Streams

Jia Xiaodong

October 25, 2021

Declarative Programming

Simply put, a declarative style means:

Declarative Programming

Simply put, a declarative style means:

- We declare what we the program wants to do.
- What to do, not how to do it.

Declarative Programming

Simply put, a declarative style means:

- We declare what we the program wants to do.
- What to do, not how to do it.

Example

```
SELECT * FROM students WHERE name="Bob"
sibling(X,Y) :- parent(Z,X), parent(Z,Y), X\==Y.
```

Being lazy

Short-circuiting

`false && big_calc && massive_calc` – we don't need to evaluate everything!

Being lazy

Short-circuiting

`false && big_calc && massive_calc` – we don't need to evaluate everything!

Copy On Write

If you make a copy of a file do we need to comply? Just keep one file, and only "really" make a copy when you start editing the copy!

Being lazy

Short-circuiting

`false && big_calc && massive_calc` – we don't need to evaluate everything!

Copy On Write

If you make a copy of a file do we need to comply? Just keep one file, and only "really" make a copy when you start editing the copy!

Lazy evaluation

Why must we evaluate a statement on assignment? Let's procrastinate until when it is really needed.

Application: Lists

The first element of $\mathbb{N}$

```
head(enum_list(0, Infinity));
```

Application: Lists

The first element of $\mathbb{N}$

```
head(enum_list(0, Infinity));
```

Some element of $\mathbb{N}$

```
list_ref((enum_list(0, BIGNUMBER), BIGNUMBER + 1);
```

Delaying Lists

Definition

A list is a pair whose head is of type `any` and whose tail is of type `list | null`.

Delaying Lists

Definition

A list is a pair whose head is of type `any` and whose tail is of type `list | null`.

Definition

A *lazy* list is a pair whose head is of type `any` and whose tail is of type `(() => lazy list) | (null)`.

Delaying Lists

Definition

A list is a pair whose head is of type `any` and whose tail is of type `list | null`.

Definition

A *lazy* list is a pair whose head is of type `any` and whose tail is of type `(() => lazy list) | (null)`.

Example

```
empty = null;
one = pair(1, () => 1);
thing = pair(1, () => thing); // recursive structures
```

Operations

Definition

A *lazy* list is a pair whose head is of type `any` and whose tail is of type `(() => lazy list) | (null)`.

Of course, we need to adapt our list operations to fit the lazy version:

Operations

Definition

A *lazy* list is a pair whose head is of type `any` and whose tail is of type `(() => lazy list) | (null)`.

Of course, we need to adapt our list operations to fit the lazy version:

- `head`: as per normal.

Operations

Definition

A *lazy* list is a pair whose head is of type `any` and whose tail is of type `(() => lazy list) | (null)`.

Of course, we need to adapt our list operations to fit the lazy version:

- `head`: as per normal.
- `tail`: Do a normal `tail`, then call the function `()`.

Operations: map

Eager map

```
function map(f, xs) {
    return is_null(xs) ? null
            : pair(f(head(xs)), map(f, tail(xs)));
}
```

Operations: map

Eager map

```
function map(f, xs) {
    return is_null(xs) ? null
             : pair(f(head(xs)), map(f, tail(xs)));
}
```

Lazy map

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

Operations: filter

Eager filter

```
function filter(pred, xs)
    return is_null(xs) ? xs : pred(head(xs))
           ? pair(head(xs), filter(pred, tail(xs)))
           : filter(pred, tail(xs));
```

Operations: filter

Eager filter

```
function filter(pred, xs)
    return is_null(xs) ? xs : pred(head(xs))
            ? pair(head(xs), filter(pred, tail(xs)))
            : filter(pred, tail(xs));
```

Lazy filter

```
function stream_filter(pred, xs)
    return is_null(xs) ? null : pred(head(xs))
        ? pair(head(xs), () =>
            stream_filter(pred, stream_tail(xs)))
        : filter(pred, stream_tail(xs));
```

An application

What does this do?

```
!is_null(head(stream_filter(x => x,
        stream_map(x => is_prime(x),
            enum_stream(A, B)))));
```

[1] They are all lazy, but some things are more lazy than others.

An application

What does this do?

```
!is_null(head(stream_filter(x => x,
        stream_map(x => is_prime(x),
            enum_stream(A, B)))));
```

It checks if there is a prime between A and B. Some questions:

- How lazy is it?
- Is `map` "equally lazy" as `filter`[1]?

[1] They are all lazy, but some things are more lazy than others.

An application

What does this do?

```
!is_null(head(stream_filter(x => x,
        stream_map(x => is_prime(x),
            enum_stream(A, B)))));
```

It checks if there is a prime between A and B. Some questions:

- How lazy is it?
- Is `map` "equally lazy" as `filter`[1]?
  - No.

[1] They are all lazy, but some things are more lazy than others.

An application

What does this do?

```
!is_null(head(stream_filter(x => x,
        stream_map(x => is_prime(x),
            enum_stream(A, B)))));
```

It checks if there is a prime between A and B. Some questions:

- How lazy is it?
- Is `map` "equally lazy" as `filter`[1]?
    - No.
    - `stream_filter` consumes *until it finds a passing candidate*. `stream_map` strictly consumes *only one element at a time*.

[1] They are all lazy, but some things are more lazy than others.

Recursive streams

Ones

```
const one = pair(1, () => one);
```

Recursive streams

Ones

```
const one = pair(1, () => one);
```

Constructing $\mathbb{N}$

```
N = pair(0, () => one + 1) // ???
const next =
    s => pair(head(s)+1, () => next(stream_tail(s)));
const N = pair(0, () => next(N));
```

Recursive streams

Ones

```
const one = pair(1, () => one);
```

Constructing ℕ

```
N = pair(0, () => one + 1) // ???
const next =
    s => pair(head(s)+1, () => next(stream_tail(s)));
const N = pair(0, () => next(N));
```

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

Repeated calls on tail:

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

Repeated calls on tail:

- `A = (1, () => scale_s(2, A))`

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

Repeated calls on tail:

- `A = (1, () => scale_s(2, A))`
- `t = (2, () => s_map((2*), t))`

S11 Q1

What is A?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

Repeated calls on tail:

- `A = (1, () => scale_s(2, A))`
- `t = (2, () => s_map((2*), t))`
- `tt = (4, () => s_map((2*), tt))`

S11 Q1

What is `A`?

```
const A = pair(1, () => scale_stream(2, A));
```

```
function scale_stream(c, stream) {
    return stream_map(x => c * x, stream);
}
function stream_map(f, xs) {
    return is_null(xs) ? null
        : pair(f(head(xs)),
            () => stream_map(f, stream_tail(xs)));
}
```

Repeated calls on tail:

- `A = (1, () => scale_s(2, A))`
- `t = (2, () => s_map((2*), t))`
- `tt = (4, () => s_map((2*), tt))`

Powers of 2.

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

- B = (1, () => mul_s(B, (1, ...)))

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

- `B = (1, () => mul_s(B, (1, ...)))`
- `t = (1, () => mul_s(t, (2, ...)))`

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

- `B = (1, () => mul_s(B, (1, ...)))`
- `t = (1, () => mul_s(t, (2, ...)))`
- `tt = (2, () => mul_s(tt, (3, ...)))`

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

- `B = (1, () => mul_s(B, (1, ...)))`
- `t = (1, () => mul_s(t, (2, ...)))`
- `tt = (2, () => mul_s(tt, (3, ...)))`
- `ttt = (6, () => mul_s(ttt, (4, ...)))`

S11 Q1

What is B?

```
const B = pair(1, () => mul_streams(B, integers));
```

```
function mul_streams(a,b) {
    return pair(head(a) * head(b),
        () => mul_streams(stream_tail(a), stream_tail(b)));
}
```

Repeated calls on tail:

- `B = (1, () => mul_s(B, (1, ...)))`
- `t = (1, () => mul_s(t, (2, ...)))`
- `tt = (2, () => mul_s(tt, (3, ...)))`
- `ttt = (6, () => mul_s(ttt, (4, ...)))`

Factorials.

S11 Q2

What does this do?

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

S11 Q2

What does this do?

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

On the finite stream 1,2,3,4,5:

```
pair(1, 2), pair(1, 3), pair(1, 4), pair(1, 5),
pair(2, 3), pair(2, 4), pair(2, 5),
pair(3, 4), pair(3, 5),
pair(4, 5)
```

S11 Q2

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

What does this do: `stream_pairs(integers)`?

```
function stream_append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs),
            () => stream_append(stream_tail(xs), ys));
}
```

S11 Q2

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

What does this do: `stream_pairs(integers)`?

```
function stream_append(xs, ys) {
    return is_null(xs)
        ? ys
        : pair(head(xs),
            () => stream_append(stream_tail(xs), ys));
}
```

It runs forever.

S11 Q2

```
function stream_append_pickle(xs, ys) {
    return is_null(xs)
        ? ys()
        : pair(head(xs),
          () => stream_append_pickle(stream_tail(xs), ys));
}
function stream_pairs2(s) {
    return is_null(s)
        ? null
        : stream_append_pickle(
          stream_map(
              sn => pair(head(s), sn),
              stream_tail(s)),
          () => stream_pairs2(stream_tail(s)));
}
```

Additions in red. How does this work?

S11 Q2

```
function stream_append_pickle(xs, ys) {
    return is_null(xs)
        ? ys()
        : pair(head(xs),
            () => stream_append_pickle(stream_tail(xs), ys));
}
function stream_pairs2(s) {
    return is_null(s)
        ? null
        : stream_append_pickle(
            stream_map(
                sn => pair(head(s), sn),
                stream_tail(s)),
            () => stream_pairs2(stream_tail(s)));
}
```

Additions in red. How does this work?

Laziness!

```
pair(1, 2), pair(1, 3), pair(1, 4), ...
```

S11 Q2

How to make our pickled version utilize `ys` as well when `xs` is infinite?

S11 Q2

How to make our pickled version utilize `ys` as well when `xs` is infinite?

S11 Q2

How to make our pickled version utilize `ys` as well when `xs` is infinite?

```
function interleave_stream_append(xs, ys) {
    return is_null(xs)
        ? ys()
        : pair(head(xs),
            () => interleave_stream_append(
                ys(), () => stream_tail(xs)));
}
```

S11 Q3

Create the streams `alt_ones`, `zeros`, `pos_integers`.

S11 Q3

Create the streams alt_ones, zeros, pos_integers.

```
const alt_ones = pair(1, () => pair(-1, () => alt_ones));
```

S11 Q3

Create the streams `alt_ones`, `zeros`, `pos_integers`.

```
const alt_ones = pair(1, () => pair(-1, () => alt_ones));

const zeros = add_streams(alt_ones, stream_tail(alt_ones));
```

S11 Q3

Create the streams alt_ones, zeros, pos_integers.

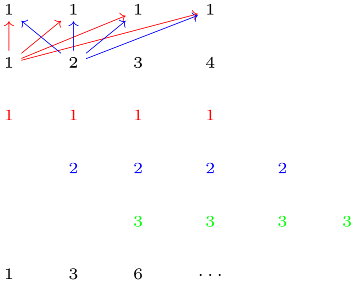
```
const alt_ones = pair(1, () => pair(-1, () => alt_ones));

const zeros = add_streams(alt_ones, stream_tail(alt_ones));

const ones = pair(1, () => ones);
const pos_integers =
    pair(1, () => add_streams(ones, pos_integers));
```

S11 Q4

Write a function to multiply two streams together like gradeschool multiplication.

S11 Q4

Write a function to multiply two streams together like gradeschool multiplication.



1 1 1 1

1 2 3 4

1 1 1 1

2 2 2 2

3 3 3 3

1 3 6 · · ·

S11 Q4

```
function mul_series(s1, s2) {
    return pair(head(s1) * head(s2),
        () => add_series(
            stream_tail(scale_series(head(s2), s1)),
            mul_series(stream_tail(s2), s1)));
}
```