Lazy Evaluation in Source Academy

CS4215 T1 Lazy in SA 2

A0161364J - Nguyen Tien Trung Kien
A0212539H - Xiao Tian Yi

Project Objective

In this project, we implemented lazy evaluation and lazy list in Source Academy. To be precise, we added the features to the interpreter of Source §1 and Source §2.

Background

Lazy evaluation is an evaluation strategy where an expression is only evaluated when it is needed.

Formally, lazy evaluation has two main properties:

- Delayed evaluation: Evaluation is delayed until necessary.
- Memoization: Any expression is only evaluated at most once.

Here are some benefits of lazy evaluation.

- By avoiding unnecessary computations, the performance would be increased for some programs.
- The control flow could be defined as a function (see Appendix A, Example 1: Define control flow with function)
- Infinite data structure could be defined (see Appendix A, Example 2: Infinite list)

Originally in js-slang, the evaluation is in applicative-order, which means all arguments of a function would be evaluated before applied to the function. And now we would like to implement another evaluation strategy, which is lazy evaluation.

Specification

Here is the list of specifications that we set in our project:

- Basic requirements of lazy evaluation
    - Delayed evaluation: In contrast to applicative-order, an expression will only be evaluated when its value is needed. If not needed, then the expression will not be evaluated.
    - Memoization: For every expression will only be evaluated at most once.
- Basic requirements of lazy list
    - Length of lazy list could be infinite.
    - Elements in lazy list would be calculated only when it is needed.
    - Each element in lazy list should only be evaluated at most once.
- Compatibility

    - Backward compatibility: There is an option to switch between the lazy evaluation mode and the original mode.
    - Compatibility with cadet-frontend: The API of all the functions in `index.ts` (e.g. `runInContext`) are compatible with the newest cadet-frontend.

Scope

- Language implementation: In our project, we only support lazy evaluation in interpreter, not in transpiler or stepper.
- Source chapter: We only support Source chapter 1 & 2 only. Due to imperative features, Source chapter 3 & 4 does not fully support lazy evaluation.

Deliverables

In this project, we have completed the following items:

- Source code
    - Completely supports lazy evaluation and lazy list.
    - Satisfies specification list and the scope above.
    - Well-documented with TypeDoc
    - Matches the project's coding style
- Tests
    - Tests for lazy evaluation
    - Tests for lazy list
- Documentation
    - API documentation
    - Report (this file)
- Pull request (#535)
    - Passed CI tests
    - Coverage increased by 0.2%

User guide

To run js-slang in lazy evaluation mode from console:

```
$ node dist/repl/repl.js [chapter] auto lazy
```

or

```
$ node dist/repl/repl.js [chapter] lazy
```

To run js-slang without lazy evaluation mode from console:

```
$ node dist/repl/repl.js [chapter] [executionMethod]
```

In the command,

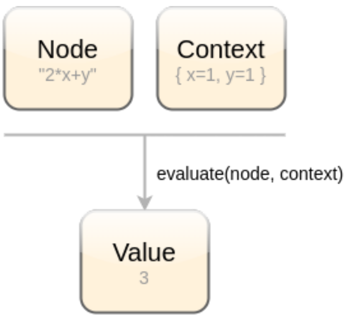
- The argument `[chapter]` determines the source chapter from 1 to 4, could not be blank if the user wants to select execution method or switch on lazy evaluation mode. Default to be 1.
- The argument `[executionMethod]` determines the execution method of programs, which are:
  - `auto`: interpreter in lazy evaluation mode, otherwise transpiler
  - `subst`: stepper
  - `interpreter`: interpreter
- The argument `[variant]` determines if we switch on the lazy evaluation mode. To switch it on, let it be `lazy`, otherwise just leave it blank.

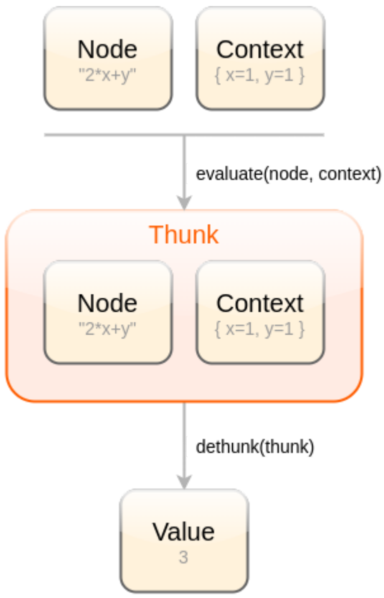
Implementation

Main idea

When the `evaluate` function is called (with parameters node and `context`), instead of evaluating immediately, the `evaluate` function simply creates a Thunk containing node and `context`. Only when the `dethunk` function is called, the actual evaluation is done.

The `dethunk` function is called when `forceEvaluate` is called, when `deepDethunk` is called, or when it is called directly. For more details, see the section about `dethunk` function below.

<table>
<tr><th>Applicative-order evaluation</th><th>Lazy evaluation</th></tr>
<tr><td>


Applicative-order strategy evaluates a node given a context, and returns a value
</td><td>


Lazy strategy creates a thunk which is a tuple of a node and a context. The thunk is only dethunked when necessary.
</td></tr>
</table>

Thunk

We defined the Thunk class as follows:

```
export default class Thunk {
    public node: es.Node

    public context: Context
    public isEvaluated: boolean
    public result: Value
    ...
}
```

The meaning of the fields above are straightforward. `isEvaluated` and `result` enable memoization.

dethunk() and memoization

We defined `Thunk.dethunk` function as follows:

```
export default class Thunk {
  ...
  public *dethunk(): Value {
    if (!this.isEvaluated) {
      this.result = yield* forceEvaluate(this.node, this.context)
      this.isEvaluated = true
    }
    return this.result instanceof Thunk ?
      this.result.dethunk() :
      this.result
  }
```

This function satisfies the following requirements:

- Each expression is only evaluated at most once. Once an expression is evaluated, its result is memoized and stored in `Thunk.result`.
- `dethunk` never returns a Thunk. If an intermediate result is a Thunk, it will be dethunked until it is no longer a Thunk.

There are three cases that dethunk is called:

- When `forceEvaluate` is called: because this function has to return a non-thunk `Value`, its return value is dethunked.
- When `deepDethunk` is called, i.e. before printing the final result to users, or before an argument (possibly a Thunk) is passed to a built-in non-thunk-aware function (e.g. `display` function).
- When it is called directly by thunk-aware functions (e.g. pair, head, tail in thunk-list.ts)

deepDethunk() and non-thunk-aware built-in functions

Issue of non-thunk-aware built-in functions

Most of the built-in functions are not thunk-aware, i.e. they cannot handle properly if at least one argument is a Thunk. Non-thunk-aware built-in functions include: `display`, `stringify`, `prompt`, `is_number`, etc.

To maintain the compatibility, when a non-thunk-aware built-in function is called, its arguments are deep-dethunked.

In our implementation, when a built-in function is being called, the following steps will be done:

1. If the function is a thunk-aware function, simply make the function call
2. Otherwise, convert it to a thunk-aware function, then make the function call

To perform the steps above, we implemented two functions, namely `isThunkAware` and `makeThunkAware`.

```
export function isThunkAware(fun: Value): boolean {
  if (fun.hasOwnProperty('isThunkAware')) {
    return fun.isThunkAware
  }
  return false
}
```

```
export function makeThunkAware(fun: Value, thisContext?: Value):
ThunkAwareFuntion {
  function* wrapper(...args: Value[]): IterableIterator<Value> {
    if (isThunkAware(fun)) {
      return yield* fun.apply(thisContext, args)
    }
    const dethunkedArgs = [...args]
    for (let i = 0; i < dethunkedArgs.length; i++) {
      dethunkedArgs[i] = yield* deepDethunk(dethunkedArgs[i])
    }
    return fun.apply(thisContext, dethunkedArgs)
  }
  wrapper.isThunkAware = true
  return wrapper
}
```

Thunk-aware functions

In `thunk-list.ts`: `pair`, `is_pair`, `head`, `tail`, `is_null`, `list`, `set_head`, `set_tail`

In `thunk-stream.ts`: `stream_tail`, `stream`, `list_to_stream`

deepDethunk function

A value is considered to be deep-dethunked iff:

- The value is not a `Thunk`, and
- If the value is an array, its elements have to be deep-dethunked.

`deepDethunk(value)` returns the deep-dethunked version of `value`.

e.g. `deepDethunk([1, Thunk(2, ...), [Thunk(3, ...)]]) = [1, 2, [3]]`

Here is the implementation of `deepDethunk`:

```
export function* deepDethunk(value: Value): Value {
  const result = value instanceof Thunk ? yield* value.dethunk() :
    value
  if (Array.isArray(result)) {
    for (let i = 0; i < result.length; i++) {
      result[i] = yield* deepDethunk(result[i])
    }
  }
  return result
}
```

`deepDethunk` are called in the two following cases:

- Calling a non-thunk-aware built-in function
- Printing the final result to users

See also

- Api documentation for interpreter: `src/interpreter/docs` (link)
- Api documentation for stdlib: `src/stdlib/docs` (link)

Appendix A: Example Programs

Example 1: Define control flow with function

Code

```
function unless(condition, if_no,   if_yes)
{
  return condition ? if_yes : if_no;
}

const xs = null;
unless(xs === null,
  head(xs),
  display("error: xs should not be null"));
```

Result

```
"error:xs should not be null"
```

Example 2: Infinite list

Code

```
const ones = pair(1, ones);
display(tail(head(ones)));
display(tail(tail(head(ones))));
```

Result

```
1
1
```

Code

```
function increment(x) {
  return pair(x,
    increment(x + 1));
}

function square(x) {
  return x * x;
}

const square_number = map(
    square, increment(0));

display(list_ref(square_number, 1));
display(list_ref(square_number, 2));
display(list_ref(square_number, 3));
display(list_ref(square_number, 4));
```

Result

```
1
4
9
16
```

Example 3: Two-dimension lazy list

Code

```
function binomial(n, k){
  function cells(t, i){
    return pair(find(t-1, i-1) +
      find(t-1, i),
      cells(t, i+1));
  }
  function rows(i){
```

Result

```
21
56
```

<pre>   return pair(cells(i, 1),
      rows(i+1));
  }
  function find(i,j){
    if(i <= 0 || j <= 0 ||
      i < j){
      display("error: invalid
        inputs");
    } else if(j === 1){
      return i;
    } else if(i === j){
      return 1;
    } else{
      return list_ref(
        head(rows(i)),
        j-1);
    }
  }
  return find(n, k);
}
 	
binomial_coefficient(7, 2);
binomial_coefficient(6, 3);</pre>	

Example 4: Expression is evaluated at most once

Code	Result

<pre>function f() {
    display("info");
    return 1;
}
const a = f();
display(a + a + a);</pre>
	“info”
	3

Code	Result

<pre>const b = pair(1, error());
display(head(b));</pre>
	1