DRAFT: 2020-02-18 10:44:50-05:00
YOCTO-CFA

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

Abstract

Primary Reader and Advisor: Dr. Scott Fraser Smith.

Readers: Dr. Zachary Eli Palmer and Dr. Matthew Daniel Green.

Acknowledgements

Contents

List of Tables

List of Figures

Chapter 1

Developing an Analyzer

1.1 The Analyzed Language: Yocto-JavaScript

Our first decision when developing an analyzer is which language it should analyze. In this dissertation we are interested in analysis techniques for higher-order functions, a feature which is supported by most languages, including JavaScript, Java, Python, Ruby, and so forth.

From all these options, we would like to choose JavaScript because it is the most popular language among programmers [22, 14], but JavaScript has many features besides higher-order functions that would complicate our analyzer, so we support only a *subset* of JavaScript features that are related to higher-order functions, resulting in a language that we call *Yocto-JavaScript* (JavaScript $\times 10^{-24}$). By design, 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 determines how difficult the analyzer is to develop, but the analyzed language may also influence the analyzer's precision and running time. For example, there is

an analysis technique called *k*-CFA [25] 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 [17].

Technical Terms

Yocto-JavaScript is a representation of something called the *λ-calculus* [26, § 6].

1.1.1 Values in Yocto-JavaScript

JavaScript has many kinds of values: strings (for example, `"Leandro"`), numbers (for example, `29`), arrays (for example, `["Leandro", 29]`), objects (for example, `{ name: "Leandro", age: 29 }`), and so forth. From all these kinds of values, Yocto-JavaScript supports only one: functions.

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). 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).

Technical Terms

The notation we use for writing functions is something called *arrow function expressions* [18]. The function given as example is called the *identity* function. The ability of acting as values is what characterizes these functions as *higher-order*.

1.1.2 Operations in Yocto-JavaScript

JavaScript has many operations: strings may have its characters accessed (for example, `"Leandro"[2]`, which results in `"a"`), numbers may be added together (for example, `29 + 1`, which results in `30`), and so forth. From all these operations, Yocto-JavaScript supports only two: functions may be called and variables may be referenced.

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:

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

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

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`, and in `f(g(h))` the call `g(h)` happens first and the result is passed as argument to `f`.

Technical Terms

The order in which operations are computed is something called their *precedence*, and operations that happen first are said to have *higher precedence*.

Advanced

1.1.3 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 is it *too* simple? In other words, in the process of pairing down JavaScript to define Yocto-JavaScript, have we removed features that make the language incapable of some computations? Perhaps surprisingly, the answer is negative: Yocto-JavaScript is equivalent to JavaScript (and Java, Python, Ruby, and so forth) in the sense that, with some effort, any program in any one language may be translated into an equivalent program in any other language [26, § 6].

As an example of how to carry out this translation, consider a JavaScript function of two parameters: `(x, y) => x`. This function is not supported by Yocto-JavaScript, which supports only functions of one parameter (see § 1.1.1), but we may encode it as a function that receives the first parameter and returns another function that receives the second parameter: `x => (y => x)`. 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

All the languages we are considering are said to be equivalent in terms of *computational power*: they are all *Turing complete* [26, § 7]. The translation technique for functions with multiple arguments is called *currying* [26, page 163].

For our goal of exploring analysis techniques, we are concerned only with computational power, but it is worth noting that programmers may be more interested in other language properties: Does the language promote writing programs of higher quality? (It most probably does not [9].) Does the language improve productivity? Does the language work well 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 without resorting to personal preference and anecdotal evidence [11].) 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

The description of Yocto-JavaScript given so far has been informal; the following is a grammar in *Backus–Naur Form* (BNF) [16] [8, § 4.2] that formalizes it:

e	::=	x => e | $e(e)$ | x	Expressions
x	::=	<*A Valid JavaScript Identifier*>	Variables

1.2 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, but that does not restrict our choice of language in which to develop the analyzer because the analyzer treats the analyzed program as data, so it could be developed in any language: JavaScript, Java, Python, Ruby, and so forth. Still, from all these options, JavaScript does offer some advantages: it is the most popular [22, 14], and it includes convenient tools to manipulate JavaScript programs (and therefore Yocto-JavaScript programs as well; see § 1.3.16 and § 1.3.17). But JavaScript lacks a way to express the *types* of data structures, functions, and so forth, which we will need (for example, see § 1.3.2), so we choose to implement our analyzer in a JavaScript extension with support for types called *TypeScript* [6, 27, 10].

1.3 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 the series of Steps in the development of the analyzer. The first Step is an interpreter that 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

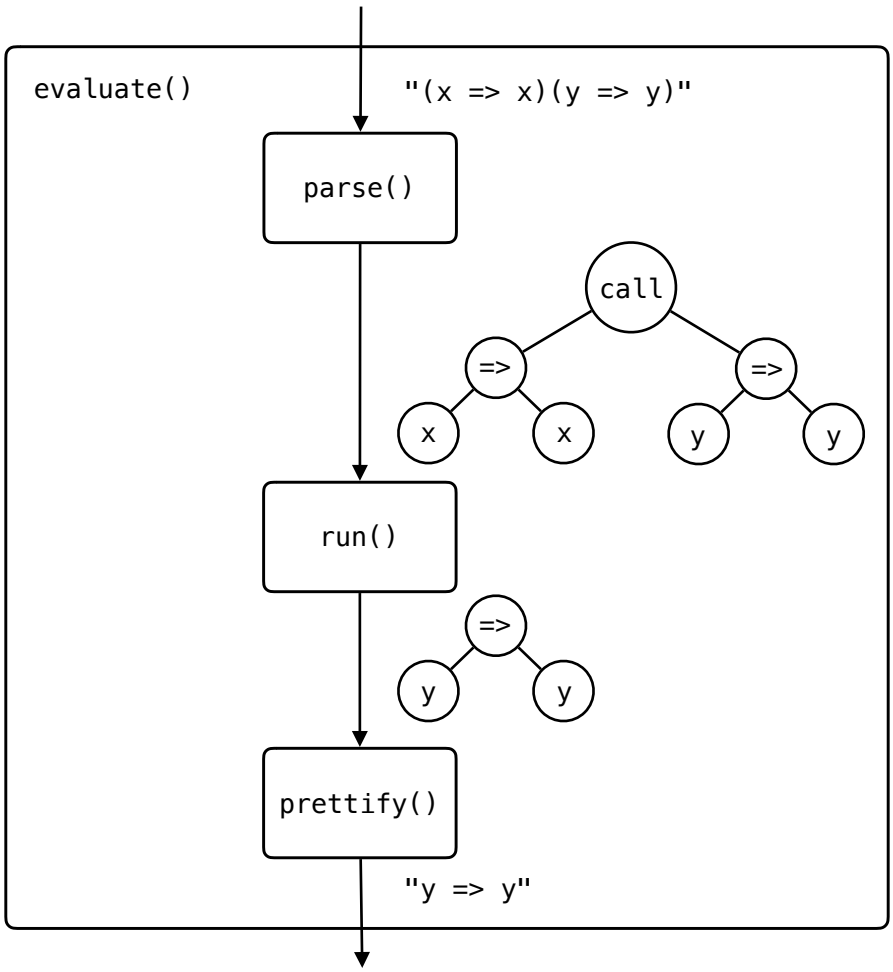
Our interpreter is defined as a function called `evaluate()`, which receives as parameter an Yocto-JavaScript program represented as a string and returns the result of running it.

> **Advanced**
>
> The `evaluate()` function is named after a native JavaScript function called `eval()`, which is similar to `evaluate()` but for JavaScript programs [20].

The following are two examples of how we will be able to use `evaluate()` by the end of this Step (the `>` represents the console):

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

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




```
export function evaluate(input: string): string {
  return prettify(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). The `run()` function is responsible for the interpretation itself. The `prettify()` function converts the outputs of `run()` into a human-readable format. In the following sections (§ 1.3.2–§ 1.3.15) we address the implementation of `run()`, deferring `parse()` to § 1.3.16 and `prettify()` to § 1.3.17.

1.3.2 Data Structures to Represent Yocto-JavaScript Programs

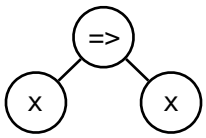
The `evaluate()` function receives an Yocto-JavaScript program represented as a string (see § 1.3.1), which is convenient for humans to write and read, but inconvenient for `run()` to manipulate directly, because `run()` is concerned with the *structure* of the program instead of the *text*: from `run()`'s perspective it does not matter, for example, whether a function is written as `x => x` or as `x=>x` (note the difference in spacing). So before `run()` starts interpreting the program, `parse()` transforms it from a string into more convenient data structures (see § 1.3.16 for `parse()`'s implementation).

Technical Terms

The process of converting a program represented as a string into more convenient data structures is known as *parsing*, and the data structures are called the *Abstract Syntax Tree* (AST) of the program [8, § 4].

The following are two examples of Yocto-JavaScript programs followed by the data structures used to represent them, first in a high-level graphical representation and then in an equivalent low-level textual representation:

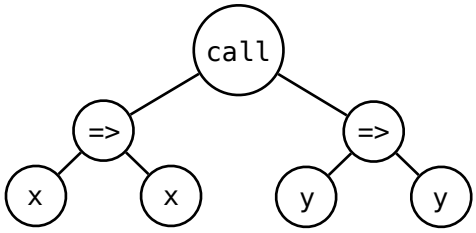
```
> parse("x => x")
```




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

```
  }
}
```

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




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

We choose to represent Yocto-JavaScript programs with the data structures above because they follow a specification called ESTree [4], and by adhering to this specification we may reuse tools from the JavaScript ecosystem (see § 1.3.16 and § 1.3.17).

In general, the data structures used to represent Yocto-JavaScript programs are of the following types (written as TypeScript types adapted from the ESTree types [7] to include only the features supported by Yocto-JavaScript):

```
type Expression = ArrowFunctionExpression | CallExpression | Identifier;

interface ArrowFunctionExpression {
  type: "ArrowFunctionExpression";
  params: [Identifier];
  body: Expression;
}

interface CallExpression {
  type: "CallExpression";
  callee: Expression;
  arguments: [Expression];
}

interface Identifier {
  type: "Identifier";
  name: string;
}
```

Advanced

The definitions above correspond to elements of the Yocto-JavaScript grammar (see § 1.1.4); for example, `Expression` corresponds to e.

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

1.3.3 An Expression That Already Is a Value

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

We start the definition of `run()` by considering the example 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 be 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 it may be returned unchanged:

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

1.3.4 A Call Involving Immediate Functions

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

Interpreting function calls is the main responsibility of our interpreter. There are several techniques to do this and in this Step we use one of the simplest: when the interpreter encounters a function call, it substitutes the variable references in the body of the function that is called with the argument that is passed. This is similar to how we reason about functions in mathematics; for example, given the function $f(x) = x + 1$, we calculate $f(29)$ by substituting the references to x in f 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.

Technical Terms

This technique for interpreting function calls is what characterizes our interpreter in this Step as a *substitution-based interpreter*.

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

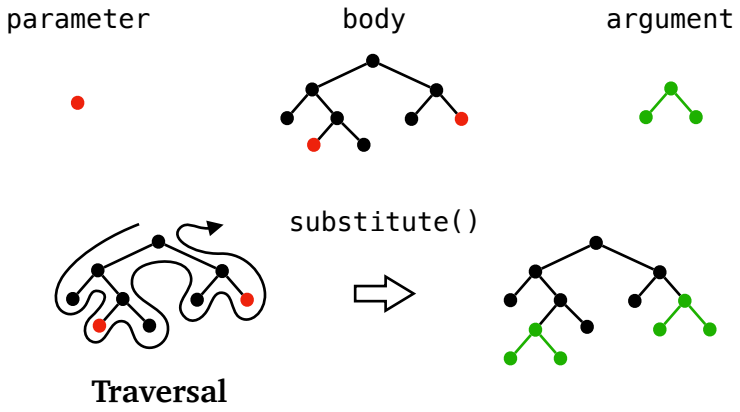
```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  throw new Error("NOT IMPLEMENTED YET");
```

Next, we unpack the called function (using something called *destructuring assignment* [19]) and the argument:

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

```
    params: [{ name: 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:



Traversal

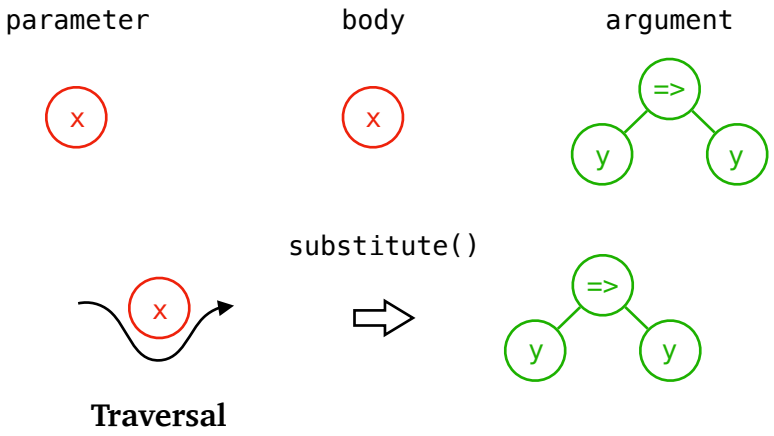
```
// run()
case "CallExpression":
  if (
    expression.callee.type !== "ArrowFunctionExpression" ||
    expression.arguments[0].type !== "ArrowFunctionExpression"
  )
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [{ name: parameter }],
    body
  } = expression.callee;
  const 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 `x`, an `Identifier`, and it must be substituted with the `argument`:

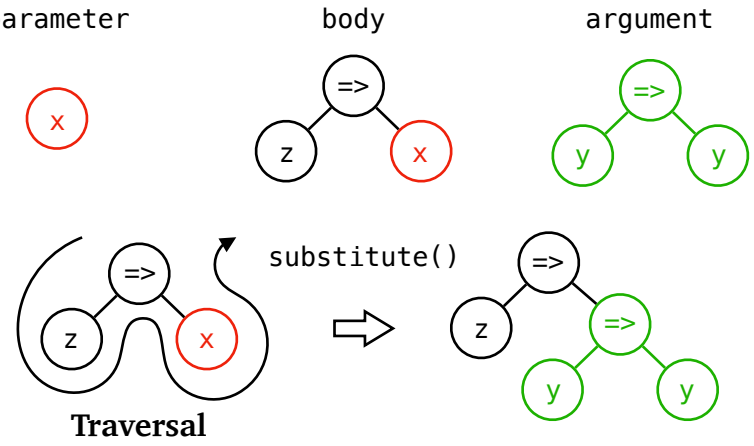



```
// substitute()
case "Identifier":
  return argument;
```

1.3.5 Substitution in Function Definitions

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

When `substitute()` (see § 1.3.4) starts traversing the `body` of the example above, the `expression` is an `ArrowFunctionExpression` (`z => x`), and we want substitution to proceed deeper to find and substitute `x`, so we call `substitute()` recursively (we use a feature called *spread syntax* [21] 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

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

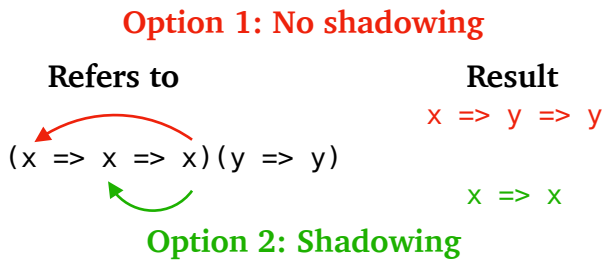
The implementation of `substitute()` in the case of `Identifier` introduced in § 1.3.4 *always* substitutes variable references, regardless of whether they refer to the `parameter` of the called function. For example, in the program above `substitute()` is substituting the `z` even though the `parameter` is `x`. 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) return expression;
  return argument;
```

1.3.7 Name Reuse

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

In the program above, there are two options for the variable reference `x` on the right of the second `=>`: it may refer to the first (outer) `x` on the left of the first `=>`, in which case the output of the program would be `x => y => y`; or it may refer to the second (inner) `x` on the left of the second `=>`, in which case the output of the program would be `x => x`:



Currently `substitute()` is implementing Option 1, but this leads to a serious issue: we are not able to reason about the inner function `x => x` independently; we must know where it appears and whether a variable called `x` is already defined there.

Technical Terms

We say that the problem with Option 1 is that it defeats something called *local reasoning*. We say that Option 2 exhibits a behavior called *shadowing*, and that the outer `x` is *shadowed* by the inner `x`, because there is no way to refer to the outer `x` from the body of the inner function.

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 (note that there is no recursive call to `substitute()` in this case):

```
// substitute()
case "ArrowFunctionExpression":
```

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

1.3.8 Substitution in Function Calls

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

This case is similar to § 1.3.5: all `substitute()` has to do is continue traversing the function call recursively:

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

1.3.9 An Argument That Is Not Immediate

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

In all example programs we considered so far the argument to a function call was an immediate function definition, but in general arguments may be the result of function calls themselves. We fix this by calling `run()` recursively on the argument (we also remove the check that the argument is an immediate function definition; if it is, then the recursive call to `run()` returns the immediate function unchanged; see § 1.3.3):

```
// run()
case "CallExpression":
  if (expression.callee.type !== "ArrowFunctionExpression")
    throw new Error("NOT IMPLEMENTED YET");
  const {
    params: [{ name: 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

This technique of calling `run()` recursively to produce an immediate function for the argument characterizes the interpreter as *big-step*.

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 [23]. JavaScript itself and most other popular programming languages are call-by-value as well, the most notable exception being Haskell, which is a *call-by-need* language. In a call-by-need language the argument is only interpreted 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 => ((a => a)(y => y))`. There is yet another policy for when to interpret arguments called *call-by-name*: the difference between call-by-name and call-by-need is that in a call-by-name language the an argument may be computed multiple times if it is used multiple

times, but in a call-by-need language an argument is guaranteed to be computed at most once.

1.3.10 A Function That Is Not Immediate

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

This is the dual of § 1.3.9 for the called function, and the solution is the same: to call `run()` recursively (we also remove the check of whether the function is immediate):

```
// run()
case "CallExpression":
  const {
    params: [{ name: 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

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

This is similar to § 1.3.9 and § 1.3.10: the result of substitution may be not an immediate function but another call, and more work may be necessary to interpret it. We solve this with yet another recursive call to `evaluate()` (we also remove yet another check and inline the `substitutedBody` variable):

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

```
  const {
    params: [{ name: 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

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

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}`);
```

Note that 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. For example, the result of running the program `x => y` is `x => y`, not an exception. This is consistent with JavaScript's behavior.

1.3.13 The Entire Runner

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

```
1  type Value = ArrowFunctionExpression;
2
3  function run(expression: Expression): Value {
4    switch (expression.type) {
5      case "ArrowFunctionExpression":
```

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

1.3.14 A Program That Does Not Terminate

Example Program	Current Output	Desired Output
`(f => f(f))(f => f(f))`	`DOES NOT TERMINATE`	`DOES NOT TERMINATE`

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: on the first call to `run()` both the called function and the argument are immediate functions (they are both `f => f(f)`), so the first two recursive calls to `run()` (see § 1.3.13, lines 11 and 12) return the functions unchanged. At this point, the

`parameter` is `f`, the `body` is `f(f)`, and the `argument` is `f => f(f)`. The result of the call to `substitute()` (see § 1.3.13, line 13) is `(f => f(f))(f => f(f))`, which is the same as the initial program, so the recursive call to `run()` (see § 1.3.13, line 13) leads to an infinite loop.

This is what we expect from an interpreter, but not from an analyzer, which must always terminate. In the next Steps we will transform this interpreter to turn it into an analyzer, and one of the main issues we will address is termination.

Advanced

1.3.15 An Operational Semantics for the Interpreter

What we accomplished so far in this section 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 one we are using so far is known as a *definitional interpreter* [24].

A definitional interpreter has some advantages over other techniques for specifying semantics: it is easier to understand for most programmers, and it is executable. 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. To address this, other techniques for defining semantics do not depend on other programming languages, and in this section we introduce one of them: *operational semantics* [15, 12, 13].

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 provide *inference rules* that are equivalent to the behavior of `run()` (see § 1.3.13, lines 3–36):

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

Finally, we provide a *metafunction* that is equivalent to the behavior of `substitute()` (see § 1.3.13, lines 14–32):

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

1.3.16 Parser

The parser is responsible for converting an Yocto-JavaScript program written as a string into data structures that are more convenient for the runner to manipulate (see § 1.3.1 and § 1.3.2 for the reasoning and the definition of the data structures). We choose to represent Yocto-JavaScript programs with data structures that are compatible with a specification for representing JavaScript programs called ESTree [4, 7] because it allows us to reuse tools from the JavaScript ecosystem, including a parser called Esprima [2], and the Esprima Interactive Online Demonstration [3], which shows the data structures used to represent a given program.

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

```
1 function parse(input: string): Expression {
```

```
2     const program = parseScript(input, {}, checkFeatures);
3     const expression = (program as any).body[0].expression as Expression;
4     return expression;
5     function checkFeatures(node: Node): void {
6       switch (node.type) {
7         case "Program":
8           if (node.body.length !== 1)
9             throw new Error(
10              "Unsupported Yocto-JavaScript feature: Program with multiple statements"
11            );
12          break;
13        case "ExpressionStatement":
14          break;
15        case "ArrowFunctionExpression":
16          break;
17        case "CallExpression":
18          if (node.arguments.length !== 1)
19            throw new Error(
20              "Unsupported Yocto-JavaScript feature: CallExpression with multiple argume
21            );
22          break;
23        case "Identifier":
24          break;
25        default:
26          throw new Error(`Unsupported Yocto-JavaScript feature: ${node.type}`);
27      }
28    }
29  }
```

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

Line 2: Call the Esprima function `parseScript()`, which parses the `input` as if it were a JavaScript program and produces a data structure following the ESTree specification. The `parseScript()` function also detects syntax errors, for example, in the program `x =>`, which is missing the function body. We pass as argument to `parseScript()` a function called `checkFeatures()` which is called with every fragment of data structure that represents a part of the program. The purpose of `checkFeatures()` is to check that the program uses only the features that are supported by Yocto-JavaScript.

Line 3: Extract the single `Expression` from within the `Program` returned by `parseScript()`. The `as <something>` forms sidestep the TypeScript type checker and assert that the `expression` is of the correct type. This is safe to do because of `checkFeatures()`.

Line 5: The `checkFeatures()` function, which is passed to Esprima's `parseScript()` is called with every fragment of data structure used to represent the program. These fragments are called *nodes*, because the data structure as a whole forms a *tree*, also known as the *Abstract Syntax Tree* (AST) of the program (see § 1.3.2). The `checkFeatures()` does not return anything (`void`); its purpose is only to throw an exception in case the program uses a feature that is not supported by Yocto-JavaScript.

Lines 6, 7, 13, 15, 17, 23, 25: Similar to `run()` and `substitute()` (see § 1.3.13), `checkFeatures()` starts by determining which type of `Node` it is given.

Lines 8–11: Check that the `Program` contains a single statement. This prevents programs such as `x => x; y => y`.

Lines 13, 15: `ExpressionStatements` and `ArrowFunctionExpressions` are supported in Yocto-JavaScript unconditionally. We could check that the `ArrowFunctionExpress` includes only one parameter and that this parameter is a variable (as opposed to being a pattern such as `[x, y]`, for example), but this would be redundant because Esprima already calls `checkFeatures()` with other unsupported `nodes` that subsume these cases. For example, in the program `(x, y) => x`, which is a function of multiple parameters, Esprima calls `checkFeatures()` with a `node` of type `SequenceExpression`. Similarly, in the program `([x, y]) => x`, which is a function in which the parameter is a pattern, Esprima calls `checkFeatures()` with a `node` of type `ArrayExpression`.

Lines 18–21: Check that the `CallExpression` contains a single argument. This

prevents programs such as `f(a, b)`.

Line 23: `Identifiers` are supported in Yocto-JavaScript unconditionally. Note that an `Identifier` may be an expression or the parameter of an `ArrowFunctionExpression`.

Line 26: All other types of `Node` are not supported by Yocto-JavaScript. This includes programs such as `29` (`Literal`) and `const f = x => x` (`VariableDeclarator`).

1.3.17 Prettifier

The prettifier transforms a `Value` produced by `run()` into a human-readable format (see § 1.3.1). Similar to the parser (see § 1.3.16), we may implement the prettifier by reusing existing tools from the JavaScript ecosystem, because we are representing Yocto-JavaScript programs and values with data structures that follow the ESTree specification. In particular, we use a library called Escodegen [1] to produce a string representation of an ESTree data structure, and a library called Prettier [5] to format that string. The following is the full implementation of the prettifier:

```
function prettify(value: Value): string {
  return format(generate(value), { parser: "babel", semi: false }).trim();
}
```

Bibliography

[1] Escodegen. https://github.com/estools/escodegen. Accessed 2020-02-18.

[2] Esprima. https://esprima.org. Accessed 2020-01-21.

[3] Esprima interactive online demonstration. https://esprima.org/demo/parse.html. Accessed 2020-01-21.

[4] The estree spec. https://github.com/estree/estree. Accessed 2020-01-21.

[5] Prettier. https://prettier.io. Accessed 2020-02-18.

[6] Typescript homepage. https://www.typescriptlang.org. Accessed 2020-01-17.

[7] Typescript types for the estree spec. https://github.com/DefinitelyTyped/DefinitelyTyped/blob/1911a1fbbe30af03f0f38a915c4bf0620d251fc6/types/estree/index.d.ts. Accessed 2020-01-27.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[9] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the impact of programming languages on code quality: A reproduction study. *ACM Trans. Program. Lang. Syst.*, 41(4), October 2019.

[10] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[11] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35 – 75, 1991.

[12] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[13] Mike Grant, Zachary Palmer, and Scott Smith. *Principles of Programming Languages*. 2020.

[14] JetBrains. The state of developer ecosystem 2019. `https://www.jetbrains.com/lp/devecosystem-2019/`. Accessed 2020-01-14.

[15] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.

[16] Matthew Might. The language of languages. `http://matt.might.net/articles/grammars-bnf-ebnf/`. Accessed 2020-01-17.

[17] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 305–315, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Mozilla. Arrow function expressions. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions. Accessed 2020-01-16.

[19] Mozilla. Destructuring assignment. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment. Accessed 2020-01-27.

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

[21] Mozilla. Spread syntax. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax. Accessed 2020-02-03.

[22] Stack Overflow. Developer survey results 2019. https://insights.stackoverflow.com/survey/2019. Accessed 2020-01-14.

[23] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125 – 159, 1975.

[24] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, page 717–740, New York, NY, USA, 1972. Association for Computing Machinery.

[25] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, PhD thesis, Carnegie Mellon University, 1991.

[26] Tom Stuart. *Understanding Computation: From Simple Machines to Impossible Programs*. O'Reilly Media, 1 edition, 2013.

[27] Basarat Ali Syed. Typescript deep dive. https://basarat.gitbook.io/typescript/. Accessed 2020-01-17.

Biographical Statement