CS1101S Studio Session Week 11: *Stream*

Niu Yunpeng

niuyunpeng@u.nus.edu

October 30, 2018

Overview

1. Lazy evaluation
   - Computational model
   - Function application
2. Stream
   - Delayed evaluation
   - Stream programming

Lazy Evaluation

Computational model

- Computational model is a useful guideline for us to understand how the interpreter works.
- Computational model may vary depending on programming language and the runtime system used.
- In CS1101S, we introduce two computational models: ***substitution model*** and ***environment model***.

What to expect

- In the coming weeks, these two models will still be valid.

Lazy Evaluation

Substitution model

For *stateless* programming *only*:

- Evaluate all actual arguments;
- Replace all formal parameters with their actual arguments;
- Apply each statement in the function body (and get the return value);
- Repeat the first 3 steps until done.

Lazy Evaluation

Environment model

For *stateful* programming:

- Each frame contains a series of bindings of names and values.
- The value of a variable depends on its environment, a sequence of frames up to the global frame.
- Each function call will create a new frame and extend its enclosing environment.

Lazy Evaluation

Function in JavaScript

- Function in JavaScript is a first-class citizen (object).
- They have a `call` method.
- The `call` method is triggered when this function is applied.

Function application

- When a function is applied, "`this`" is prepanded to the list of parameters.

Lazy Evaluation

What does "evaluation" mean?

- JavaScript is a scripting language.
- The interpreter will only evaluate line-by-line sequentially.
- Thus, the value of a JavaScript program is always the value of the last statement (the last line).

Notice

- In other words, the other statements (except for the last one) do not affect the overall value of the program.
- However, they may have "side effects".

Lazy Evaluation

Function value & return value

- A function itself already represents a value, of "`function`" type.
- The return value of a function application is the value of the last statement, which is the `return` statement. It may be of "`number`", "`boolean`", "`string`", "`function`" type.

Thus...

- Function evaluation: evaluates one statement (the function object itself);
- Function application: evaluates all statements in the function body.

Lazy Evaluation

Exercise

- In the following slides, you are going to see a few short programs.
- Also, you will see a single line of comment.
- Identify the value of `x` at the point of that comment.
- Notice: *You may want to draw environment model diagram*.

Lazy Evaluation

Exercise 1

```
let x = 0;

function foo() {
    x = x + 1;
}

function bar(func) {
    // Here
    return func;
}

bar(foo());
```

Lazy Evaluation

Exercise 2

```
let x = 0;

function foo() {
    x = x + 1;
}

function bar(func) {
    // Here
    return func;
}

bar(foo);
```

Lazy Evaluation

Exercise 3

```
let x = 0;

function foo() {
    x = x + 1;
}

function bar(func) {
    return func;
}

bar(foo)();
// Here
```

Overview

1. Lazy evaluation
   - Computational model
   - Function application

2. Stream
   - Delayed evaluation
   - Stream programming

Stream

Inspiration

In order to generate an infinite list of $\{1, 1, 1, ...\}$, you are given the two approaches as follows:

```
// 1st approach
const ones = pair(1, ones);

// 2nd approach
const ones = pair(1, () => ones);
```

Think about it...

- Which one is correct?
- What is the output?

Stream

The 1st approach

- It will give rize to an error.
- The right side of an assignment statement will be evaluated before the actual assignment is done.
- The `pair` function will evaluate the values of its arguments before the pair is constructed.
- Thus, the tail of the pair has not been defined yet.

Notice

- The following also does not work

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

Stream

The 2nd approach

- There will not be any error.
- However, it is in fact not correct.
- It is not precise to say it is a list of $\{1, 1, 1, ...\}$.
- It is in fact a stream of $\{1, 1, 1, ...\}$.

Stream

JavaScript is not "*lazy*"

- For any assignment statement in JavaScript, the right side will always be evaluated before the actual assignment is done.
- Variable declaration and binding of function parameters to the values of actual arguments behave similar to assignment statements.
- Applicative order of reduction.

Delayed ~~lazy~~ evaluation

- Unlike some other languages like Haskell, JavaScript is not *lazy*.
- Thus, to **delay** the evaluation of some statements, we have to *wrap them into a function*.

Stream

Stream

- A stream is either an empty list, or a pair whose tail is a nullary function that returns a stream.
- A nullary function is a funcion with no parameters.

Revisit list library

Up to now, the list library supports different kinds of functions:

- List builder: `list`, `build_list`, `enum_list`;
- List getter: `head`, `tail`, `list_ref`, `member`, `is_member`;
- List information: `is_list`, `is_empty_list`, `length`, `equal`;
- List modifier: `append`, `reverse`, `remove`, `remove_all`, `filter`, `map`, `for_each`;
- List converter: `accumulate`, `list_to_string`.

Stream

Stream library

Up to now, the stream library supports different kinds of functions:

- Stream builder: `stream`, `build_stream`, `enum_stream`, `integers_from`;
- Stream getter: `stream_tail`, `stream_ref`, `stream_member`;
- List information: `is_stream`, `stream_length`;
- List modifier: `stream_append`, `stream_reverse`, `stream_map`, `stream_for_each`, `stream_remove`, `stream_remove_all`, `stream_filter`;
- List converter: `list_to_stream`, `stream_to_list`, `eval_stream`.

Stream

Apart from them - interleave

```
function interleave(s1, s2) {
    if (is_empty_list(s1)) {
        return s2;
    } else {
        return pair(head(s1),
                    () => interleave(s2, stream_tail(s1)));
    }
}
```

Use interleave - pairs

```
function pairs(s, t) {
    return pair(pair(head(s), head(t)), function () {
        const part1 = stream_map(x => pair(head(s), x),
                                 stream_tail(t));
        const part2 = pairs(stream_tail(s), stream_tail(t));

        return interleave(part1, part2);
    });
}
```

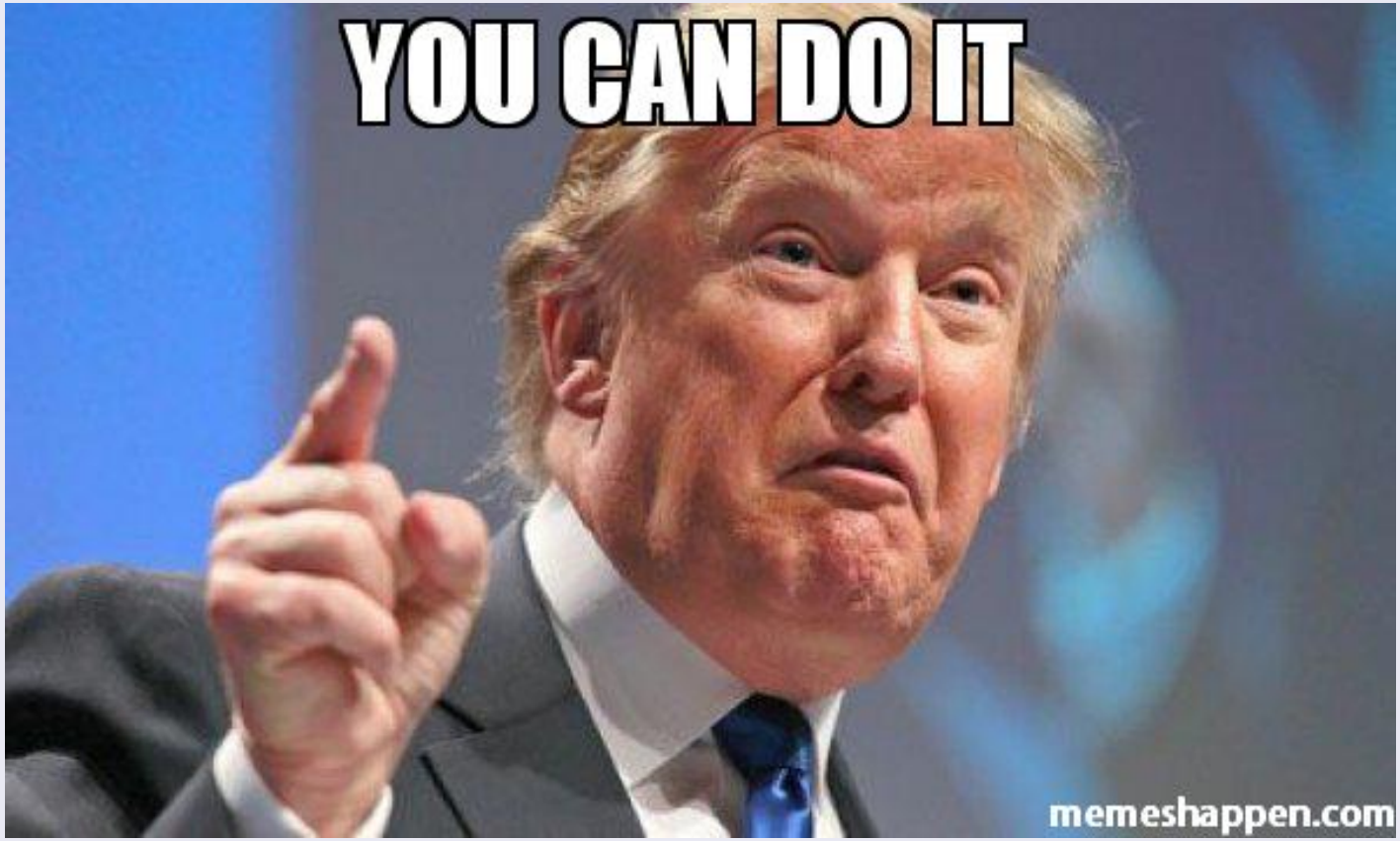
Stream

To become good at stream

- Do not forget the pair/list;
- Understand higher-order programming well;
- Always do wishful thinking!

Stream

Try your best...



Let's discuss them now.

End

The End

Copyright

This work is mostly an original by *Niu Yunpeng*. It may either directly or indirectly benefit from the previous work of Martin Henz, Cai Deshun. For illustration purposes, some pictures in the public domain are used. Upon request, detailed acknowledgments will be provided.