B9: Streams I

CS1101S: Programming Methodology

Boyd Anderson

October 15, 2021

Module Overview

- **Unit 1 — Functions** (textbook Chapter 1)
  - Getting acquainted with the elements of programming, using **functional abstraction**
  - Learning to read programs, and using the **substitution model**
  - Example applications: runes, curves
- **Unit 2 — Data** (textbook Chapter 2)
  - Getting familiar with data: pairs, lists, trees
  - Searching in lists and trees, sorting of lists
  - Example application: sound processing

Module Overview

- **Unit 3 — State** (parts of textbook Chapter 3)
  - Programming with **stateful abstractions**
  - **Mutable data** processing
  - Arrays, loops, searching in and sorting of arrays
  - Reading programs using the **environment model**
  - Example application: robotics, video processing

Module Overview

- **Unit 3 — State** (parts of textbook Chapter 3)
  - Programming with **stateful abstractions**
  - **Mutable data** processing
  - Arrays, loops, searching in and sorting of arrays
  - Reading programs using the **environment model**
  - Example application: robotics, video processing

- **Unit 4 — Beyond** (parts of textbook Chapters 3 and 4)
  - Streams
  - Understanding the environment model by *programming it*

Readings

- Textbook Sec. 3.5

Outline

- Motivation
- Streams
- More Examples

The Basics

- **Representing conditionals**

The Basics

- **Representing conditionals**

  `E1 ? E2 : E3` can be represented using function:

The Basics

- **Representing conditionals**

  `E1 ? E2 : E3` can be represented using function:

  What about: `cond(E1, E2, E3)`

The Basics

- **Representing conditionals**

  `E1 ? E2 : E3` can be represented using function:

  What about: `cond(E1, E2, E3)`

  ```
  cond(E1, () => E2, () => E3)
  ```

The Basics

- **Representing conditionals**

  `E1 ? E2 : E3` can be represented using function:

  What about: `cond(E1, E2, E3)`

  ```
  cond(E1, () => E2, () => E3)
  ```

  where

  ```
  function cond(x, y, z) {
      if (x) { return y(); } else { return z(); }
  }
  ```

Delayed Evaluation

```
cond(E1, () => E2, () => E3)
```

Delayed Evaluation

```
cond(E1, () => E2, () => E3)
```

- **Main idea**

Delayed Evaluation

```
cond(E1, () => E2, () => E3)
```

- **Main idea**
  - We **delayed the evaluation** of `E2` and `E3` until we had enough information to decide which one was needed

Delayed Evaluation

```
cond(E1, () => E2, () => E3)
```

- **Main idea**
  - We **delayed the evaluation** of `E2` and `E3` until we had enough information to decide which one was needed
- **Instrument of delay**

Delayed Evaluation

```
cond(E1, () => E2, () => E3)
```

- **Main idea**
  - We **delayed the evaluation** of `E2` and `E3` until we had enough information to decide which one was needed
- **Instrument of delay**
  - **Functions** allow us to describe an activity without actually doing the activity

A Simple Example

```
function f(x) {
    return () => x + 1;
}
```

A Simple Example

```
function f(x) {
    return () => x + 1;
}


// returns a function that stores computation
const y = f(99);
```

A Simple Example

```
function f(x) {
    return () => x + 1;
}


// returns a function that stores computation
const y = f(99);

// two weeks later
let z = y();
```

Sum of Primes

```
// returns the sum of all prime
// numbers in the range [a, b].

function sum_primes(a, b) {
    function iter(count, accum) {
        if (count > b) {
            return accum;
        } else if (is_prime(count)) {
            return iter(count + 1, count + accum);
        } else {
            return iter(count + 1, accum);
        }
    }
    return iter(a, 0);
}
```

Show in Playground

Sum of Primes, Too

```
// returns the sum of all prime
// numbers in the range [a, b].

function sum_primes(a, b) {
    return accumulate(
        (x, y) => x + y,
        0,
        filter(is_prime, enum_list(a, b))
    );
}
```

Show in Playground

Extreme Example

```
head(tail(filter(
    is_prime,
    enum_list(10000, 1000000)
)));
```

Show in Playground

Extreme Example

```
head(tail(filter(
    is_prime,
    enum_list(10000, 1000000)
)));
```

Show in Playground

- **What is wrong here?**

Extreme Example

```
head(tail(filter(
    is_prime,
    enum_list(10000, 1000000)
)));
```

Show in Playground

- **What is wrong here?**
  - We only want the **second** prime number out of the list of **990,001** numbers!

Outline

- Motivation
- Streams
- More Examples

Idea of Streams

Idea of Streams

- **Delayed lists**

Idea of Streams

- **Delayed lists**
  - Our pairs contain a **data item** as **head** (as usual), but a **function as tail** that can be activated when needed

Idea of Streams

- **Delayed lists**
  - Our pairs contain a **data item** as **head** (as usual), but a **function as tail** that can be activated when needed
- **Streams**

Idea of Streams

- **Delayed lists**
  - Our pairs contain a **data item** as **head** (as usual), but a **function as tail** that can be activated when needed
- **Streams**
  - A ***stream*** is either the **empty list**, or a **pair** whose **tail** is a **nullary function** that returns a stream

Idea of Streams

- **Delayed lists**
  - Our pairs contain a **data item** as **head** (as usual), but a **function as tail** that can be activated when needed
- **Streams**
  - A ***stream*** is either the **empty list**, or a **pair** whose **tail** is a **nullary function** that returns a stream
- **Stream discipline**

Idea of Streams

- **Delayed lists**
  - Our pairs contain a **data item** as **head** (as usual), but a **function as tail** that can be activated when needed
- **Streams**
  - A ***stream*** is either the **empty list**, or a **pair** whose **tail** is a **nullary function** that returns a stream
- **Stream discipline**
  - Like list discipline, now using streams

Example Streams

Example Streams

```
// An empty stream
const s1 = null;
```

Example Streams

```
// An empty stream
const s1 = null;

// A stream with element 1
const s2 = pair(1, () => null);
```

Example Streams

```
// An empty stream
const s1 = null;

// A stream with element 1
const s2 = pair(1, () => null);

// A stream with elements 1, 2, 3
const s3 =
    pair(1,
         () => pair(2,
                    () => pair(3,
                               () => null)));
```

An “Infinite” Stream

```
function ones_stream() {
    return pair(1, ones_stream);
}
```

Show in Playground

An “Infinite” Stream

```
function ones_stream() {
    return pair(1, ones_stream);
}

const ones = ones_stream();
```

Show in Playground

An “Infinite” Stream

```
function ones_stream() {
    return pair(1, ones_stream);
}

const ones = ones_stream();

head(ones); ➔ 1
```

Show in Playground

An “Infinite” Stream

```
function ones_stream() {
    return pair(1, ones_stream);
}

const ones = ones_stream();

head(ones); ➔ 1

head(tail(ones)()); ➔ 1
```

Show in Playground

An “Infinite” Stream

```
function ones_stream() {
    return pair(1, ones_stream);
}

const ones = ones_stream();

head(ones); ➔ 1

head(tail(ones)()); ➔ 1

head(tail(tail(ones)())()); ➔ 1
```

Show in Playground

A Convenient Function

```
function stream_tail(stream) {
    return tail(stream)();
}
```

Show in Playground

A Convenient Function

```
function stream_tail(stream) {
    return tail(stream)();
}

const ones = ones_stream();
```

Show in Playground

A Convenient Function

```
function stream_tail(stream) {
    return tail(stream)();
}

const ones = ones_stream();

head(ones); ➔ 1
```

Show in Playground

A Convenient Function

```
function stream_tail(stream) {
    return tail(stream)();
}

const ones = ones_stream();

head(ones); ➔ 1

head(stream_tail(ones)); ➔ 1
```

Show in Playground

A Convenient Function

```
function stream_tail(stream) {
    return tail(stream)();
}

const ones = ones_stream();

head(ones); ➔ 1

head(stream_tail(ones)); ➔ 1

head(stream_tail(stream_tail(ones))); ➔ 1
```

Show in Playground

Streams Are Lazy Lists

```
function enum_stream(low, hi) {
    return low > hi
        ? null
        : pair(low,
               () => enum_stream(low + 1, hi));
}
```

Show in Playground

Streams Are Lazy Lists

```
function enum_stream(low, hi) {
    return low > hi
        ? null
        : pair(low,
               () => enum_stream(low + 1, hi));
}

const s = enum_stream(1, 100);
```

Show in Playground

Streams Are Lazy Lists

```
function enum_stream(low, hi) {
    return low > hi
        ? null
        : pair(low,
               () => enum_stream(low + 1, hi));
}

const s = enum_stream(1, 100);

head(s); ➔ 1
```

Show in Playground

Streams Are Lazy Lists

```
function enum_stream(low, hi) {
    return low > hi
        ? null
        : pair(low,
               () => enum_stream(low + 1, hi));
}

const s = enum_stream(1, 100);

head(s); ➔ 1

head(stream_tail(s)); ➔ 2
```

Show in Playground

Streams Are Lazy Lists

```
function enum_stream(low, hi) {
    return low > hi
        ? null
        : pair(low,
               () => enum_stream(low + 1, hi));
}

const s = enum_stream(1, 100);

head(s); ➔ 1

head(stream_tail(s)); ➔ 2

head(stream_tail(stream_tail(s))); ➔ 3
```

Show in Playground

A Useful Function

```
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}
```

Show in Playground

A Useful Function

```
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}

const s = enum_stream(1, 100);
```

Show in Playground

A Useful Function

```
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}

const s = enum_stream(1, 100);

stream_ref(s, 0); ➔ 1
```

Show in Playground

A Useful Function

```
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}

const s = enum_stream(1, 100);

stream_ref(s, 0); ➔ 1

stream_ref(s, 10); ➔ 11
```

Show in Playground

A Useful Function

```
function stream_ref(s, n) {
    return n === 0
        ? head(s)
        : stream_ref(stream_tail(s), n - 1);
}

const s = enum_stream(1, 100);

stream_ref(s, 0); ➔ 1

stream_ref(s, 10); ➔ 11

stream_ref(s, 99); ➔ 100
```

Show in Playground

More Useful Functions

Show in Playground

More Useful Functions

```
function stream_map(f, s) {
    return is_null(s)
        ? null
        : pair(f(head(s)),
               () => stream_map(f, stream_tail(s)));
}
```

Show in Playground

More Useful Functions

```
function stream_map(f, s) {
    return is_null(s)
        ? null
        : pair(f(head(s)),
               () => stream_map(f, stream_tail(s)));
}

function stream_filter(p, s) {
    return is_null(s)
        ? null
        : p(head(s))
          ? pair(head(s),
                 () => stream_filter(p, stream_tail(s)))
          : stream_filter(p, stream_tail(s));
}
```

Show in Playground

Back to the Extreme Example

```
head(stream_tail(stream_filter(
    is_prime,
    enum_stream(10000, 1000000)
)));
```

Show in Playground

Back to the Extreme Example

```
head(stream_tail(stream_filter(
    is_prime,
    enum_stream(10000, 1000000)
)));
```

Show in Playground

- **General idea**

Back to the Extreme Example

```
head(stream_tail(stream_filter(
    is_prime,
    enum_stream(10000, 1000000)
)));
```

Show in Playground

- **General idea**
  - Only compute what is needed

Back to the Extreme Example

```
head(stream_tail(stream_filter(
    is_prime,
    enum_stream(10000, 1000000)
)));
```

Show in Playground

- **General idea**
  - Only compute what is needed
  - **Be lazy!**

Outline

- Motivation
- Streams
- More Examples

More Examples

```
function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}
```

Show in Playground

More Examples

```
function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}

const integers = integers_from(1);
```

Show in Playground

More Examples

```
function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}

const integers = integers_from(1);

stream_ref(integers, 0); ➔ 1
```

Show in Playground

More Examples

```
function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}

const integers = integers_from(1);

stream_ref(integers, 0); ➔ 1

stream_ref(integers, 10); ➔ 11
```

Show in Playground

More Examples

```
function integers_from(n) {
    return pair(n, () => integers_from(n + 1));
}

const integers = integers_from(1);

stream_ref(integers, 0); ➔ 1

stream_ref(integers, 10); ➔ 11

stream_ref(integers, 99); ➔ 100
```

Show in Playground

More Examples

```
function is_divisible(x, y) {
    return x % y === 0;
}
```

Show in Playground

More Examples

```
function is_divisible(x, y) {
    return x % y === 0;
}

const no_fours =
    stream_filter(
        x => !is_divisible(x, 4),
        integers
    );
```

Show in Playground

More Examples

```
function is_divisible(x, y) {
    return x % y === 0;
}

const no_fours =
    stream_filter(
        x => !is_divisible(x, 4),
        integers
    );

stream_ref(no_fours, 3); ➔ 5
```

Show in Playground

More Examples

```
function is_divisible(x, y) {
    return x % y === 0;
}

const no_fours =
    stream_filter(
        x => !is_divisible(x, 4),
        integers
    );

stream_ref(no_fours, 3); ➔ 5

stream_ref(no_fours, 100); ➔ 134
```

Show in Playground

From Streams to Lists

```
function eval_stream(s, n) {
    return n === 0
        ? null
        : pair(head(s),
               eval_stream(stream_tail(s), n - 1));
}
```

Show in Playground

From Streams to Lists

```
function eval_stream(s, n) {
    return n === 0
        ? null
        : pair(head(s),
               eval_stream(stream_tail(s), n - 1));
}


eval_stream(no_fours, 10);
```

Show in Playground

From Streams to Lists

```
function eval_stream(s, n) {
    return n === 0
        ? null
        : pair(head(s),
               eval_stream(stream_tail(s), n - 1));
}


eval_stream(no_fours, 10);
// [1, [2, [3, [5, [6, [7, [9, [10, [11, [13,
// ]]]]]]]]]]
```

Show in Playground

Fibonacci Numbers

Show in Playground

Fibonacci Numbers

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}
```

Show in Playground

Fibonacci Numbers

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}


const fibs = fibgen(0, 1);
```

Show in Playground

Fibonacci Numbers

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}


const fibs = fibgen(0, 1);

eval_stream(fibs, 10);
```

Show in Playground

Fibonacci Numbers

```
function fibgen(a, b) {
    return pair(a, () => fibgen(b, a + b));
}


const fibs = fibgen(0, 1);

eval_stream(fibs, 10);
// [0, [1, [1, [2, [3, [5, [8, [13, [21, [34,
// [][][][][][][][][][]]
```

Show in Playground

More and More…

Show in Playground

More and More…

- **Wanted:** Stream containing 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

Show in Playground

More and More…

- **Wanted:** Stream containing 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

```
function more(a, b) {
    return (a > b)
        ? more(1, 1 + b)
        : pair(a, () => more(a + 1, b));
}
```

Show in Playground

More and More…

- **Wanted:** Stream containing 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

```
function more(a, b) {
    return (a > b)
        ? more(1, 1 + b)
        : pair(a, () => more(a + 1, b));
}

const more_and_more = more(1, 1);
```

Show in Playground

More and More…

- **Wanted:** Stream containing 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

```
function more(a, b) {
    return (a > b)
        ? more(1, 1 + b)
        : pair(a, () => more(a + 1, b));
}

const more_and_more = more(1, 1);

eval_stream(more_and_more, 15);
```

Show in Playground

More and More…

- **Wanted:** Stream containing 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, …

```
function more(a, b) {
    return (a > b)
        ? more(1, 1 + b)
        : pair(a, () => more(a + 1, b));
}

const more_and_more = more(1, 1);

eval_stream(more_and_more, 15);
// [1, [1, [2, [1, [2, [3, [1, [2, [3, [4,
// [1, [2, [3, [4, [5, []]]]]]]]]]]]]]]]
```

Show in Playground

Stream Processing

Stream Processing

- **Like lists, except**

Stream Processing

- **Like lists, except**
  - Wrap **tail** in a **function**

Stream Processing

- **Like lists, except**
  - Wrap **tail** in a **function**
  - Use `stream_tail` instead of `tail`

Stream Processing

- **Like lists, except**
  - Wrap **tail** in a **function**
  - Use `stream_tail` instead of `tail`

- **Stream support in Source §3**
  - STREAMS provides pre-declared functions for stream processing
    - Examples: `stream_tail`, `stream_map`, `stream_filter`

Stream Processing

- **Like lists, except**
  - Wrap **tail** in a **function**
  - Use `stream_tail` instead of `tail`

- **Stream support in Source §3**
  - STREAMS provides pre-declared functions for stream processing
    - Examples: `stream_tail`, `stream_map`, `stream_filter`
  - Refer to documentation here

Stream Processing

- **Like lists, except**
  - Wrap **tail** in a **function**
  - Use `stream_tail` instead of `tail`

- **Stream support in Source §3**
  - STREAMS provides pre-declared functions for stream processing
    - Examples: `stream_tail`, `stream_map`, `stream_filter`
  - Refer to documentation here
  - Source specification for Source§3 here

Summary

Summary

- In streams, functions serve as pickles
  - Open the jar (apply the function) only when you need them

Summary

- In streams, functions serve as pickles
  - Open the jar (apply the function) only when you need them
- Streams can represent “infinite” data structures

Summary

- In streams, functions serve as pickles
  - Open the jar (apply the function) only when you need them
- Streams can represent “infinite” data structures
- Laziness of streams avoids problem of non-termination