B11: Meta-Circular Evaluator II; Language Processing

CS1101S: Programming Methodology

Martin Henz

October 29, 2021

Review

Most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

Review

Most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- A calculator language: primitive operators, literals, `evaluate`

Review

Most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- A calculator language: primitive operators, literals, `evaluate`
- Adding booleans, conditionals, sequences

Review

Most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- A calculator language: primitive operators, literals, `evaluate`
- Adding booleans, conditionals, sequences
- Adding blocks and declarations

Review

Most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- A calculator language: primitive operators, literals, `evaluate`
- Adding booleans, conditionals, sequences
- Adding blocks and declarations
- Adding compound functions (but no `return`)

Return statements in JavaScript

The Problems

- Functions that do not evaluate a return statement must return `undefined`

Return statements in JavaScript

The Problems

- Functions that do not evaluate a return statement must return `undefined`
- Evaluation of a return statement *anywhere* in the function body will return from the function with the result of evaluating the return expression

Adding return statements

$$\begin{array}{lcll} \textit{stmt} & ::= & \cdots & \\ & | & \texttt{return}\ \textit{expr}\ \texttt{;} & \text{return statement} \end{array}$$

Example:

```
function fact(n) {
    return n === 1 ? 1 : n * fact(n - 1);
}
fact(5);
```

Evaluating compound functions

```
function evaluate(comp, env) {
 return ...
   : is_return_statement(comp)
   ? eval_return_statement(component, env)
   : error(comp, "Unknown component:");
}
```

Handling return statements

Example: `return n * fact(n - 1);`

```
function eval_return_statement(stmt, env) {
    return make_return_value(
        evaluate(return_statement_expression(stmt),
                 env));
}
function make_return_value(content) {
    return list("return_value", content);
}
function is_return_value(value) {
    return is_tagged_list(value, "return_value");
}
```

Handling return values in sequences

```
function eval_sequence(stmts, env)          {
  if (is_empty_sequence(stmts)) {
    return undefined;
  } else if (is_last_statement(stmts)) {
    return evaluate(first_statement(stmts), env);
  } else {
    const first_stmt_value =
          evaluate(first_statement(stmts),env);
    if (is_return_value(first_stmt_value)) {
      return first_stmt_value;
    } else {
      return eval_sequence(
                 rest_statements(stmts),env);
  } }                                        }
```

Handling return values in apply

```
function apply(fun, args) {
  ...
  const result = evaluate(...body...);
  if (is_return_value(result)) {
    return return_value_content(result);
  } else {
    return undefined;
  } ...
}
```

Evaluator with loops

```
function evaluate(stmt, env) {
  return ...
         : is_while_loop(stmt)
         ? eval_while_loop(stmt, env)
         : is_for_loop(stmt)
         ? eval_for_loop(stmt, env)
         : ...;
}
```

Evaluation of while loops

```
function eval_while_loop(stmt, env) {
  if (evaluate(while_loop_predicate(stmt),
                    env) {
    evaluate(while_loop_statements(stmt), env);
    return eval_while_loop(stmt, env);
  } else {
    return true;
  }
}
```

Evaluation of while loops

```
function eval_while_loop(stmt, env) {
  if (evaluate(while_loop_predicate(stmt),
               env) {
    evaluate(while_loop_statements(stmt), env);
    return eval_while_loop(stmt, env);
  } else {
    return true;
  }
}
```

Challenges

While loops in JavaScript return the result of the last evaluation of the body!

Evaluation of while loops

```
function eval_while_loop(stmt, env) {
  if (evaluate(while_loop_predicate(stmt),
               env) {
    evaluate(while_loop_statements(stmt), env);
    return eval_while_loop(stmt, env);
  } else {
    return true;
  }
}
```

Challenges

While loops in JavaScript return the result of the last evaluation of the body! How about `break;` and `continue;`?

Evaluation of For-loops: A first approximation

```
for (E1; E2; E3) { statements }
```

is an abbreviation for

```
{ E1;
  while (E2) {
      statements
      E3;
  }
}
```

In our evaluator: A first approximation

```
function eval_for_loop(stmt, env) {
 const whole_block =
  make_block(
   make_sequence(
    list(
     for_loop_initialiser(stmt),
     make_while_loop(for_loop_predicate(stmt),
      make_block(
       make_sequence(
        list(
         block_body(for_loop_statements(stmt)),
         for_loop_finaliser(stmt))))))));
 return evaluate(whole_block, env);
}
```

A strange example

```
const a = [];
for (let i = 0; i < 10; i = i + 1) {
    a[i] = x => x + i;
}
a[5](100);
```

JavaScript design decision

```
const a = [];
for (let i = 0; i < 10; i = i + 1) {
    a[i] = x => x + i;
}
a[5](100);
```

JavaScript design decision

```
const a = [];
for (let i = 0; i < 10; i = i + 1) {
    a[i] = x => x + i;
}
a[5](100);
```

Special treatment of loop control variable

Each iteration of the body has its own “copy” of the loop control variable.

JavaScript design decision

```
const a = [];
for (let i = 0; i < 10; i = i + 1) {
    a[i] = x => x + i;
}
a[5](100);
```

Special treatment of loop control variable

Each iteration of the body has its own "copy" of the loop control variable.

Therefore, the result of the program is 105.

An example

```
unless(is_null(xs), head(xs), display("xs null"))
```

An example

```
unless(is_null(xs), head(xs), display("xs null"))
```

Can we implement `unless` like as follows?

```
function unless(condition, usual, exceptional) {
    return condition ? exceptional : usual;
}
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)
```

```
->  sum_of_sqs(5 + 1, 5 * 2)
```

```
->  sum_of_sqs(6, 10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sum_of_sqs(6, 10)

->  sq(6) + sq(10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sum_of_sqs(6, 10)

->  sq(6) + sq(10)

->  (6 * 6) + (10 * 10)
```

Applicative order reduction, 1.1.5

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->   sum_of_sqs(5 + 1, 5 * 2)

->   sum_of_sqs(6, 10)

->   sq(6) + sq(10)

->   (6 * 6) + (10 * 10)

⋮

->   136
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}

f(5)

->  sum_of_sqs(5 + 1, 5 * 2)
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)
```

```
->  sum_of_sqs(5 + 1, 5 * 2)
```

```
->  sq(5 + 1) + sq(5 * 2)
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sq(5 + 1) + sq(5 * 2)

->  ((5 + 1) * (5 + 1)) +
    ((5 * 2) * (5 * 2))
```

Normal order reduction (L2)

```
function sq(x) { return x * x; }
function sum_of_sqs(x,y) { return sq(x)+sq(y);}
function f(a) { return sum_of_sqs(a+1, a*2);}
```

```
f(5)

->  sum_of_sqs(5 + 1, 5 * 2)

->  sq(5 + 1) + sq(5 * 2)

->  ((5 + 1) * (5 + 1)) +
    ((5 * 2) * (5 * 2))

⋮

->  136
```

Lazy evaluation

Laziness

In applications, arguments are not evaluated. They are passed to the function as expressions.

Lazy evaluation

Laziness

In applications, arguments are not evaluated. They are passed to the function as expressions.

Force evaluation when needed

The argument expressions are evaluated when needed, during evaluation of body

Implementation

"Eager" evaluation (applicative order)

```
: is_application(comp)
? apply(evaluate(function_expression(comp),
                 env),
        list_of_values(arg_expressions(comp),
                       env))
```

Implementation

"Eager" evaluation (applicative order)

```
: is_application(comp)
? apply(evaluate(function_expression(comp),
                 env),
        list_of_values(arg_expressions(comp),
                       env))
```

"lazy" evaluation (normal order)

```
: is_application(comp)
? apply(actual_value(function_expression(comp),
                     env),
        arg_expressions(comp),
        env)
```

Implementation of thunks

```
function delay_it(exp, env) {
    return list("thunk", exp, env);
}
function is_thunk(obj) {
    return is_tagged_list(obj, "thunk");
}
function thunk_exp(thunk) {
    return head(tail(thunk));
}
function thunk_env(thunk) {
    return head(tail(tail(thunk)));
}
```

Implementation of forcing (simple version)

```
function force_it(obj) {
    if (is_thunk(obj)) {
        return actual_value(thunk_exp(obj),
                            thunk_env(obj));
    } else {
        return obj;
    }
}
```

Implementation of forcing (with memoization)

```
function force_it(obj)                                   {
  if (is_thunk(obj)) {
    const res = actual_value(thunk_exp(obj),
                             thunk_env(obj));
    set_head(obj, "evaluated_thunk");
    set_head(tail(obj), res);  // replace value
    set_tail(tail(obj), null); // forget env
    return res;
  } else if (is_evaluated_thunk(obj)) {
    return thunk_value(obj);
  } else {
    return obj;
  }                                                      }
```

Summary

Summary

- Return statements

Summary

- Return statements
- While and for loops

Summary

- Return statements
- While and for loops
- Lazy evaluation

Summary

- Return statements
- While and for loops
- Lazy evaluation
- Challenge: Can you make our evaluator truly *meta-circular*? Can you get it to evaluate itself?