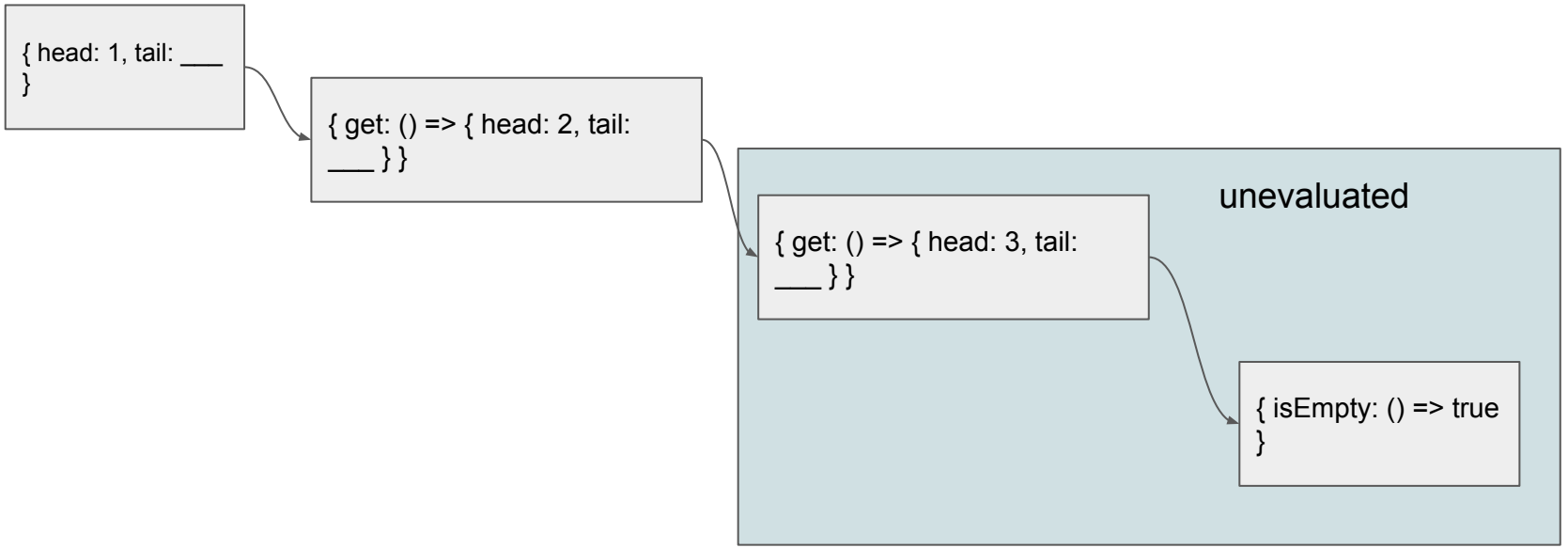
- If we take a list and we memoize the tail, then we get a stream
- The nodes get created as they are accessed during execution of the program
- This is because tail is actually a function that is not executed when the stream is first created
- Streams could be potentially infinite - with only a finite portion getting instantiated during execution

```
interface Stream<T> {
  head: () => T;
  tail: () => Stream<T>;
  isEmpty: () => boolean;
  toString: () => string;
  map: <U>(f: (x: T) => U) => Stream<U>;
  filter: (f: (x: T) => boolean) => Stream<T>;
}

export function sempty<T>(): Stream<T> {
    return {
        head: () => {
        throw new Error();
        },
        tail: () => {
        throw new Error();
        },
        isEmpty: () => true,
        toString: () => "sempty",
        map: () => sempty(),
        filter: () => sempty(),
    };
}

export function snode<T>(head: T, tail: () => Stream<T>): Stream<T> {
    const memoizedTail = memo(tail);

    return {
      isEmpty: () => false,
      head: () => head,
      tail: memoizedTail.get,
      toString: () => `snode(${head}, ${tail.toString()})`,
      map: f => snode(f(head), () => memoizedTail.get().map(f)),
      filter: f => (f(head) ? snode(head, () => memoizedTail.get().filter(f))
                            : memoizedTail.get().filter(f)),
    };
}
```

```
stream = snode(1, () => snode(2, () => snode(3, () => sempty())));
assert(stream.head() === 1);
assert(stream.toString() === "snode(1, <unevaluated>)");
```



```
stream = snode(1, () => snode(2, () => snode(3, () => sempty())));
assert(stream.tail().head() === 2);
assert(stream.toString() === "snode(1, snode(2, <unevaluated>))");
```



Problem 1: Creating streams

Create an infinite stream of non-negative integers that are multiples of 3. Do not use higher-order functions.

HINT 1:

```
function every3(n: number): Stream<number> {
    // always define a function with recursive structure
}
const mult3stream = every3(0); // call to create stream
console.assert(mult3stream.tail().tail().head() === 6);
```

HINT 2:

```
function from(start: number, step = 1): Stream<number> {
    return snode(start, () => from(start + step, step));
}
```

Problem 2: Filtering streams

Write a function that takes as argument a stream of numbers and returns a stream of those numbers which are multiples of their predecessor in the original stream.

E.g.: Original stream: 1, 2, 3, 6, 7, 14, ... => Output: 2, 6, 14

Original stream: 1, 2, 3, 4, 5, ... => Output: 2

Problem 3

Write a function that takes two streams (finite or infinite) and returns a single stream with interleaves the values from each of the inputs.
ie. take one element from the first stream, then one from the second, then the next from the first, etc. If one of the streams ends, continue with just the other one.
Example:
stream1 (first four squares) : [1, 4, 9, 16]
stream2 (natural numbers from 1) : [1, 2, 3 ,4, ...]
Output stream: [1, 1, 4, 2, 9, 3, 16, 4, 5, 6, ...]

Extra Practice

- Fall 2020 Final
- Fall 2020 final, with solutions
- Practice Worksheet for Streams
- Final Exam, Spring 2021

Look in canvas for a practice worksheet for streams

Bonus Problem 1

- Create an infinite stream of square of natural numbers

```
function sqstream (start: number): Stream<number> {


}
const stream = sqstream(1)
console.log(stream.tail().head())
console.log(stream.tail().toString())
```