YOCTO-CFA:
A PROGRAM ANALYZER THAT YOU CAN UNDERSTAND

by
Leandro Facchinetti

A dissertation submitted to Johns Hopkins University
in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
August 2020

2020-07-07T15:10:07.564Z

Table of Contents

1 Developing an Analyzer

[](#developing-an-analyzer)

1.1 The Analyzed Language: Yocto-JavaScript

[](#the-analyzed-language-yocto-javascript)

Our first decision when developing a program analyzer is which language it should analyze. This decision is important, among other reasons, because it influences how difficult it is to develop the analyzer. In this dissertation we are interested in analysis techniques for higher-order functions, a feature that is present in most languages, so we have plenty of options from which to choose, including JavaScript, Java, Python, Ruby, and so forth.

From all these options, we would like to choose JavaScript because it is the most popular programming language [30, 11], but JavaScript has many features besides higher-order functions that would complicate our study. As a compromise, we analyze some parts of JavaScript, not the entire language. We select the JavaScript features that are related to higher-order functions to design our own artificial little language called *Yocto-JavaScript* ($\text{JavaScript} \times 10^{-24}$), which becomes the language over which our analyzer works. We design Yocto-JavaScript such that every Yocto-JavaScript program is also a JavaScript program, but the converse does not hold.

Advanced

On the surface the choice of analyzed language is important because it influences how difficult it is to develop the analyzer, but that choice has deeper consequences as well: the analyzed language may also influence the analyzer's precision and running time. For example, there is an analysis technique called k-CFA [29] that may be slower when applied to a language with higher-order functions than when applied to a language with objects, because the algorithmic complexity of the former is exponential and of the latter is polynomial [18].

Technical Terms

- **λ-Calculus [31 (§ 6)]:** A mathematical theory to study higher-order functions and their applications. Yocto-JavaScript is a representation of the λ -calculus.

1.1.1 Values in Yocto-JavaScript [](#values-in-yocto-javascript)

JavaScript has many kinds of values:

Kind of JavaScript Value	Example
`String`	`"Leandro"`
`Number`	`29`
`Array`	`["Leandro", 29]`
`Object`	`{ name: "Leandro", age: 29 }`
`Function`	`x => x`
⋮	⋮

From all these kinds of values, Yocto-JavaScript supports only one: `Function`. An Yocto-JavaScript function is written as `<parameter> => <body>`, for example, `x => x`, in which the `<parameter>` is called `x` and the `<body>` is a reference to the variable `x` (see § 1.1.2 for more on variable references). An Yocto-JavaScript function must have exactly one parameter. Because an Yocto-JavaScript function is a value, it may be passed as argument in a function call or returned as the result of a function call (see § 1.1.2 for more on function calls).

Technical Terms

- **Arrow Function Expressions [19]:** The notation we use for writing functions.
- **Identity Function:** The function given as example, `x => x`.
- **High-Order Functions:** Functions that may act as values.

1.1.2 Operations in Yocto-JavaScript

[](#operations-in-yocto-javascript)

JavaScript has many kinds of operations on the values introduced in § 1.1.1:

Kind of JavaScript Operation	Example	Result
Access a character in a `String`	`"Leandro"[2]`	`"a"`
Add `Numbers`	`29 + 1`	`30`
Call a `Function`	`parseInt("29")`	`29`
⋮	⋮	⋮

From all these operations, Yocto-JavaScript supports only two: function calls and variable references. A function call is written as <function>(<argument>), for example, `f(a)`, in which the <function> is a hypothetical function `f` and the <argument> is a hypothetical argument `a`. An Yocto-JavaScript function call must have exactly one argument (because an Yocto-JavaScript function must have exactly one parameter; see § 1.1.1). A variable reference is written as a bare identifier, for example, `x`.

The following is a complete Yocto-JavaScript program that exemplifies all the supported operations:

Example Yocto-JavaScript Program	Result
`(y => y)(x => x)`	`x => x`

This program is a function call in which the <function> is `y => y` and the <argument> is `x => x`. When an Yocto-JavaScript function is called, it returns the result of computing its <body>, and the <body> of `y => y` is a reference to the variable `y`, so `y => y` is a function that returns its argument unchanged and the result of the program above is `x => x`.

In general, all kinds of Yocto-JavaScript expressions (function definitions, function calls, and variable references) may appear in the <body> of a function definition, or as the <function> or <argument> of a call. For example, in the program `(f(a))(b)` the function call `f(a)` appears as the <function> of a call.

We use parentheses to resolve ambiguities on where function definitions start and end, and in which order operations are computed. For example, given hypothetical functions `f`, `g`, and `h`, in `(f(g))(h)` the call `f(g)` happens first and the result is a function that is called with `h` as argument, and in `f(g(h))` the call `g(h)` happens first and the result is passed as argument in a call to `f`. If there are no parentheses in a sequence of expressions, then the following conventions apply:

Kind of Sequence	Reading Direction	Example	Equivalent to
Function Definitions	Right-to-Left	`x => y => x`	`x => (y => x)`
Function Calls	Left-to-Right	`f(a)(b)`	`(f(a))(b)`

Technical Terms

- **Precedence:** The order in which operations of different kinds are computed. Operations that are computed first have *higher precedence* and operations that are computed later have *lower precedence*.

- **Associativity:** The order in which a sequence of operations of the same kind is computed. Function definitions are *right-associative* and function calls are *left-associative*.

Advanced

1.1.3 The Computational Power of Yocto-JavaScript

[](#the-computational-power-of-yocto-javascript)

Yocto-JavaScript has only a few features, which makes it the ideal language for discussing the analysis of higher-order functions, but does it have enough features to support all kinds of computation? Perhaps surprisingly, the answer is positive: Yocto-JavaScript is equivalent to JavaScript (and Java, Python, Ruby, and so forth) in the sense that any program in any one of these languages may be translated into an equivalent program in any other language [31 (§ 6)].

As an example of how this translation could be done, consider a JavaScript function of two parameters: `(x, y) => x`. This function is not supported by Yocto-JavaScript because it does not have exactly one parameter (see § 1.1.1), but we may encode it as `x => y => x`, which is a function that receives the first parameter and returns another function that receives the second parameter. Similarly, we may encode a call with multiple arguments as a sequence of calls that passes one argument at a time; for example, `f(a, b)` may be encoded as `f(a)(b)`.

Technical Terms

- **Computational Power:** The ability of expressing computations, for example, adding two numbers together, selecting a character from a string, and so forth.

- **Turing Complete [31 (§ 7)]:** The property of a language that may express any computation of which a computer is capable. Yocto-JavaScript, JavaScript, Java, Python, Ruby, and so forth are all Turing Complete.
- **Currying [31 (page 163)]:** The translation technique we used to encode functions with multiple parameters and calls with multiple arguments.

For our goal of exploring analysis techniques, we are concerned only with computational power, but it is worth noting that programmers would be more interested in other language properties: Does the language promote writing programs of higher quality? (It most probably does not [4].) Does the language improve productivity? Is the language appropriate for the domain of the problem? (For example, we would probably write an operating system in C and a web application in JavaScript, not the other way around.) Is the language more expressive than others? (Perhaps surprisingly, it is possible to make formal arguments about expressiveness instead of resorting to personal preference and anecdotal evidence [7].) Despite having the same computational power as other languages, Yocto-JavaScript fares badly in these other aspects: it is remarkably unproductive and inexpressive.

1.1.4 A Formal Grammar for Yocto-JavaScript

[](#a-formal-grammar-for-yocto-javascript)

The description of Yocto-JavaScript given in § 1.1.1–§ 1.1.2 is informal; the following is a grammar in *Backus–Naur Form* (BNF) [17, 1 (§ 4.2)] that formalizes it:

$$\begin{array}{rcll} e & ::= & x \texttt{ => } e \mid e\texttt{(}e\texttt{)} \mid x & \text{Expressions} \\ x & ::= & \text{«A JavaScript Identifier»} & \text{Variables} \end{array}$$

1.2 The Analyzer Language: TypeScript

[](#the-analyzer-language-typescript)

After choosing our analyzed language (Yocto-JavaScript; see § 1.1), we must decide in which language to develop the analyzer itself. Our analyzed language is based on JavaScript, so JavaScript is a natural first candidate for analyzer language as well. But JavaScript lacks a feature which we will need: the ability to express the *types* of data structures, functions, and so forth (see, for example, § 1.3.2), so we choose to implement our analyzer in a JavaScript extension with support for types called *TypeScript* [33, 32, 5].

1.3 Step 0: Substitution-Based Interpreter

[](#step-0-substitution-based-interpreter)

Having chosen the analyzed language (Yocto-JavaScript; see § 1.1) and the language in which to develop the analyzer itself (TypeScript; see § 1.2), we are ready to start developing the analyzer. This development happens in Steps: In Step 0 we develop an interpreter and in each subsequent Step we modify the program from the previous Step in some way until it becomes an analyzer.

The interpreter in Step 0 executes Yocto-JavaScript programs and produces the same outputs that would be produced by a regular JavaScript interpreter. This is a good starting point for two reasons: first, this interpreter is the basis upon which we will build the analyzer; and second, the outputs of this interpreter are the ground truth against which we will validate the outputs of the analyzer.

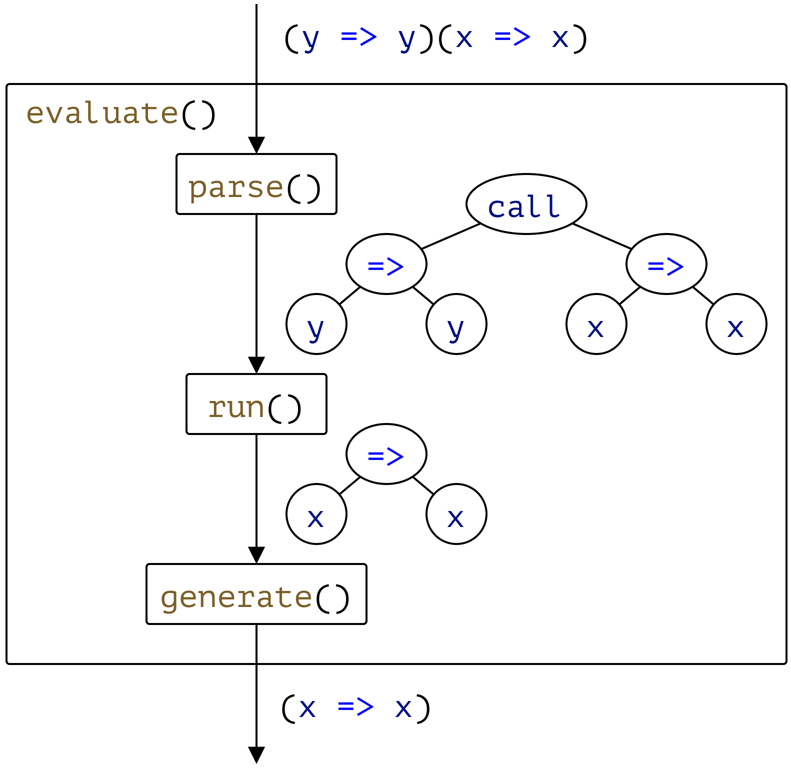
1.3.1 Architecture [](#architecture)

Our interpreter is defined as a function called `evaluate()`, which receives an Yocto-JavaScript program represented as a string and returns the result of running it.

The following are two examples of how we will be able to use `evaluate()` by the end of Step 0 (the > represents the console, and by convention strings that represent Yocto-JavaScript programs are delimited by backticks (`) [25]):

```
> evaluate(`x => x`)
`x => x`
> evaluate(`(y => y)(x => x)`)
`x => x`
```

The implementation of `evaluate()` is separated into three parts called `parse()`, `run()`, and `generate()`:




```
export function evaluate(input: string): string {
  return generate(run(parse(input)));
}
```

The `parse()` function prepares the `input` for interpretation, converting it from a string into more convenient data structures (see § 1.3.2 for more on these data structures). The `run()` function is responsible for the interpretation itself. The `generate()` function converts the outputs of `run()` into a human-readable format. In the following sections (§ 1.3.2–§ 1.3.14) we address the implementation of `run()`, deferring `parse()` to § 1.3.15 and `generate()` to § 1.3.16.

In later Steps the implementations of `run()` and `generate()` will change, but the implementations of `evaluate()` and `parse()` will remain the same, because the architecture and the data structures used to represent Yocto-JavaScript programs will remain the same.

Advanced

The `evaluate()` function is named after a native JavaScript function called `eval()` [21], which is similar to `evaluate()` but for JavaScript programs instead of Yocto-JavaScript. The `parse()` and `generate()` functions are named after the library functions used to implement them (see § 1.3.15 and § 1.3.16).

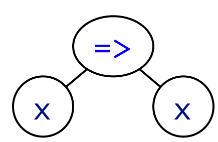
1.3.2 Data Structures to Represent Yocto-JavaScript Programs

[](#data-structures-to-represent-yocto-javascript-programs)

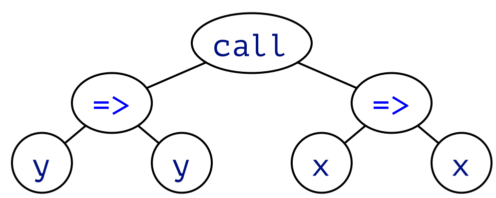
The `evaluate()` function receives an Yocto-JavaScript program represented as a string (see § 1.3.1), which is convenient for humans to write and read, but inconvenient for `run()` to manipulate directly, because `run()` is concerned with the *structure* rather than the *text* of the program: for `run()` it does not matter, for example, whether a function is written as `x => x` or `x=>x`. So before `run()` starts interpreting the program, `parse()` transforms it from a string into more convenient data structures (see § 1.3.15 for `parse()`'s implementation).

The following are two examples of Yocto-JavaScript programs and the data structures used to represent them:

```
> parse(`x => x`)

{
  "type": "ArrowFunctionExpression",
  "params": [
    {
      "type": "Identifier",
      "name": "x"
    }
```



```
  ],
  "body": {
    "type": "Identifier",
    "name": "x"
  }
}


> parse(`(y => y)(x => x)`)

{
  "type": "CallExpression",
  "callee": {
    "type": "ArrowFunctionExpression",
    "params": [
      {
        "type": "Identifier",
        "name": "y"
      }
    ],
    "body": {
      "type": "Identifier",
      "name": "y"
    }
  },
  "arguments": [
    {
      "type":
"ArrowFunctionExpression",
      "params": [
        {
          "type": "Identifier",
          "name": "x"
        }
      ],
      "body": {
        "type": "Identifier",
        "name": "x"
      }
    }
```



```
  ]
}
```

We choose to represent Yocto-JavaScript programs with the data structures above because they match the data structures used by Babel [2], which is a library to manipulate JavaScript programs that we use to implement the `parse()` and `generate()` functions (see § 1.3.15 and § 1.3.16).

In general, the data structures used to represent Yocto-JavaScript programs have the following types (written as TypeScript types adapted from the Babel types [3] to include only the features supported by Yocto-JavaScript):

```
type Expression = ArrowFunctionExpression | CallExpression | Identifier;
type ArrowFunctionExpression = {
  type: "ArrowFunctionExpression";
  params: [Identifier];
  body: Expression;
};
type CallExpression = {
  type: "CallExpression";
  callee: Expression;
  arguments: [Expression];
};
type Identifier = {
  type: "Identifier";
  name: string;
};
```

In later Steps almost everything about the interpreter will change, but the data structures used to represent Yocto-JavaScript programs will remain the same.

Technical Terms

- **Parsing [1 (§ 4)]:** The process of converting a program represented as a string into more convenient data structures.

- **Abstract Syntax Tree (AST) [1 (§ 4)]:** The data structures that represent a program.

Advanced

The data structures used to represent programs correspond to the Yocto-JavaScript grammar (see § 1.1.4); for example, `Expression` corresponds to *e*.

1.3.3 An Expression That Already Is a Value

[](#an-expression-that-already-is-a-value)

Example Program	Current Output	Expected Output
`x => x`	—	`x => x`

Having defined the architecture (§ 1.3.1) and the data structures to represent Yocto-JavaScript programs (§ 1.3.2), we start developing the `run()` function. The development is driven by a series of example programs that highlight different aspects of the interpreter. In § 1.3.3–§ 1.3.12 we begin with these example programs and modify the implementation to achieve the expected output.

Consider the example program above. As mentioned in § 1.3.2, the `run()` function receives as parameter an Yocto-JavaScript program represented as an `Expression`. The `run()` function is then responsible for interpreting the program and producing a value. In Yocto-JavaScript, the only kind of value is a function (see § 1.1.1), so we start the implementation of `run()` with the following (we use `throw` as a placeholder for code that has not been written yet to prevent the TypeScript compiler from signaling type errors):

```
type Value = ArrowFunctionExpression;
```

```
function run(expression: Expression): Value {
  throw new Error("NOT IMPLEMENTED YET");
}
```

The first thing that `run()` has to do is determine which type of `expression` it is given:

```
function run(expression: Expression): Value {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "CallExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "Identifier":
      throw new Error("NOT IMPLEMENTED YET");
  }
}
```

In our current example, the `expression` already is a `Value`, so we return it unchanged:

```
// run()
case "ArrowFunctionExpression":
  return expression;
```

1.3.4 A Call Involving Immediate Functions

`[](#a-call-involving-immediate-functions)`

Example Program	Current Output	Expected Output
`(y => y)(x => x)`	`NOT IMPLEMENTED YET`	`x => x`

Interpreting function calls is the main responsibility of our interpreter. There are several techniques to do this and in Step 0 we use one of the simplest: when the interpreter encounters a function call, it substitutes the variable references in the body of the called function with the argument. This is similar to how we reason about functions in mathematics; for example, given the function $f(x) = x + 1$, we begin to calculate $f(29)$ by substituting the references to x in $x + 1$ with the argument 29: $f(29) = 29 + 1$. The implementation of this substitution technique starts in this section and will only be complete in § 1.3.8.

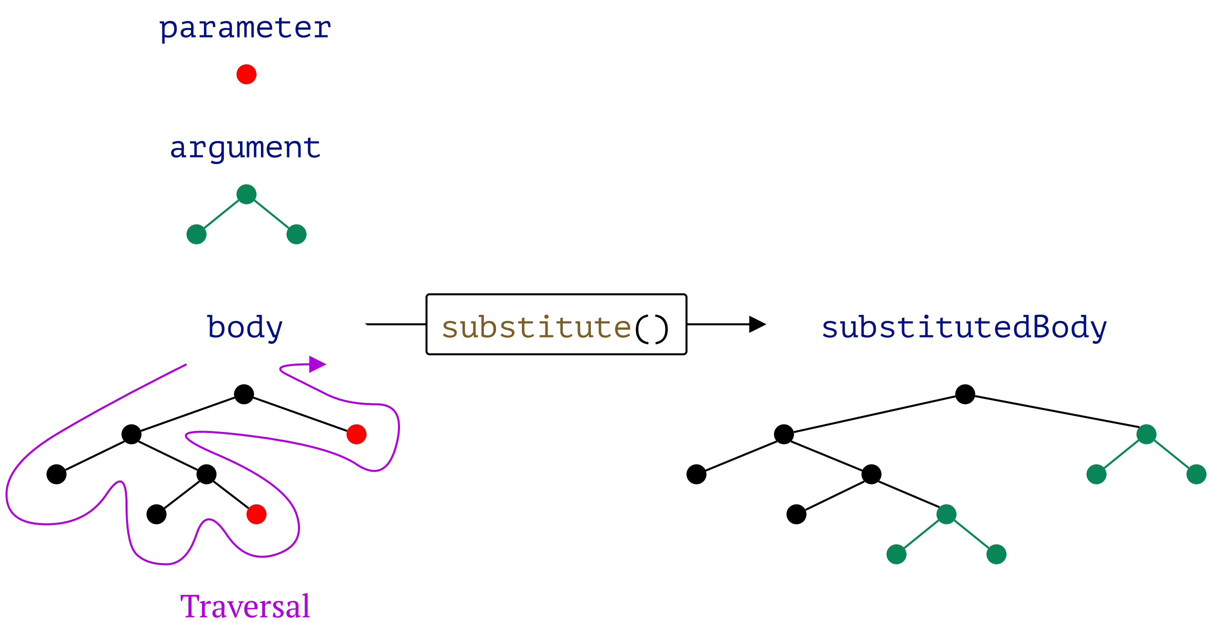
In the example program above, both the function that is called (`y => y`) and the argument (`x => x`) are immediate functions, as opposed to being the result of other operations, so for now we may restrict the interpreter to handle only this case:

```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  throw new Error("NOT IMPLEMENTED YET");
```

Next, we unpack the called function (using something called *destructuring assignment* [20]) and the argument:

```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = expression.arguments[0];
  throw new Error("NOT IMPLEMENTED YET");
```

Finally, we setup an auxiliary function called `substitute()` that implements the traversal of the `body` looking for references to `parameter` and substituting them with the `argument` (for now the result of substitution is restricted to be an `ArrowFunctionExpression`):

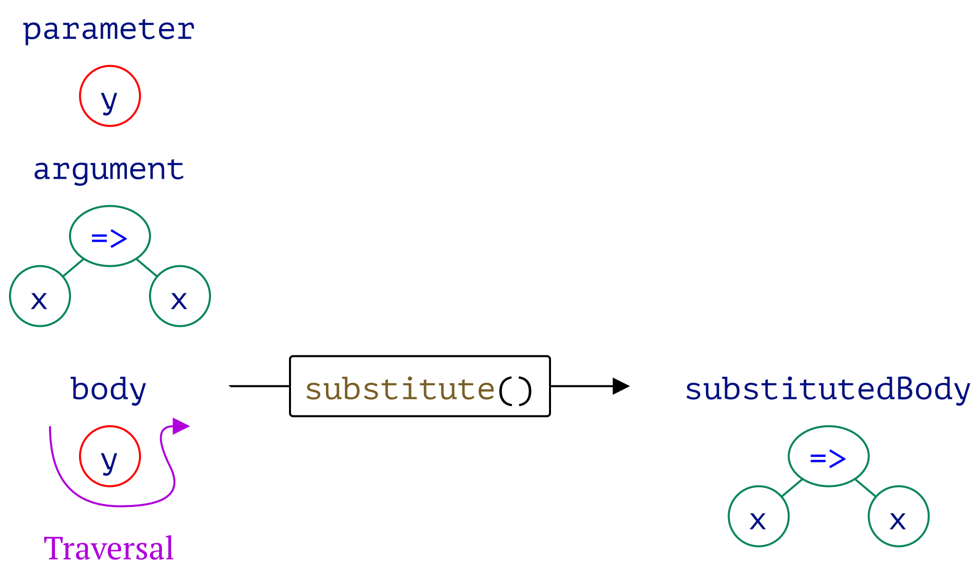



```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = expression.arguments[0];
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    throw new Error("NOT IMPLEMENTED YET");
  }
```

Similar to `run()` itself, `substitute()` starts by determining which type of `expression` is passed to it:

```
function substitute(expression: Expression): Expression {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "CallExpression":
      throw new Error("NOT IMPLEMENTED YET");
    case "Identifier":
      throw new Error("NOT IMPLEMENTED YET");
  }
}
```

In our current example the `expression` is `y`, which is an `Identifier` that must be substituted with the `argument` (`x => x`):



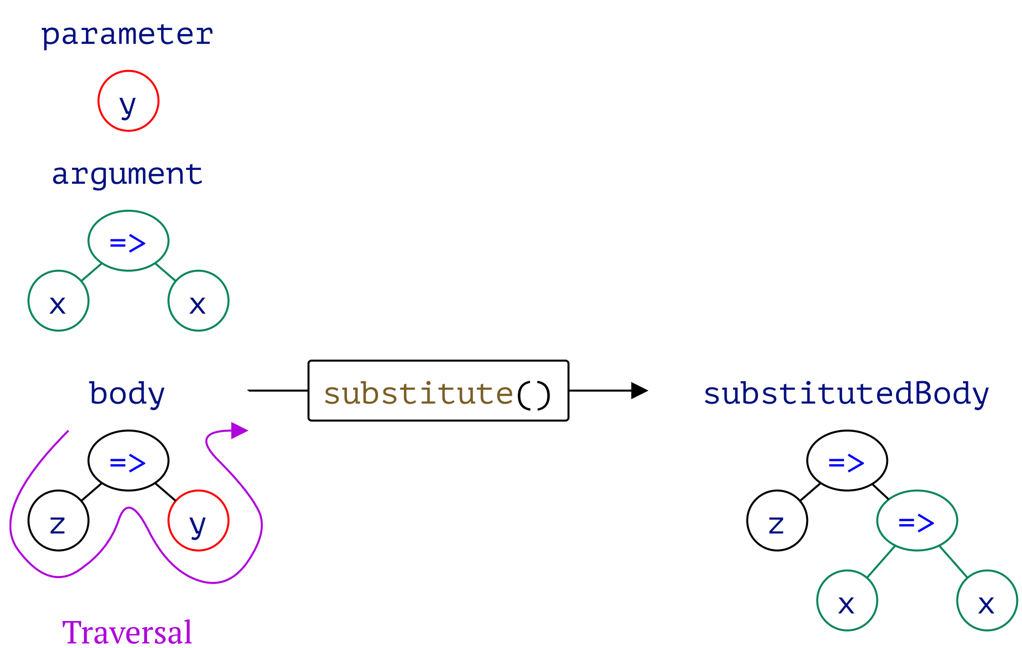

```
// substitute()
case "Identifier":
  return argument;
```

1.3.5 Substitution in Function Definitions

[](#substitution-in-function-definitions)

Example Program	Current Output	Expected Output
`(y => z => y)(x => x)`	`NOT IMPLEMENTED YET`	`z => x => x`

When `substitute()` (see § 1.3.4) starts traversing the `body` of the example above, the `expression` is an `ArrowFunctionExpression` (`z => y`), and we want substitution to proceed deeper to find and substitute `y`, so we call `substitute()` recursively (we use a feature called *spread syntax* [24] to build an `expression` based on the existing one with a new `body`):

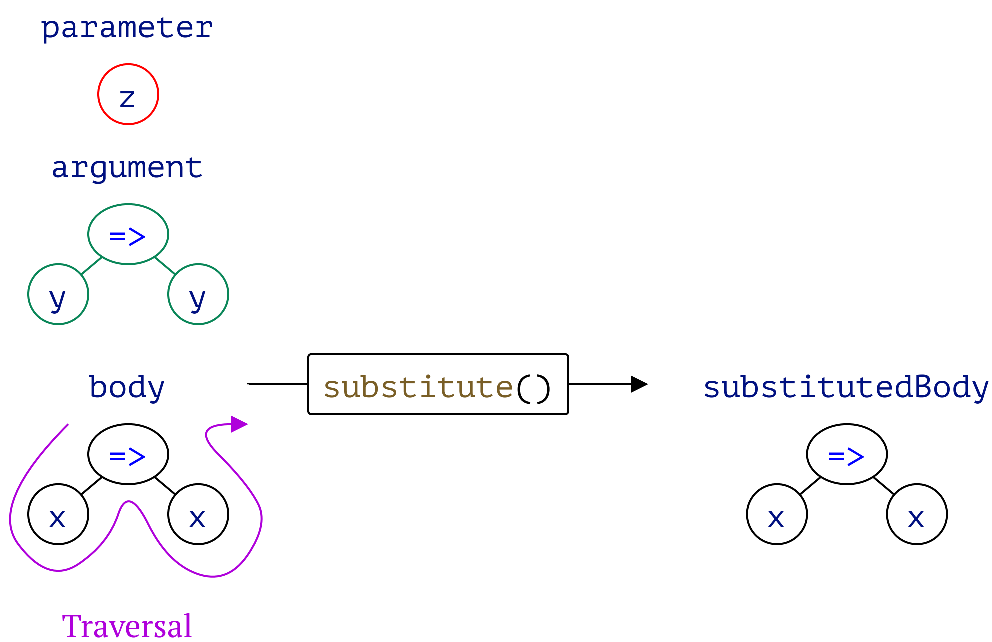



```
// substitute()
case "ArrowFunctionExpression":
  return {
    ...expression,
    body: substitute(expression.body),
  };
```

1.3.6 Name Mismatch [](#name-mismatch)

Example Program	Current Output	Expected Output
`(z => x => x)(y => y)`	`x => (y => y)`	`x => x`

The implementation of `substitute()` introduced in § 1.3.4 *always* substitutes variable references, regardless of whether they refer to the `parameter`. For example, in the program above `substitute()` is substituting the `x` even though the `parameter` is `z`. To fix this, we check whether the variable reference matches the `parameter`, and if it does not then we prevent the substitution by retuning the variable reference unchanged:




```
// substitute()
case "Identifier":
  if (expression.name !== parameter.name) return expression;
  return argument;
```

1.3.7 Name Reuse [](#name-reuse)

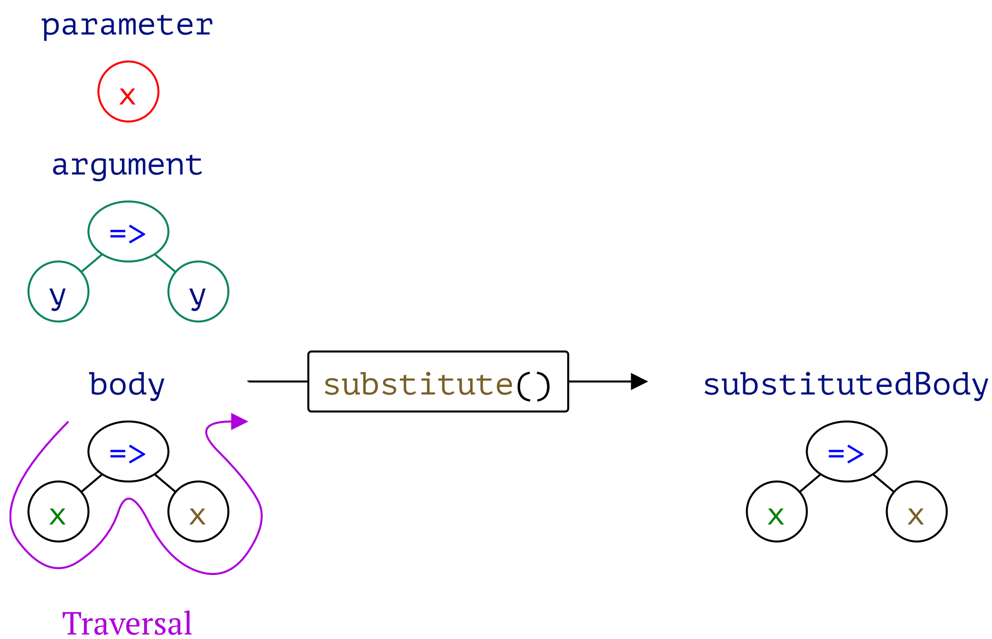
Example Program	Current Output	Expected Output
`(x => x => x)(y => y)`	`x => (y => y)`	`x => x`

In the program above, `x` could refer to either `x` or `x`:

	If `x` Refers to	Then the Output of Example Program Is
Option 1	`x`	`x => (y => y)`
Option 2	`x`	`x => x`

Currently `substitute()` is implementing Option 1, but this leads to an issue: we are unable to reason about `x => x` independently; we must know where it appears and whether a variable called `x` is already defined there.

We avoid this issue by modifying `substitute()` to implement Option 2, which is also the choice of JavaScript and every other popular programming language. We change `substitute()`'s behavior when encountering a function definition so that if the parameter of the function definition matches the parameter that `subsitute()` is looking for, then `subsitute()` returns the function unchanged, preventing further substitution (there is no recursive call to `substitute()` in this case):




```
// substitute()
case "ArrowFunctionExpression":
  if (expression.params[0].name === parameter.name) return expression;
  return {
    ...expression,
    body: substitute(expression.body),
  };
```

Technical Terms

- **Local Reasoning:** The ability to reason about a function without having to know the context under which it is defined. Option 1 defeats local reasoning and Option 2 enables it.
- **Shadowing:** The behavior exhibited by Option 2: `x` is *shadowed* by `x` because there is no way to refer to `x` from the body of the inner function.

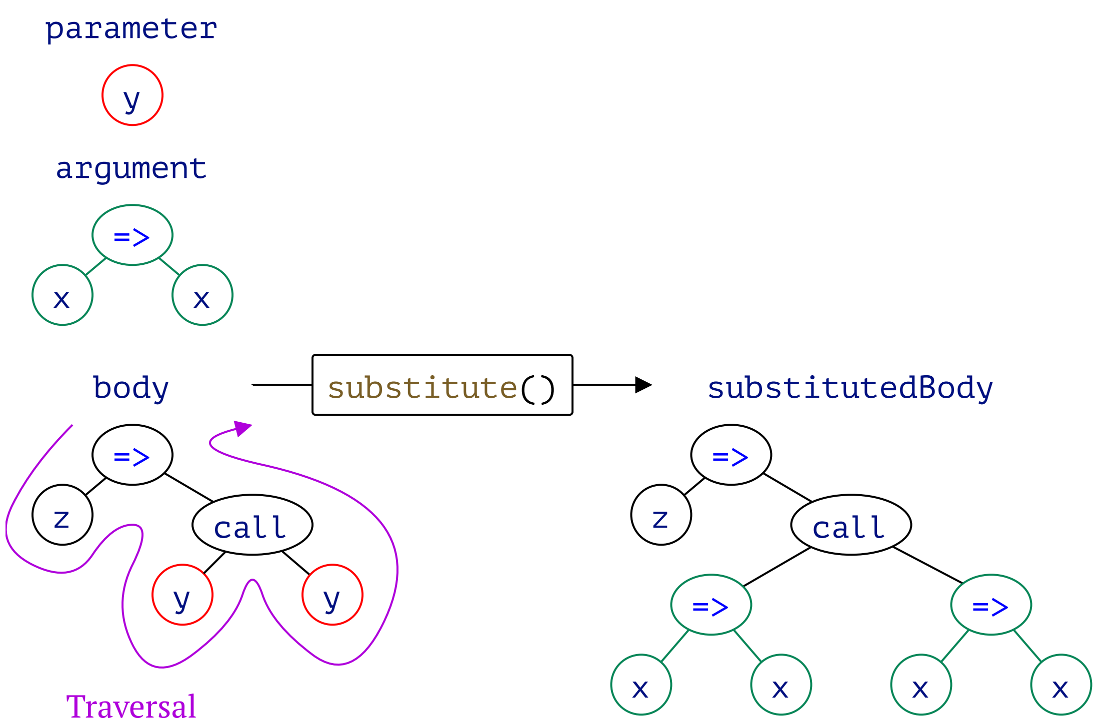
1.3.8 Substitution in Function Calls

[](#substitution-in-function-calls)

Example Program `(y => z => y(y))(x => x)`

Current Output NOT IMPLEMENTED YET
Expected Output z => (x => x)(x => x)

This case is analogous to § 1.3.5 for function calls. The substitute() function must continue traversing the function call recursively:




```
// substitute()
case "CallExpression":
  return {
    ...expression,
    callee: substitute(expression.callee),
    arguments: [substitute(expression.arguments[0])],
  };
```

This concludes the implementation of substitute().

1.3.9 An Argument That Is Not Immediate

[](#an-argument-that-is-not-immediate)

Example Program	`(a => z => a)((y => y)(x => x))`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`z => x => x`

The `arguments` in the example programs from § 1.3.3–§ 1.3.8 are `ArrowFunctionExpressions`, but in general an `argument` may be any kind of `Expression`; for example, in the program above the `argument` is a `CallExpression` `((y => y)(x => x))`. We address the general case by calling `run()` recursively on the `argument` to evaluate it to a `Value`:

```
// run()
case "CallExpression":
  if (expression.callee.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [parameter],
    body,
  } = expression.callee;
  const argument = run(expression.arguments[0]);
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    // ...
  }
```

Technical Terms

- **Big-Step Interpreter [12]:** An interpreter using the technique of calling `run()` recursively to evaluate the `argument`.

Advanced

The notion that the argument is interpreted to produce a value as soon as the function call is encountered characterizes Yocto-JavaScript as a *call-by-value* language [26]. JavaScript and most other popular programming languages are call-by-value as well, but there is one notable exception, Haskell, which is a *call-by-need* language. In a call-by-need language the argument is interpreted only if it is *needed*, for example, if it is used in the function position of another call (see § 1.3.10), or if it is the result of the program (see § 1.3.11). In a call-by-need language the result of the program above would be `z => ((y => y)(x => x))`, and the call `(y => y)(x => x)` would not be computed, because it is not needed. Besides call-by-value and call-by-need, there is yet another policy for when to interpret arguments: *call-by-name*. The policy of call-by-name is similar to call-by-need, except that an argument that is used multiple times could be computed multiple times in a call-by-name language, and it is guaranteed to be computed at most once in a call-by-need language.

1.3.10 A Function That Is Not Immediate

[](#a-function-that-is-not-immediate)

Example Program	`((z => z)(y => y))(x => x)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`x => x`

This is the dual of § 1.3.9 for the called function, and the solution is the same: to call `run()` recursively:

```
// run()
case "CallExpression":
```

```
  const {
    params: [parameter],
    body,
  } = run(expression.callee);
  const argument = run(expression.arguments[0]);
  const substitutedBody = substitute(body);
  if (substitutedBody.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  return substitutedBody;
  function substitute(expression: Expression): Expression {
    // ...
  }
```

1.3.11 Continuing to Run After a Function Call

[](#continuing-to-run-after-a-function-call)

Example Program	`(z => (y => y)(z))(x => x)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`x => x`

This is similar to § 1.3.9 and § 1.3.10: the result of substitution may be not an immediate function but another call, in which case more computation is required. We solve this with yet another recursive call to `run()`:

```
// run()
case "CallExpression":
  const {
    params: [parameter],
    body,
  } = run(expression.callee);
  const argument = run(expression.arguments[0]);
  return run(substitute(body));
  function substitute(expression: Expression): Expression {
    // ...
  }
```

1.3.12 A Reference to an Undefined Variable

[](#a-reference-to-an-undefined-variable)

Example Program	`(y => u)(x => x)`
Current Output	`NOT IMPLEMENTED YET`
Expected Output	`Reference to undefined variable: u`

The only case in which `run()` may encounter a variable reference directly is if the referenced variable is undefined, otherwise `substitute()` would have already substituted it (see § 1.3.4–§ 1.3.11). In this case, we throw an exception:

```
// run()
case "Identifier":
  throw new Error(`Reference to undefined variable:
${expression.name}`);
```

Example Program	**Current Output**	**Expected Output**
`y => u`	`y => u`	`y => u`

If the reference to an undefined variable occurs in the body of a function that is not called, then we do not reach the case addressed in this section and an exception is not thrown. This is consistent with JavaScript's behavior.

1.3.13 The Entire Runner [](#the-entire-runner)

The implementation of the `run()` function is complete:

```
type Value = ArrowFunctionExpression;

function run(expression: Expression): Value {
  switch (expression.type) {
    case "ArrowFunctionExpression":
      return expression;
    case "CallExpression":
      const {
        params: [parameter],
        body,
```

```
      } = run(expression.callee);
      const argument = run(expression.arguments[0]);
      return run(substitute(body));
      function substitute(expression: Expression): Expression {
        switch (expression.type) {
          case "ArrowFunctionExpression":
            if (expression.params[0].name === parameter.name) return expression;
            return {
              ...expression,
              body: substitute(expression.body),
            };
          case "CallExpression":
            return {
              ...expression,
              callee: substitute(expression.callee),
              arguments: [substitute(expression.arguments[0])],
            };
          case "Identifier":
            if (expression.name !== parameter.name) return expression;
            return argument;
        }
      }
    case "Identifier":
      throw new Error(`Reference to undefined variable: ${expression.name}`);
  }
}
```

Advanced

1.3.14 An Operational Semantics for the Interpreter

[](#an-operational-semantics-for-the-interpreter)

What we accomplished in § 1.3.3–§ 1.3.13 is more than defining an interpreter for Yocto-JavaScript; we also defined formally the *meaning* of Yocto-JavaScript programs: an Yocto-JavaScript program means what the interpreter produces for it. The definition of the meaning of programs in a language is something called the *semantics* of the language, and there are several techniques to specify semantics; the technique we have been using so far is known as a *definitional interpreter* [28].

A definitional interpreter has some advantages over other techniques for specifying semantics: it is executable and it is easier to understand for most programmers. But a definitional interpreter also has one disadvantage: to understand the meaning of an Yocto-JavaScript program we have to understand an interpreter written in TypeScript, which is a big language with many complex features. To address this, there are other techniques for defining semantics that do not depend on other programming languages, and in this section we introduce one of them: *operational semantics* [12, 8, 10].

First, we extend the grammar from § 1.1.4 with the notion of values that is equivalent to the type `Value` (see § 1.3.13, line 1):

$$v \quad ::= \quad x \texttt{ => } e \quad \text{Values}$$

Next, we define a *relation* $e \Rightarrow v$ using *inference rules* that are equivalent to the behavior of `run()` (see § 1.3.13, lines 3–36):

Value $$\frac{}{v \Rightarrow v}$$

Function Call $$\frac{e_f \Rightarrow \texttt{(}x_p \texttt{ => } e_b\texttt{)} \quad e_a \Rightarrow v_a \quad e_b[x_p \backslash v_a] \Rightarrow v}{e_f\texttt{(}e_a\texttt{)} \Rightarrow v}$$

Finally, we define a *metafunction* $e[x\backslash v] = e$ that is equivalent to the behavior of `substitute()` (see § 1.3.13, lines 14–32):

$$
\begin{array}{rcll}
(x \texttt{ => } e)[x_p \backslash v_a] & = & x \texttt{ => } (e[x_p \backslash v_a]) & \text{if } x \neq x_p \\
(x_p \texttt{ => } e)[x_p \backslash v_a] & = & x_p \texttt{ => } e & \\
(e_f(e_a))[x_p \backslash v_a] & = & (e_f[x_p \backslash v_a](e_a[x_p \backslash v_a])) & \\
x[x_p \backslash v_a] & = & x & \text{if } x \neq x_p \\
x_p[x_p \backslash v_a] & = & v_a &
\end{array}
$$

1.3.15 Parser [](#parser)

The parser is responsible for converting a string representing an Yocto-JavaScript program into data structures that are more convenient for the runner to manipulate (see § 1.3.1 for a high-level view of the architecture and § 1.3.2 for the definition of the data structures). We choose data structures that are compatible with Babel [2], which is a library to manipulate JavaScript programs that we use to implement the Yocto-JavaScript parser and the generator (see § 1.3.16).

Our strategy to implement the Yocto-JavaScript parser is to delegate most of the work to Babel and check that the input program is using only features supported by Yocto-JavaScript. The following is the full implementation of the parser:

```
function parse(input: string): Expression {
  const expression = babelParser.parseExpression(input);
  babelTypes.traverse(expression, (node) => {
    switch (node.type) {
      case "ArrowFunctionExpression":
        if (node.params.length !== 1)
          throw new Error(
            "Unsupported Yocto-JavaScript feature: ArrowFunctionExpression doesn’t have exactly one parameter"
          );
        if (node.params[0].type !== "Identifier")
```

```
11           throw new Error(
12             "Unsupported Yocto-JavaScript feature: ArrowFunctionExpression param isn't an Identifier"
13           );
14         break;
15       case "CallExpression":
16         if (node.arguments.length !== 1)
17           throw new Error(
18             "Unsupported Yocto-JavaScript feature: CallExpression doesn't have exactly one argument"
19           );
20         break;
21       case "Identifier":
22         break;
23       default:
24         throw new Error(`Unsupported Yocto-JavaScript feature: ${node.type}`);
25     }
26   });
27   return expression as Expression;
28 }
```

- **Line 1:** The parser is defined as a function called `parse()`, which receives a `string` called `input` that represents a program and returns an `Expression` (see § 1.3.2).

- **Line 2:** Call `babelParser.parseExpression()`, which parses the `input` as a JavaScript program and produces data structures of the Babel types [3]. The `babelParser.parseExpression()` function throws an exception if there is a syntax error (for example, the missing function body in the program `x =>`) or if the `input` is not a simple JavaScript expression, and therefore is not supported by Yocto-JavaScript (for example, `x => x; y => y`, which is a sequence of two expressions, and `const f = x => x`, which is a variable declaration).

- **Lines 3–26:** Traverse the `expression` produced by Babel to check that the input program uses only the features supported by Yocto-JavaScript. This traversal is similar to what happens in `substitute()` (see § 1.3.13, lines 14–32), but we use the Babel auxiliary function `babelTypes.traverse()` to drive it. The data structure fragments are called `node`s because they form something called the *Abstract Syntax Tree* (AST) of the program (see § 1.3.2).

- **Line 6:** Check that a function definition has exactly one parameter (see § 1.1.1). This rejects programs such as `() => x` and `(x, y) => x`.
- **Line 10:** Check that the parameter in a function definition is a variable instead of a pattern for destructuring assignment [20]. This rejects programs such as `([x, y]) => x`, in which the parameter is the array pattern `[x, y]`.
- **Line 16:** Check that a function call has exactly one argument (see § 1.1.2). This rejects programs such as `f()` and `f(a, b)`.
- **Line 21:** An `Identifier` is accept unconditionally.
- **Line 23:** Any kind of `node` that has not been explicitly accepted above is reject as unsupported in Yocto-JavaScript. This rejects programs such as `29`, which is a `NumericLiteral`.
- **Line 27:** We convert the type of `expression` from a Babel `Expression` into an Yocto-JavaScript `Expression` (see § 1.3.2). This is safe because of the checks performed above.

In later Steps almost everything about the interpreter will change, but the parser will remain the same.

1.3.16 Generator [](#generator)

The generator transforms a `Value` produced by `run()` into a human-readable format (see § 1.3.1 for a high-level view of the architecture). Similar to what happened in the parser (see § 1.3.15), we delegate most of the implementation to Babel [2]. The following is the full implementation of the generator:

```
function generate(value: Value): string {
  return babelGenerator.default(value as any).code;
}
```

We convert the `value` from the Yocto-JavaScript `Value` type into the `any` type to sidestep the TypeScript type checker. This conversion is safe because the Yocto-JavaScript `Value` type is compatible with the parts of the Babel `Node` type that the `babelGenerator.default()` function needs.

1.3.17 Programs That Do Not Terminate

[](#programs-that-do-not-terminate)

Example Program	Current Output	Expected Output
`(f => f(f))(f => f(f))`	Does not terminate	Does not terminate

Technical Terms

- **Ω-Combinator:** The example program: `(f => f(f))(f => f(f))`.
- **U-Combinator:** The function `f => f(f)` that is part of the Ω-combinator ($\Omega = (U)(U)$).

Yocto-JavaScript may express any program that a computer may run (see § 1.1.3), including some programs that do not terminate. For example, consider the program above; the following is a trace of the first call to `run()` (see § 1.3.13) on this program:

Line			
3	`expression`	=	`(f => f(f))(f => f(f))`
9	`parameter`	=	`f`
10	`body`	=	`f(f)`
12	`argument`	=	`f => f(f)`
13	`substitute(body)`	=	`(f => f(f))(f => f(f))`

This causes the `run()` function to go into an infinite loop because the result of substitution that is passed as argument to the recursive call to `run()` in line 13 is the same as the initial `expression`.

Example Program	`(f => (f(f))(f(f)))(f => (f(f))(f(f)))`
Current Output	Does not terminate
Expected Output	Does not terminate

There are also programs for which interpretation does not terminate that never produce the same expression twice. For example, consider the program above, which is a variation of the first program in which every variable reference to `f` has been replaced with `f(f)`. The following is a trace of the first call to `run()`:

Line			**where** `F = f => (f(f))(f(f))`
3	`expression`	=	`F(F)`
9	`parameter`	=	`f`
10	`body`	=	`(f(f))(f(f))`
12	`argument`	=	`F`
13	`substitute(body)`	=	`(F(F))(F(F))`

The result of substitution (`(F(F))(F(F))`) is an expression that contains the initial expression (the first `F(F)`) in addition to some extra work (the second `F(F)`), so when it is passed as argument to the recursive call to `run()` in line 13, it causes `run()` to go into an infinite loop. Unlike what happened in the first example, when interpreting this program the expressions that are passed to `run()` never repeat themselves:

```
(F(F))
(F(F))(F(F))
(F(F))(F(F))(F(F))
(F(F))(F(F))(F(F))(F(F))
          ⋮
```

Non-termination is what we expect from an interpreter, but not from an analyzer, and as the second example demonstrates, preventing non-termination is not as simple as checking whether `run()` is being called with the same expression multiple times. As we move forward from an interpreter to an analyzer in the next Steps one of the main issues we address is termination: even if it takes a long time, an analyzer must eventually terminate regardless of the program on which it is working.

Advanced

Detecting non-termination in an interpreter without losing any information about the original program is a problem that cannot be solved, regardless of the sophistication of the detector and the computational power available to it. The problem, which is known as the *halting problem*, is said to be *uncomputable* [31 (§ 8)], and is a direct consequence of the Turing completeness of Yocto-JavaScript (see § 1.1.3). In our analyzer we will guarantee termination by allowing some information to be lost.

1.4 Step 1: Environment-Based Interpreter

[](#step-1-environment-based-interpreter)

The interpreter in Step 0 may not terminate for some programs, and preventing non-termination is one of the main issues we must address when developing an analyzer (see § 1.3.17). The source of non-termination in Step 0 is substitution, which may produce infinitely many new expressions and cause the interpreter to loop forever. In Step 1, we modify the interpreter so that it does not perform substitution, and as a consequence it considers only the finitely many expressions found in the input program. The interpreter in Step 1 may still not terminate, but that is due to other sources of non-termination that will be addressed in subsequent Steps.

1.4.1 Avoiding Substitution by Introducing Environments

[](#avoiding-substitution-by-introducing-environments)

When the interpreter from Step 0 encounters a function call, it produces a new expression by traversing the body of the called function and substituting the references to the parameter with the argument (see § 1.3.5), for example:

Example Program	`(y => z => y)(x => x)`
Step 0 Output	`z => x => x`

The issue with this strategy is that the expression `z => x => x` does not exist in the original program, and as mentioned in § 1.3.17, it is possible that the interpreter tries to produce infinitely many new expressions and loops forever. In Step 1 we want to avoid producing new expressions, so that the interpreter has to consider only the finitely many expressions found in the original program. We accomplish this by introducing a map from variables to the values with which they should be substituted: when we encounter a function call, we add to the map; and when we encounter a variable reference, we look it up on the map, for example:

Example Program	`(y => z => y)(x => x)`
Step 1 Output	Function: `z => y` Environment: `{ "y": ``x => x`` }`

Technical Terms

- **Environment:** A map from variables to the values with which they should be substituted, for example, `{ "y": `x => x` }`.

The following is the data structure used to represent environments:

```
type Environment = Map<Identifier["name"], Value>;
```

1.4.2 Setting up an Environment on the Runner

[](#setting-up-an-environment-on-the-runner)

The runner needs to maintain an environment, so we modify the implementation of `run()` from § 1.3.13 to introduce an auxiliary function called `step()` that receives an `environment` as an extra parameter:

```
function run(expression: Expression): Value {
  return step(expression, new Map());
  function step(expression: Expression, environment: Environment): Value {
    switch (expression.type) {
      case "ArrowFunctionExpression":
        return expression;
      case "CallExpression":
        const {
          params: [parameter],
          body,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        return step(substitute(body), environment);
```

```
14        function substitute(expression: Expression): Expression {
15          switch (expression.type) {
16            case "ArrowFunctionExpression":
17              if (expression.params[0].name === parameter.name)
18                return expression;
19              return {
20                ...expression,
21                body: substitute(expression.body),
22              };
23            case "CallExpression":
24              return {
25                ...expression,
26                callee: substitute(expression.callee),
27                arguments: [substitute(expression.arguments[0])],
28              };
29            case "Identifier":
30              if (expression.name !== parameter.name) return expression;
31              return argument;
32          }
33        }
34      case "Identifier":
35        throw new Error(`Reference to undefined variable: ${expression.name}`);
36    }
37  }
38 }
```

- **Line 3:** We define the `step()` auxiliary function that receives an `environment` as an extra parameter.
- **Line 2:** The `environment` starts empty.
- **Lines 11–13:** The recursive calls to `run()` are changed to recursive calls to `step()` and the `environment` is propagated.

With these modifications the environment is available to the runner, but it is not used for anything yet; it is propagated through the recursive calls but remains empty and is never looked up.

1.4.3 Using the Environment [](#using-the-environment)

Example Program	Current Output	Expected Output
`(y => y)(x => x)`	`x => x`	`x => x`

In § 1.4.2 we setup the environment on the runner, but did not use it for anything. We now modify the runner to use the environment:

```
1  function run(expression: Expression): Value {
2    return step(expression, new Map());
3    function step(expression: Expression, environment: Environment): Value {
4      switch (expression.type) {
5        case "ArrowFunctionExpression":
6          return expression;
7        case "CallExpression":
8          const {
9            params: [parameter],
10           body,
11         } = step(expression.callee, environment);
12         const argument = step(expression.arguments[0], environment);
13         return step(body, new Map(environment).set(parameter.name, argument));
14       case "Identifier":
15         const value = environment.get(expression.name);
16         if (value === undefined)
17           throw new Error(
18             `Reference to undefined variable: ${expression.name}`
19           );
20         return value;
21     }
22   }
23 }
```

- **Line 13:** Remove substitution. Instead, when encountering a function call, add to the environment a mapping from the parameter to the argument.
- **Lines 15–18:** When encountering a variable reference, look it up on the environment.

1.4.4 Introducing Closures [](#introducing-closures)

Example Program	`(y => z => y)(x => x)`
Current Output	`z => y`
Expected Output	Function: `z => y` Environment: `{ "y": `x => x` }`

With the modifications introduced in § 1.4.3 the interpreter returns a function in which substitutions have not occurred yet, and to make sense of this function we need the environment to be returned as well:

Technical Terms

- **Closure [13]:** A data structure containing a function and an environment, for example:

  ```
  {
    "function": `z => y`,
    "environment": { "y": `x => x` }
  }
  ```

```
type Value = Closure;
type Closure = {
  function: ArrowFunctionExpression;
  environment: Environment;
};
function run(expression: Expression): Value {
  return step(expression, new Map());
  function step(expression: Expression, environment: Environment):
Value {
    switch (expression.type) {
      case "ArrowFunctionExpression":
        return { function: expression, environment };
      case "CallExpression":
        const {
          function: {
            params: [parameter],
            body,
```

```
          },
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        return step(body, new Map(environment).set(parameter.name,
argument));
      case "Identifier":
        const value = environment.get(expression.name);
        if (value === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return value;
    }
  }
}
```

The notion of what constitutes a *value* in Step 1 is different from that of Step 0: while in Step 0 the interpreter produced a *function*, now it produces a *closure*. This closure may be used to recreate the output of Step 0 by substituting the variable references in the function body with the corresponding values from the closure's environment. We may do this to check that the outputs of the interpreters are equivalent.

Alternative Argument

Another way to reason about an environment-based interpreter is that it is a substitution-based interpreter in which the substitutions are *delayed* until needed.

Implementation Details

The `Map` data structure is provided by a JavaScript package developed by the author called Collections Deep Equal [6]. A `Map` is similar to a native JavaScript `Map` [23], but the keys are compared differently: on a `Map` the keys are compared by whether they are the same reference to the same object, and on a `Map` the keys are compared by whether they are objects with the same keys and values, for example:

```
> const anObject = { age: 29 }
> const anotherObjectWithTheSameKeysAndValues = { age: 29 }
> const aValue = "Leandro"
> new Map().set(anObject, aValue)
           .get(anotherObjectWithTheSameKeysAndValues)
undefined
> new Map().set(anObject, aValue)
                    .get(anotherObjectWithTheSameKeysAndValues)
"Leandro"
```

1.4.5 A Function Definition [](#a-function-definition)

\begin{tabular}{ll} \textbf{Example Program} & `x => x` \\ \textbf{Current Output} & — \\ \textbf{Expected Output} & ⟨ js(`x => x`), [] \rangle` \\ \end{tabular}

When the interpreter encounters a function definition, it captures the current `environment` in a closure:

```
// step()
case "ArrowFunctionExpression":
  return { function: expression, environment };
```

1.4.6 A Function Call [](#a-function-call)

\begin{tabular}{ll} \textbf{Example Program} & `(x => z => x)(y => y)` \\ \textbf{Current Output} & — \\ \textbf{Expected Output} & ⟨ js(`z => x`) ``, [ `` jsx \mapsto \langle js(`y => y`), [] \rangle] \rangle` \\ \end{tabular}

First, we remove `substitute()`, which is the goal of Step 1:

```
// step()
case "CallExpression":
  const {
    params: [parameter],
    body,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(body, environment);
```

Next, we fix the pattern that matches the result of the interpretation of the called function to take in account the closure:

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
    environment: functionEnvironment,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(body, environment);
```

Finally, we modify the recursive call to `step()` that interprets the function body so that it receives a new augmented `environment` including a mapping from the `parameter` (for example, `x`) to the `argument` (for example, 〈 js(y => y), [] \rangle`):

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
```

```
8       environment: functionEnvironment,
9     } = step(expression.callee, environment);
10    const argument = step(expression.arguments[0], environment);
11    return step(
12      body,
13      new Map(environment).set(parameter.name, argument)
14    );
```

1.4.7 Name Reuse [](#name-reuse-1)

\begin{tabular}{ll} \textbf{Example Program} & `(x => x => z => x)(a => a)(y => y)` \\textbf{Current Output} & ⟨ js`(z => x)` ``, [ `` jsx \mapsto \langle js`(y => y)` ``, [] \rangle] \rangle `` \\ \textbf{Expected Output} & `` math\langle `(z => x)`, [`x` \mapsto \langle `(y => y)`, [] \rangle] \rangle `` \ \end{tabular}

If a name is reused (for example, `x` in the example program above), then the second time it is encountered by `step()` it is overwritten in the `environment` (see the call to `set()` in line 13 of § \ref{A Function Call}, which overwrites an existing map key). This causes the variable reference to `x` to refer to the second (inner) `x`, which is the expected behavior (it is what we called Option 2 in § \ref{Step 0: Name Reuse}).

1.4.8 A Variable Reference [](#a-variable-reference)

\begin{tabular}{ll} \textbf{Example Program} & `(y => y)(x => x)` \ \textbf{Current Output} & — \ \textbf{Expected Output} & ⟨ js`(y => y)` ``, [] \rangle `` \\ \\ \textbf{Example Program} & `` js(x => y)(y => y) \\ \textbf{Current Output} & — \\ \textbf{Expected Output} & textReference to undefined variable: y `` \\ \\

\textbf{Example Program} & `` jsx => y \\ \textbf{Current Output} & – \\ \textbf{Expected Output} & math\langle `` js(x => y) , [] \rangle \\end{tabular}

When we encounter a variable reference, we look it up in the current environment:

```
// step()
case "Identifier":
  const value = environment.get(expression.name);
  if (value === undefined)
    throw new Error(
      `Reference to undefined variable: ${expression.name}`
    );
  return value;
```

1.4.9 A Function Body Is Evaluated with the Environment in Its Closure

[](#a-function-body-is-evaluated-with-the-environment-in-its-closure)

\begin{tabular}{ll} \textbf{Example Program} & \\multicolumn{2}{l}{(f => (x => f(x))(a => a))((x => z => x)(y => y))} \\textbf{Current Output} & ⟨ js(a => a) ``, [ `` jsf \mapsto \langle js(z => x) ``, [ `` jsx \mapsto \langle js(y => y) ``, [] \rangle ] \rangle] \rangle `` \\ \textbf{Expected Output} & `` math\langle (y => y), [] \rangle `` \\end{tabular}

This example program shows the difference between the current environment with which an expression is evaluated and the environment that comes from a closure. The following is a trace of the first call to `step()`, when a closure is created:

\begin{tabular}{rrcl} \multicolumn{4}{c}{\textbf{Trace 1: First Call to `step()` · Closure Creation}} \\ \textbf{Line} & \multicolumn{1}{l}{(see § \ref{A Function Call})} & & \ & `expression.callee` & = & `f => (x => f(x))(a => a)` \ & `expression.arguments[0]` & = & `(x => z => x)(y => y)` \\rowcolor[rgb]{.88,1,1} 10 & `argument` & = & ⟨ js`(z => x)` ``, [ `` jsx \mapsto \langle js`(y => y)`, [] \rangle] \rangle` \\end{tabular}

And the following is a trace of the recursive call to `step()` in which that closure is called:

\begin{tabular}{rrcl} \multicolumn{4}{c}{\textbf{Trace 2: Recursive Call to `step()` · Closure Call}} \\ \textbf{Line} & \multicolumn{1}{l}{(see § \ref{A Function Call})} & & \ & `expression` & = & `f(x)` \ & `expression.callee` & = & `f` \\rowcolor[rgb]{.88,1,1} & `expression.arguments[0]` & = & `x` \\rowcolor[rgb]{.88,1,1} & `environment` & = & [jsx `` \mapsto \langle `` js(a => a) , [\cdots] \rangle, \cdots] \ 9 & tsstep(``\cdots `` ts) & = & math\langle `` js (z => x) , [js x\mapsto \langlejs (y => y) , [] \rangle] \rangle \ 5 & ts `parameter `& = &` js` z \\rowcolor[rgb]{.88,1,1} 6 & ts `body `& = &` js` x \\rowcolor[rgb]{.88,1,1} 8 & ts `functionEnvironment `& = &` math` [js x\mapsto \langlejs(y => y) , [] \rangle] \\multicolumn{4}{c}{Paused before line 10} \ \end{tabular}

At this point, there are two expressions left to evaluate: the argument (`expression.arguments[0]`; line 10), and the body of the called function (`body`; lines 11–14). Both of these expressions have the same code (`x`), and our current implementation looks up this variable both times on the current `environment`, which produces the same value: ⟨ js`(a => a)`, [\cdots] \rangle`.

But this leads to an issue: we may not reason about `z => x` by looking only at where it is defined; we must also examine all the places in which it may be called. This is the same issue we had to solve when considering name reuse in Step 0 (see § \ref{Step 0: Name Reuse}). We would like, instead, for each `x` to refer to the value that existed in the environment where the closure is *created*, not where it is *called*:

\includegraphics[page = 8]{images.pdf}

To implement this, we change the recursive call to `step()` that evaluates the function body so that it uses the environment coming from the closure (`functionEnvironment`) instead of the current environment (`environment`):

```
// step()
case "CallExpression":
  const {
    function: {
      params: [parameter],
      body,
    },
    environment: functionEnvironment,
  } = step(expression.callee, environment);
  const argument = step(expression.arguments[0], environment);
  return step(
    body,
    new Map(functionEnvironment).set(parameter.name, argument)
  );
```

Technical Terms

The principle of being able to reason about a function only by looking at its definition is something called *local reasoning* (see § \ref{Step 0: Name Reuse}). The treatment given to the environment before this section is something called *dynamic scoping*, because the *scope* of a variable (where a variable is defined) is *dynamic*, depending on where the function is called. The treatment given to the environment in this section is something called *static scoping* or *lexical scoping*, because the scope of a variable is determined before we start interpreting the program.

Advanced

There are languages that implement dynamic scoping. In some cases dynamic scoping is the only option, for example, in the original implementation of LISP [16], though that was later considered a mistake [15]. In some cases dynamic scoping is the default, but there is an option to use static scoping, for example, in Emacs Lisp [14 (§ 12.10)]. In some cases dynamic scoping is an extra feature to be used sparingly, for example, in Racket's `parameterize` [9 (§ 4.13)].

1.4.10 The Entire Runner [](#the-entire-runner-1)

We completed the changes necessary to transform the `run()` function from the substitution-based interpreter in Step 0 into an environment-based interpreter:

```
type Value = Closure;

type Closure = {
  function: ArrowFunctionExpression;
  environment: Environment;
};

type Environment = Map<Identifier["name"], Value>;

function run(expression: Expression): Value {
  return step(expression, new Map());
  function step(expression: Expression, environment: Environment):
```

```
  Value {
    switch (expression.type) {
      case "ArrowFunctionExpression":
        return { function: expression, environment };
      case "CallExpression":
        const {
          function: {
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        return step(
          body,
          new Map(functionEnvironment).set(parameter.name, argument)
        );
      case "Identifier":
        const value = environment.get(expression.name);
        if (value === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return value;
    }
  }
}
```

Advanced

1.4.11 Operational Semantics [](#operational-semantics)

We adapt the operational semantics from § 1.3.14 to the interpreter defined in Step 1. First, we change the notion of values:

\begin{tabular}{rcll} v & = & ⟨ js(``x `` js => e js) ``, \rho \rangle `` & Values / Closures \\ `` math\rho & = & math`{x

\mapsto v, \cdots} `` & Environments \ \end{tabular}

We then define the relation $\rho \vdash e \Rightarrow v$ to be equivalent to the new implementation of `run()`:

\begin{mathpar} \inferrule { } {\rho \vdash `(x => e)` \Rightarrow \langle `(x => e)`, \rho \rangle}

\inferrule { \rho \vdash e_f \Rightarrow \langle `(x => e_b)`, \rho_f \rangle \ \rho \vdash e_a \Rightarrow v_a \ \rho_f \cup {x \mapsto v_a} \vdash e_b \Rightarrow v \ } {\rho \vdash `e_f(e_a)` \Rightarrow v}

\inferrule { } {\rho \vdash x \Rightarrow \rho(x)} \end{mathpar}

1.4.12 Generator `[](#generator-1)`

We modify the generator from § 1.3.16 to support closures. For example, the following is the representation of the closure from § \ref{A Function Call}:

\langle js`(z => x)` ``, [ `` `jsx \mapsto \langle` js`(y => y)`, [] \rangle]

\rangle`

```json
{
  "function": "z => x",
  "environment": [
    [
      "x",
      {
        "function": "y => y",
        "environment": []
      }
    ]
  ]
}
```

The following is the modified implementation of `generate()`:

```
1  function generate(value: any): string {
2    return JSON.stringify(
3      value,
4      (key, value) => {
5        if (value.type !== undefined)
6          return prettier
7            .format(escodegen.generate(value), {
8              parser: "babel",
9              semi: false,
10             arrowParens: "avoid",
11           })
12           .trim();
13       return value;
14     },
15     2
16   );
17 }
```

\begin{description} \item [Line 1:]

Change the input type from `Value` to `any`, because this implementation of `strinfigy()` supports any data structure, including the data structures we will define in later Steps.

\item [Line 2:]

Call `JSON.stringify()` [22], which traverses any data structure and converts it into a string.

\item [Line 4:]

Provide a `replacer` that is responsible for converting data structures that represent Yocto-JavaScript programs into strings.

\item [Line 5:]

Check whether a data structure represents an Yocto-JavaScript program by checking the existence of a field called `type` (see § 1.3.2), in which case we use the previous implementation of `generate()` (see § 1.3.16) to produce a string.

\item [Line 15:]

Format the output with indentation of two spaces. \end{description}

This implementation of `generate()` supports not only closures but any data structure (because of `JSON.stringify()`), so it will remain the same in the following Steps.

1.4.13 Programs That Do Not Terminate

`[](#programs-that-do-not-terminate-1)`

The programs that do not terminate in Step 0 (see § 1.3.17) do not terminate in Step 1 either, because the interpreters are equivalent except for the technique used to interpret function calls, but the sources of non-termination are different. In Step 0 substitution may produce infinitely many expressions, including expressions that do not occur in the original program. In Step 1 the interpreter considers only the finitely many expressions that occur in the original program, but it may produce infinitely many environments.

This difference is significant because there are programs that produce infinitely many different expressions in Step 0, but produce the same expression and environment repeatedly in Step 1, and in these cases we could detect non-termination by checking whether the runner is in a loop with the same arguments, for example:

\begin{tabular}{ll} \multicolumn{2}{c}{`(F(F))`, where `F = f => (f(f))(f(f))`}\\multicolumn{1}{c}{\textbf{Step 0}} & \multicolumn{1}{c}{\textbf{Step 1}}\ `(F(F))` & ⟨ js`(F(F)) ``, [ `` jsf \mapsto \langle` jsF ` ``, [] \rangle] \rangle `` \\ `` ` js(F(F))(F(F)) &

math\langle `` js(F(F)) , [ jsf `` \mapsto \langle `` jsF , [] \rangle] \rangle\ (F(F))(F(F))(F(F)) & ⟨ js(F(F)) ``, [`` jsf \mapsto \langle jsF ``, [] \rangle] \rangle `` \\ `` js(F(F))(F(F))(F(F))(F(F)) & math\langle `` js(F(F)) , [ jsf `` \mapsto \langle `` jsF , [] \rangle] \rangle\\multicolumn{1}{c}{$\vdots$} & \multicolumn{1}{c}{$\vdots$}\\end{tabular}

But this strategy is insufficient to guarantee termination, because there are programs that do not terminate which produce infinitely many different environments, for example:

\begin{tabular}{l}\multicolumn{1}{c}{(f => c => f(f)(x => c))(f => c => f(f)(x => c))(y => y)}\\multicolumn{1}{c}{or F(F)(y => y), where F = f => c => f(f)(C) and C = x => c}\ ⟨ jsf(f)(C) ``, [ `` jsc \mapsto \langle js(y => y) ``, [] \rangle, \cdots] \rangle `` \\ `` math\langle f(f)(C), [c \mapsto \langle C, [c \mapsto \langle (y => y), [] \rangle, \cdots] \rangle, \cdots] \rangle \\ math\langle `` jsf(f)(C) , [jsc `` \mapsto \langle `` jsC , [jsc `` \mapsto \langle `` jsC , [jsc `` \mapsto \langle `` js(y => y) , [] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle\ ⟨ jsf(f)(C) ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle js(y => y) ``, [] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle `` \\ \multicolumn{1}{c}{`` math\vdots ``}\\end{tabular}

The program above is a variation on the shortest non-terminating program, (f

`=> f(f))(f => f(f))`, in which each `f => f(f)` receives an additional parameter `c`, and each `f(f)` receives an additional argument `x => c`.

The interpreter in Step 1 may produce infinitely many different environments because environments may be nested. That is the issue that we address in Step 2.

Technical Terms

The nesting of environments in Step 1 characterizes them as something called *recursive data structures*: data structures that may contain themselves. The data structures used to represented Yocto-JavaScript programs are recursive as well (see § 1.3.2).

1.5 Step 2: Store-Based Interpreter

`[](#step-2-store-based-interpreter)`

The source of non-termination in Step 1 is the nesting of the environments (see § \ref{Step 1: Programs That Do Not Terminate}). In Step 2, we address this issue by introducing a layer of indirection between a name in the environment and its corresponding value. The interpreter in Step 2 may still not terminate, but that is due to other sources of non-termination that will be addressed in subsequent Steps.

1.5.1 Avoiding Nested Environments by Introducing a Store

`[](#avoiding-nested-environments-by-introducing-a-store)`

In Step 1 a closure contains an environment mapping names to other closures, which in turn contain their own environments mapping to yet more closures. In

Step 2, we remove this circularity by introducing a layer of indirection: an environment maps names to *addresses*, which may be used to lookup values in a *store*, for example:

\begin{tabular}{rcc} (See § \ref{A Variable Reference}) & \textbf{Step 1} & \textbf{Step 2} \\ \textbf{Variable Reference} & x & x \\ \textbf{Environment} & [jsx `` \mapsto \langle `` js(y => y) , [] \rangle] & [jsx `` \mapsto `` js0] \\ \textbf{Store} & — & [js0 `` \mapsto \langle `` js(y => y) , [] \rangle] \\ \textbf{Value} & ⟨ js(y => y) ``, [] \rangle `` & `` math\langle (y => y), [] \rangle `` \\ \end{tabular}

Each closure continues to include its own environment, because it needs to look up variable references from where the closure was created (see § \ref{A Function Body Is Evaluated with the Environment in Its Closure}), but there is only one store for the entire interpreter, and we avoid ambiguities by allocating different addresses, for example:

\begin{tabular}{rll} (See § \ref{A Function Body Is Evaluated with the Environment in Its Closure}) & \multicolumn{1}{c}{\textbf{Step 1}} & \multicolumn{1}{c}{\textbf{Step 2}} \ environment & [jsx `` \mapsto \langle `` js(a => a) , [\cdots] \rangle, \cdots] & [jsx `` \mapsto `` js0 , \cdots] \ funcEnv. & [jsx `` \mapsto \langle `` js(y => y) , [] \rangle] & [jsx `` \mapsto `` js1] \\ \textbf{Store} & \multicolumn{1}{c}{—} & [js0 `` \mapsto \langle `` js(a => a) , [\cdots] \rangle, \ & & js1 `` \mapsto \langle `` js(y => y) , [] \rangle] \end{tabular}

The runner must return the store along with the value, for the variable references to be looked up, for example:

\begin{tabular}{ll} \textbf{Example Program} (see § \ref{A Function Call}) & (x => z => x)(y => y) \\textbf{Step 1 Output} & ⟨ js(z => x) ``, [ `` jsx \mapsto \langle js(y => y) ``, [] \rangle] \rangle `` \\ \textbf{Step 2 Output} & `` tsvalue = math\langle `` js(z => x) , [jsx `` \mapsto `` js0] \rangle \ & store = [js0 `` \mapsto \langle `` js(y => y) , [] \rangle] \\end{tabular}

Advanced

The technique used in Step 2 is related to the run-time environments that are the target of traditional compilers for languages such as C [1 (§ 7)]: an environment corresponds to some of the data stored in an activation frame on the call stack, and the store corresponds to the heap.

1.5.2 New Data Structures [](#new-data-structures)

The following are the data structures used to represent environments, stores, and addresses:

```
type Environment = Map<Identifier["name"], Address>;
type Store = Map<Address, Value>;
type Address = number;
```

1.5.3 Adding a Store to the Runner

[](#adding-a-store-to-the-runner)

We modify the implementation of `run()` from § \ref{Step 1: The Entire Runner} to introduce a `store`:

```
1 function run(expression: Expression): { value: Value; store: Store } {
2   const store: Store = new Map();
3   return { value: step(expression, new Map()), store };
4   function step(expression: Expression, environment: Environment): Value {
5     // ...
6   }
7 }
```

\begin{description} \item [Lines 1 and 3:]

The `store` is returned because it is necessary to look up variable references in the `value`.

\item [Lines 2 and 4:]

The `store` is unique for the whole interpreter, unlike `environment`s which are different for each closure, so we create only one store that is always available to `step()` instead of adding it as an extra parameter. \end{description}

1.5.4 Adding a Value to the Store [](#adding-a-value-to-the-store)

\begin{tabular}{ll} \textbf{Example Program} & `(x => z => x)(y => y)` \ \textbf{Current Output} & — \ \textbf{Expected Output} & `value` = ⟨ js(z => x) ``, [ `` jsx \mapsto js0 ``] \rangle `` \\ & `` tsstore = math[ `` js0 \mapsto \langle js(y => y), [] \rangle]` \ \end{tabular}

In Step 1, when we encounter a function call we extend the `functionEnvironment` with a mapping from the `parameter.name` to the `argument` (see § \ref{A Function Call}, \ref{A Function Body Is Evaluated with the Environment in Its Closure}). In Step 2, we introduce the `store` as a layer of indirection:

```
1  // step()
2  case "CallExpression": {
3    const {
4      function: {
5        params: [parameter],
6        body,
7      },
8      environment: functionEnvironment,
9    } = step(expression.callee, environment);
10   const argument = step(expression.arguments[0], environment);
11   const address = store.size;
12   store.set(address, argument);
13   return step(
14     body,
15     new Map(functionEnvironment).set(parameter.name, address)
16   );
17 }
```

\begin{description} \item [Line 11:]

Allocate an `address`. We use the `store.size` as the `address` because as the `store` grows this number changes, so it is guaranteed to be unique throughout the interpretation of a program.

\item [Line 15:]

Extend the `functionEnvironment` with a mapping from the `parameter.name` to the `address`.

\item [Line 12:]

Extend the `store` with a mapping from the `address` to the `argument`. This is mutating the unique `store` that is available to the entire `step()` function, not creating a new `store` in the same way that extending an `environment` creates a new `environment` (see line 15). \end{description}

1.5.5 Retrieving a Value from the Store

[](#retrieving-a-value-from-the-store)

\begin{tabular}{ll} \textbf{Example Program} & (y => y)(x => x) \\ \textbf{Current Output} & — \\ \textbf{Expected Output} & value = ⟨ js(y => y) ``, [] \rangle `` \\ & `` tsstore = math[ `` js0 \mapsto \langle js(y => y), [] \rangle]` \\ \end{tabular}

In Step 1 we retrieved values directly from the `environment` (see § \ref{A Variable Reference}), but in Step 2 we have to go through the `store`:

```
1 // step()
2 case "Identifier": {
3   const address = environment.get(expression.name);
4   if (address === undefined)
5     throw new Error(
6       `Reference to undefined variable: ${expression.name}`
7     );
8   return store.get(address)!;
9 }
```

\begin{description} \item [Line 3:]

Retrieve the `address` from the `environment`.

\item [Line 8:]

Retrieve the `value` from the `store` at the given `address` found in the `environment`. The `address` is guaranteed to be in the `store` because we extend the `store` and the `environment` together (see § \ref{Adding a Value to the Store}), so we use `!` to indicate that `get()` may not return `undefined`. \end{description}

1.5.6 The Entire Runner [](#the-entire-runner-2)

We completed the changes necessary to remove the circularity between closures and environments:

```
type Environment = Map<Identifier["name"], Address>;

type Store = Map<Address, Value>;

type Address = number;

function run(expression: Expression): { value: Value; store: Store } {
  const store: Store = new Map();
  return { value: step(expression, new Map()), store };
  function step(expression: Expression, environment: Environment): Value {
    switch (expression.type) {
      case "ArrowFunctionExpression": {
        return { function: expression, environment };
      }
      case "CallExpression": {
        const {
          function: {
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } = step(expression.callee, environment);
        const argument = step(expression.arguments[0], environment);
        const address = store.size;
        store.set(address, argument);
        return step(
          body,
          new Map(functionEnvironment).set(parameter.name, address)
        );
      }
      case "Identifier": {
        const address = environment.get(expression.name);
        if (address === undefined)
```

```
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return store.get(address)!;
      }
    }
  }
}
```

Advanced

1.5.7 Operational Semantics [](#operational-semantics-1)

We adapt the operational semantics from § \ref{Step 2: Operational Semantics} to the interpreter defined in Step 2. First, we change the notion of environments:

\begin{tabular}{rcll} ρ & = & $\{x \mapsto A, \cdots\}$ & Environments \ σ & = & $\{A \mapsto v, \cdots\}$ & Stores \ A & = & $\mathbb{N}$ & Addresses \ \end{tabular}

We then define the relation $\rho, \sigma \vdash e \Rightarrow \langle v, \sigma \rangle$ to be equivalent to the new implementation of `run()`:

\begin{mathpar} \inferrule { } {\rho, \sigma \vdash (x =>e) \Rightarrow \langle \langle (x =>e), \rho \rangle, \sigma \rangle}

\inferrule { \rho, \sigma \vdash e_f \Rightarrow \langle \langle (x =>e_b), \rho_f \rangle, \sigma_f \rangle \ \rho, \sigma_f \vdash e_a \Rightarrow \langle v_a, \sigma_a \rangle \ A = \lvert \sigma_a \rvert \ \rho_f \cup {x \mapsto A}, \sigma_a \cup {A \mapsto v_a} \vdash e_b \Rightarrow \langle v, \sigma_v \rangle \ } {\rho, \sigma \vdash e_f(e_a) \Rightarrow \langle v, \sigma_v \rangle}

\inferrule { } {\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x))}

\end{mathpar}

1.5.8 Programs That Do Not Terminate

[](#programs-that-do-not-terminate-2)

The programs that do not terminate in Step 1 (see § \ref{Step 1: Programs That Do Not Terminate}) do not terminate in Step 2 either, because the interpreters are equivalent except for the treatment of environments, but the sources of non-termination are different. In both cases the issue is that the interpreter may create infinitely many environments, but in Step 1 the environments are nested, and in Step 2 they contain different addresses, for example:

\begin{tabular}{l} \multicolumn{1}{c}{(`f => c => f(f)(x => c))(f => c => f(f)(x => c))(y => y)`} \\ \multicolumn{1}{c}{or `F(F)(y => y)`, where `F = f => c => f(f)(C)` and `C = x => c`} \\ \multicolumn{1}{c}{\textbf{Step 1}} \ ⟨ jsf(f)(C) ``, [ `` jsc \mapsto \langle js(y => y) ``, [] \rangle, \cdots] \rangle `` \\ `` math\langle `f(f)(C)`, [`c` \mapsto \langle `C`, [`c` \mapsto \langle (`y => y`), [] \rangle, \cdots] \rangle, \cdots] \rangle \\ math\langle `` jsf(f)(C) , [jsc `` \mapsto \langle `` jsC , [jsc `` \mapsto \langle `` jsC , [jsc `` \mapsto \langle `` js(y => y) , [] \rangle, \cdots] \rangle, \cdots] \rangle \ ⟨ jsf(f)(C) ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle jsC ``, [ `` jsc \mapsto \langle js(y => y) ``, [] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle, \cdots] \rangle `` \\ \multicolumn{1}{c}{``

math\vdots } \\ \multicolumn{1}{c}{\textbf{Step 2}} \\ \multicolumn{1}{c}{ math\langle `` jsf(f)(C) , [ jsc `` \mapsto `` js0 , \cdots] \rangle}\\multicolumn{1}{c}{⟨ jsf(f)(C) ``, [`` jsc \mapsto js1 ``, \cdots] \rangle ``} \\ \multicolumn{1}{c}{`` math\langle f(f)(C), [c \mapsto 2, \cdots] \rangle } \\ \multicolumn{1}{c}{ math\langle `` jsf(f)(C) , [ jsc `` \mapsto `` js3 , \cdots] \rangle}\\multicolumn{1}{c}{⋮}\\multicolumn{1}{c}{[ js0 `` \mapsto \langle `` js(y => y) , [] \rangle, js1 `` \mapsto \langle `` jsC , [ jsc `` \mapsto `` c0 , \cdots] \rangle, js2 `` \mapsto \langle `` jsC , [ jsc `` \mapsto `` c1 , \cdots] \rangle, \cdots]}\end{tabular}

We address this issue in Step 3.

1.6 Step 3: Finitely Many Addresses

[](#step-3-finitely-many-addresses)

1.6.1 The Entire Runner [](#the-entire-runner-3)

We completed the changes necessary to produce only finitely many addresses:

```
type Value = Set<Closure>;

type Address = Identifier;

function run(expression: Expression): { value: Value; store: Store
} {
  const store: Store = new Map();
  return { value: step(expression, new Map()), store };
  function step(expression: Expression, environment: Environment):
```

```
Value {
    switch (expression.type) {
      case "ArrowFunctionExpression": {
        return new Set([{ function: expression, environment }]);
      }
      case "CallExpression": {
        const value: Value = new Set();
        for (const {
          function: {
            params: [parameter],
            body,
          },
          environment: functionEnvironment,
        } of step(expression.callee, environment)) {
          const argument = step(expression.arguments[0], environment);
          const address = parameter;
          store.merge(new Map([[address, argument]]));
          value.merge(
            step(
              body,
              new Map(functionEnvironment).set(parameter.name, address)
            )
          );
        }
        return value;
      }
      case "Identifier": {
        const address = environment.get(expression.name);
        if (address === undefined)
          throw new Error(
            `Reference to undefined variable: ${expression.name}`
          );
        return store.get(address)!;
      }
    }
  }
}
```

Bibliography

1. Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. 2006.
2. *Babel*. `https://babeljs.io`. Accessed 2020-04-06.
3. *Babel Types*. `https://git.io/JfZF8`. Accessed 2020-05-06.
4. Emery Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. *On the Impact of Programming Languages on Code Quality: A Reproduction Study*. ACM Transactions on Programming Languages and Systems. 2019. `https://doi.org/10.1145/3340571`.
5. Gavin Bierman, Martín Abadi, and Mads Torgersen. *Understanding TypeScript*. European Conference on Object-Oriented Programming (ECOOP). 2014.
6. **Leandro Facchinetti**. *Collections Deep Equal*. `https://github.com/leafac/collections-deep-equal`. Accessed 2020-04-01.
7. Matthias Felleisen. *On the Expressive Power of Programming Languages*. Science of Computer Programming. 1991. `https://doi.org/10.1016/0167-6423(91)90036-W`.
8. Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press. 2009.
9. Matthew Flatt, Robert Bruce Findler, and PLT. *The Racket Guide*. `https://docs.racket-lang.org/guide/`. Accessed 2020-04-13.
10. Mike Grant, Zachary Palmer, and Scott Smith. *Principles of Programming Languages*. 2020.
11. JetBrains. *The State of Developer Ecosystem 2019*. `https://www.jetbrains.com/lp/devecosystem-2019/`. Ac-

cessed 2020-01-14.

12. Gilles Kahn. *Natural Semantics*. Annual Symposium on Theoretical Aspects of Computer Science. 1987.
13. Peter Landin. *The Mechanical Evaluation of Expressions*. The Computer Journal. 1964.
14. Bil Lewis, Dan LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual*. 2015.
15. John McCarthy. *History of LISP*. History of Programming Languages. 1978. `https://doi.org/10.1145/800025.1198360`.
16. John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. Communications of the ACM. 1960. `https://doi.org/10.1145/367177.367199`.
17. Matthew Might. *The Language of Languages*. `http://matt.might.net/articles/grammars-bnf-ebnf/`. Accessed 2020-01-17.
18. Matthew Might, Yannis Smaragdakis, and David Van Horn. *Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs Object-Oriented Program Analysis*. Programming Language Design and Implementation (PLDI). 2010. `https://doi.org/10.1145/1806596.1806631`.
19. Mozilla. *Arrow Function Expressions*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions`. Accessed 2020-01-16.
20. Mozilla. *Destructuring Assignment*. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/`

Destructuring_assignment. Accessed 2020-01-27.

21. Mozilla. eval(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval. Accessed 2020-02-13.

22. Mozilla. JSON.stringify(). https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify. Accessed 2020-04-13.

23. Mozilla. Map. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map. Accessed 2020-06-25.

24. Mozilla. *Spread Syntax*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax. Accessed 2020-02-03.

25. Mozilla. *Template Literals*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals. Accessed 2020-02-03.

26. Gordon Plotkin. *Call-By-Name, Call-By-Value and the λ-Calculus*. Theoretical Computer Science. 1975. https://doi.org/10.1016/0304-3975(75)90017-1.

27. *Prettier*. https://prettier.io. Accessed 2020-02-18.

28. John Reynolds. *Definitional Interpreters for Higher-Order Programming Languages*. Proceedings of the ACM Annual Conference. 1972. https://doi.org/10.1145/800194.805852.

29. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD Dissertation, Carnegie Mellon University. 1991.

30. Stack Overflow. *Developer Survey Results 2019*.

https://insights.stackoverflow.com/survey/2019. Accessed 2020-01-14.

31. Tom Stuart. *Understanding Computation: From Simple Machines to Impossible Programs*. O'Reilly Media. 2013.

32. Basarat Ali Syed. *TypeScript Deep Dive*. https://basarat.gitbook.io/typescript/. Accessed 2020-01-17.

33. *TypeScript Documentation*. https://www.typescriptlang.org/docs. Accessed 2020-01-17.