# Streams

Jia Xiaodong

October 25, 2021

# 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!

### 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.



There are times to be eager, or over-eager. For example modern CPUs speculatively execute instructions that you haven't told it to, for example when waiting for memory to be addressed. It may have wasted a load of effort, but statistically speaking it is definitely faster.

However there is a benefit to laziness. Sometimes it is an optimization. Often the most elegant or creative solutions are the laziest. Yet machines don't really slack off. So we have to artificially introduce some laziness into the system.

# 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);
```



A contrived example.

These things don't work. This is because we aren't lazy! We are trying to evaluate these huge objects. However, do we really need all the way to infinity in one go? Can't we just hand out the next value when we need it?

# 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
```



We have talked about this before. There is one way for us to delay the evaluation of our lists, and that is with functions. With an anonymous function we can create a closure around the scope the object resides in and potentially create a recursive structure. This is what makes streams so powerful.

## 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 `()`.



Now we need new operations to go with our lazy lists. Looking at the definition it is easy to find a new definition. What about the other operations?

# 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));
```

## 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)))));
```

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.



Here is an example of something we can do. By mapping every number in a range to its `is_prime` result, we can check if there is a prime in this range by filtering for `true`. On an ordinary list, this will check all elements, but not so for a stream. It will stop once it finds the first one.

Naturally filter is less lazy than map since it has to actually find the next item that satisfies the predicate.

## 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));
```



There is a method presented in the lecture slides to generate the natural numbers. Here we present another way to showcase how you can build recursive structures.

Starting with 0, we want to constantly add 1 to it. Essentially, we can think of it being like `map`, but instead of applying the function (+1) just once, we apply it more and more each time we go down the list (compare the program here with the definition of map). The secret to stacking the applications is to make `N` have a tail that is `next(N)`, which will give us `(0, (1, () => next((1, ...))))` on the first evaluation. Another time, `(0, (1, (2, () => next((2, ...)))))`.

# 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.



We can evaluate this a few times to get a feel for what's going on. I have used some shorthand here but it will quickly become clear what they mean.

- The original list.
- Call `stream_tail`. This calls `stream_map` that returns a new list with the head multiplied by 2, and the tail is `stream_tail(A)` (since `xs <- A` here). The tail evaluates to `scale_stream(2, A)`, which is basically the pair we have here, so I mark it as `t` for tail.
- Call `stream_tail` again. Same thing happens. `tt` stands for tail tail.

Note: here we evaluate the bodies of the lambda functions. This is not correct behaviour, but only illustrative. We can get away with it here and in most places, and it saves a lot of space worrying about scopes and closures.

# 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.



We do the same thing.

- The original list.
- Calls `stream_tail`. This will create a new pair whose head is the multiplication of the heads of `B` and `ints`, which is `1 * 1`. The tail is then the lambda in the function body.
- The pattern repeats.

Note: again the order of evaluation here is wrong on purpose to save some space and make life easier.

## 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)
```



The function produces a stream containing all pairs $(p_i, p_j)$ where $p_i$ comes before $p_j$ in the input stream `s`.

We can see this from the `s_map` call. This converts every element in `s_tail(s)` into a pair `(head(s), sn)`. This is then appended with a call to `s_pairs(s_tail(s))`. Using some wishful thinking, we obtain our conclusion.

# 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.



We do not even need how `stream_append` is implemented. The problem is `stream_pairs` is not exactly lazy. Each time it calls `stream_append` (which is lazy), but by doing so the arguments have to be evaluated which causes another recursive call to `stream_pairs` to be made and the cycle will never stop since the input stream is infinite.

# 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), ...`



We have solved the problem by making the function lazy. We do this with the same way we make lists lazy, by delaying evaluation using an anonymous function. Now `append` does not demand two proper streams, but one stream `xs` and another delayed/pickled one `ys` that waits for `xs` to be exhausted first before being evaluated.

However this introduces one problem. If `xs` is infinite, then `ys` will never actually be activated. This is reflected in the output of `stream_pairs2(integers)`.

# 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)));
}
```



To give a chance to both streams, we will have to *interleave* them. This is quite a common thing to do in the electronics and computing world. For example, you have interlaced video and images.

In our case, we will have to change our append function to just swap `xs` and `ys` around after every call so they are utilized equally.

# S11 Q3

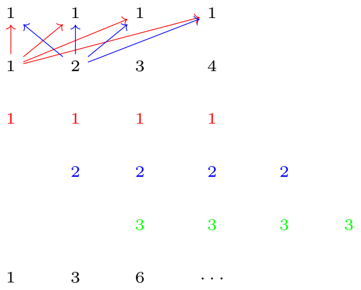
Create the streams alt_ones, zeros, pos_integers.

```
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));
```

These are fairly simple exercises. We have already seen `ones` before. `alt_ones` is similar in spirit, a simple circular list kind of thing. `zeros` can be achieved with adding two `alt_ones` together with one of them offset by 1 unit. `pos_integers` is created by adding `ones` to `pos_integers`, similar to how we created the stream `N` previously.

# S11 Q4

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






Given two streams, we want to pretend each element is a digit and multiply them in the gradeschool fashion.

In the figure, what we want to achieve is the black output at the bottom. We see that we will need two things: `add_streams` and `scale_streams`. It is quite simple once we get the recursive relationship down. In this example, we see the relation is

$$1111 \times 1234 = 1111 \times 234 + 1111 \times 1000.$$

Hence what we can do is add two series together: the wishful thinking series where we multiply `s1` with `s_tail(s2)`, and the other series comprising of actually multiplying `s1` with `head(s2)`

# 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)));
}
```