HandCrank:
Hybrid Partial Evaluation for JavaScript

Master Thesis Computer Science

Marco Gunnink
University of Groningen

Supervisors:
prof. dr. T. van der Storm
prof. dr. G.R. Renardel de Lavalette

7th April 2021

Abstract

JavaScript is the language for creating dynamic web content, which is becoming increasingly more complex. At the same time, users of websites expect them to respond fast, requiring good runtime performance. One method for improving the performance of computer programs is partial evaluation. Partial Evaluation is of particular interest for Domain-Specific Languages (DSLs), where a partial evaluator can improve the performance of programs written in a DSL by eliminating the overhead of the DSL interpreter. However, this technique is not easy to implement for such a complicated language as JavaScript, and may still lead to problems like code explosion. An alternative approach, called Hybrid Partial Evaluation (HPE) [16] offers to alleviate these problems. This thesis describes a semantics of HPE for JavaScript and HandCrank: an implementation based on it. In addition to the existing semantics of HPE, we offer a novel implementation of non-local control flow and tracking of object modification in HPE. The implementation is tested on a number of benchmark programs to see if their specialized versions perform better than the original ones. The results of these benchmarks show that not all partially evaluated programs are faster, but in certain cases, Hybrid Partial Evaluation can result in increased perfomance.

The only difference between science and screwing around, is writing it down.

Alexander Jason,
via Adam Savage

Acknowledgements

First and foremost, I would like to thank my supervisor, Tijs van der Storm, for his patience and feedback. It took a while, but we got it done.

Secondly, I thank my mom & dad, brother, and sisters for their emotional support, and polite nodding when I ramble on about my research.

Finally, I want to thank Matthia Sabatelli and Ynte Tijsma for their feedback and occasional doses of (in)sanity.

Contents

1 Introduction

JavaScript is one of the most popular programming languages today. Not only is it effectively the only language for making web pages dynamic, it is now also used on the web servers that serve those pages, and in database systems that provide their content. With the ever increasing popularity of the web, and interactive web pages, JavaScript programs have gotten larger and more complicated. At the same time, visitors of these web pages expect them to be fast and responsive. Developers of web pages and web browsers are therefore constantly working towards faster execution of JavaScript code.

One technique for optimizing computer programs is partial evaluation, also called partial computation [6]. With partial evaluation a program is specialized by evaluating parts of the program that depend on static data. This is of particular interest for the implementation of DSLs [7]. Programs written in a DSL are usually executed by an interpreter written in a general-purpose language. While that makes writing those programs easier, their performance suffers because the interpreter of the DSL causes an execution overhead at runtime. This problem can be solved with the help of partial evaluation. A partial evaluator can combine a program written in a DSL with its interpreter, and generate the same program in the general purpose language that the interpreter is written in. It creates a specialized version of the interpreter for just that program.

Partial evaluators generally fall into one of two categories: online and offline. Offline partial evaluation uses binding time analysis [8] to determine which parts can be evaluated at compile-time to generate optimized code which is executed later, whereas online partial evaluation gathers the required information during evaluation.

Partial evaluation is not without its problems, one of which is that it can lead to code explosion, which is undesirable when the code needs to be transported over the internet to end users when a website is visted. HPE [16] is a technique for online partial evaluation combined with ideas from offline partial evaluation. This technique forgoes the binding time analysis in order to simplify the partial evaluator, and instead relies on the programmer to indicate which parts of the program are to be partially evaluated. With this control over the partial evaluation process, the programmer can prevent code explosion.

1.1 Research questions

In this thesis we apply the ideas of Hybrid Partial Evaluation to the JavaScript programming language by building a hybrid partial evaluator for JavaScript in JavaScript. The two main research questions we aim to answer in this thesis are:

- How can Hybrid Partial Evaluation be implemented for JavaScript?
- Does Hybrid Partial Evaluation of JavaScript result in measurable performance gains?

We will first give a detailed overview of the JavaScript language in section 2, followed by partial evaluation and in particular Hybrid Partial Evaluation in section 3.

To answer the first research question, we built a JavaScript interpreter in JavaScript, which is explained in section 4, and on top of it a hybrid partial evaluator, described in section 5. The second half of the research is described in sections 6 and 7, where we test the validity of our software and measure the performance impact of hybrid partial evaluated code. Finally, in sections 8 and 9 we discuss our work and findings, provide perspectives on future work, and answer the research questions.

2 JavaScript

JavaScript is a high-level, imperative, object-oriented programming language. It is used primarily in combination with HTML and CSS to provide interactivity on web pages. This section will give an overview of the JavaScript language, and will focus in particular on the features and behaviours of JavaScript that set it apart from other languages.

While most commonly known and referred to as 'JavaScript', the language described herein is technically known as 'ECMAScript'. Named after the organization that manages the language standard, Ecma International, because the name JavaScript is a trademark of the Oracle Corporation and the members of the standards committee could not agree on another. Nevertheless, we shall use the term JavaScript for most of this document and use ECMAScript when referring to the standard or a specific version of it.

To keep the description relatively simple, we will only discuss the strict mode variant of JavaScript, introduced in ECMAScript 5, which is recommended for use in new code. Strict mode has slightly different syntax and semantics from non-strict 'sloppy mode' to accomodate potential future changes to the language, it converts silent mistakes into errors, and removes some features that inhibit optimizations. Furthermore, we will not discuss the APIs involved in manipulating web pages, which are not part of the ECMAScript specification itself but managed by various different standards under the purview of the World Wide Web Consortium (W3C). Finally, as the language continually evolves, this document cannot reflect the latest version of the standard, instead it is based on the 9th edition: ECMAScript 2018.

2.1 History

JavaScript was developed in 1995 by Brendan Eich for the Netscape Navigator web browser [14]. Despite the name and syntax similarities, JavaScript is not a subset or dialect of the Java programming language. Its development, however, was prompted by Java being positioned as the language to add interactivity to the web. While Netscape Communications had initially partnered up with Sun's JavaSoft to integrate the Java language with their browser, they simultaneously tasked Eich with developing a language that could be run by the browser directly. Eich's original design was a Scheme-like language, named LiveScript, but the name and syntax were soon changed to appeal more to programmers familiar with the C family of languages. The first version of JavaScript was finally shipped with Netscape Navigator 2.0 in december of 1995.

In 1996, Microsoft released version 3.0 of their own browser, Internet Explorer, with JScript, reverse-engineered from the Netscape implementation. While their version was mostly compatible with that of Netscape, the differences between the two were large enough to require significant workarounds, especially when interacting with elements on a web page. This lead to programmers favouring one implementation over the other and labeling their site as "Best viewed with Internet Explorer" or "Best viewed with Netscape". To overcome these issues, Netscape Communications submitted JavaScript for standardization to Ecma International, who published the first version of the ECMA-262, the ECMAScript

standard in June 1997 [13].

The second edition of the standard, published in 1998, only made some small changes to keep the text in line with the corresponding ISO/IEC standard. But the third edition added major features and fixes, in 1999. Although work on it started right after the release of version 3, the fourth edition was never finished. By 2003 progress toward the new language version had halted almost entirely. The standards committee could not agree on which direction to take the language in: some members wanted to add a large number of new features to support *programming in the large*, while others were afraid those features would be incompatible with existing code and would thus "break the web". Eventually, in 2007, it was decided to have two subcommittees working in parallel, with one subcommittee working on the new features as ECMAScript 4, while the other worked on version 3.1. The latter a less ambitious update to the existing standard focused on security, stability, and compatibility between implementations. This version progressed steadily, but ECMAScript 4 turned out too ambitious and once again progress on it stalled. Finally, in 2008, the committee decided that version 4 would be withdrawn and the next standard of the language would be based on version 3.1. It was released in December 2009 as ECMAScript 5.

The efforts on ECMAScript 4 were not entirely wasted as some of its features were incorporated into ECMAScript 6, under the name *Harmony*. This version was released in 2015 as ECMAScript 2015. From this version on, the standard is updated yearly with smaller changes that remain backwards compatible. This backwards compatibility is an important goal for the ECMAScript standards committee as they want to ensure that existing websites will keep functioning as intended even when they are no longer updated. At the same time, supporting old code also means retaining features that are no longer desirable, or behaviours that were originally unintended quirks of the engine, but which old code came to rely upon.

While the Netscape browser is now obsolete, its legacy lives on in Firefox, which uses the SpiderMonkey JavaScript engine. Other notable webbrowsers include Chromium and Google Chrome, which use V8, Microsoft Edge, which formerly used Chakra and now also V8, Safari, using JavaScriptCore, and Opera, which used various proprietary JavaScript engines before also switching to V8.

Although JavaScript was originally designed for use as client-side code in web pages, it has also become popular on the server via Node.js. This is a JavaScript platform based on the V8 engine that also powers Chromium. In addition to the core ECMAScript library, Node.js also provides built-in APIs for manipulating files, creating arbitrary network connections, and running other programs. Beyond those, Node.js comes bundled with the Node Package Manager (NPM), a tool and ecosystem for using, creating and distributing JavaScript packages. These packages range from small single-function utilities to grand frameworks, and the official NPMjs.com repository has almost 1.5 million publicly available packages.

2.2 Basic Syntax

The syntax of JavaScript is very similar to that of Java and C. Scripts are composed of one or more statements, which are executed in order. Statements

Listing 1: JavaScript example

```
// This is a line comment
/* This is a
   block comment */

// Execute the script in strict mode
"use strict";

// Declare function power that takes 2 parameters named 'x', and 'n'
function power(x, n){
    // Declare local variable 'ret' initialized to 1
    let ret = 1;
    // Loop while n is greater than 0
    while(n > 0){
        // Assign ret times x to ret
        ret = ret * x;
        // Decrement n by 1
        n--;
    }
    // Return the value of ret
    return ret;
}

// Declare global variable 'x' initialized to a random value using the
// built-in method random of the global Math object
var x = Math.random();
// Call power with arguments x and 2
var p = power(x, 2);
// Print the result to the standard output, prefixed by a string
console.log("x*x =", p);
```

are terminated with a semicolon, except for block statements, but the programmer may omit them. Where expected semicolons are absent, the interpreter will insert them automatically, however, this can have surprising results. Listing 2 shows a very brief overview of the JavaScript syntax, and an example program can be found in listing 1. Note that the real syntax is far more complicated, however the elements shown are most relevant to the further research in this thesis.

2.3 Data Types

JavaScript has 7 primitive data types:

1. Boolean: the values **`true`** and **`false`**.
2. Number: a value with the range and behaviour of the IEEE-754 double precision floating point format [9].
3. String: an immutable sequence of UTF-16 code points.
4. `undefined`: which indicates a missing value.
5. **`null`**: which indicates an empty value.
6. Object: a set of key/value pairs, used as the building block for more complex data.
7. Symbol: a unique and opaque value, used mainly as object property keys to avoid namespace conflicts.

Additionally, there are two more data types that are a special kind of object:

1. Array, an object with primarily numeric property keys, a `length` property, and methods for manipulating sequences of values.
2. Function, an object that can be invoked by a call expression to execute code associated with the object.

As a dynamically typed language, variables, parameters, and functions do not need to be declared with a type, and differently typed values may be assigned to the same identifier. Similarly, objects and arrays can store values of different types within the same container.

Some operators and built-in functions will convert incompatible operands. For example, the equality operator `==` will try to convert operands of different types before comparing them. It considers `"1" == 1` to be true. This is also known as type coercion. The 'strict' equality operator `===`, on the other hand, does not do this, and considers operands of different types unequal.

The plus operator `+` is used for both addition and string concatenation. If either of its operands is a string, it will convert the other to a string as well and return their concatenation, otherwise their sum. The other arithmetic operators, `-`, `*`, `/`, `%` (remainder), `**` (exponentiation), will attempt to convert both their operands to numbers, returning `NaN` (Not a Number) when that is impossible.

The **`typeof`** operator can be used to retrieve the type of a variable or expression. It returns one strings `"boolean"`, `"number"`, `"string"`, `"undefined"`, or `"symbol"` for values with that type. For objects, arrays, and **`null`**, it returns `"object"`, which

Listing 2: JavaScript syntax

```
Script ::= ( Stmt | Decl )*
Stmt ::= Expr ';'
    | 'if' '(' Expr ')' Stmt
    | 'while' '(' Expr ')' Stmt
    | 'for' '(' (Expr | Decl)? ';' Expr? ';' Expr? ')' Stmt
    | 'break' label? ';'
    | 'continue' label? ';'
    | 'return' Expr? ';'
    | '{' ( Stmt | Decl )* '}' /* block statement */
Expr ::= Literal
    | id  /* read variable */
    | Expr '.' id /* read object property */
    | Expr '[' Expr ']' /* read object property */
    | Expr '(' Expr* ')' /* function call */
    | 'function' id? '(' id* ')' Body /* function expression */
    | '(' id* ')' '=>' (Expr | Body) /* arrow function */
    | 'class' id? ('extends' Expr)? '{' ClsElm* '}' /* class expression
        */
    | id AssOp Expr /* variable (re-)assignment */
    | id BinOp Expr
    | UnOp id
Decl ::= ('var' | 'let') id ('=' Expr)? /* variable declaration */
    | 'const' id '=' Expr /* constant declaration & initialization */
    | 'function' id '(' id* ')' Body /* function declaration */
    | 'class' id ('extends' Expr)? '{' ClsElm* '}' /* class declaration
        */
Literal ::= <number> | "<string>" | '<string>' | true | false | null
    | '{' Prop* '}' /* object literal */
    | '[' Expr* ']' /* array literal */
Body ::= '{' ( Stmt | Decl )* '}'
ClsElm ::= id '(' id* ')' Body /* class method */
    | 'constructor' '(' id* ')' Body /* constructor method */
Prop ::= id ':' Expr
    | '[' Expr ']' ':' Expr
AssOp ::= '=' | '+=' | '-=' | ... /* assignment operators */
BinOp ::= '+' | '-' | ... | '==' | '===' | '!=' | ... /* binary operators
      */
UnOp ::= '+' | '-' | 'typeof' | 'instanceof' | 'delete' | '++' | '--' /*
     unary operators */
```

Listing 3: Declarations

```
var a = 1;
if(true){
    var b = 2;
    let c = 3;
}
function d(e){
    var f = 6;
    const g = 7;
    function h(){
        var i = 8;
    }
}
const j = function(){};
class K { }
let m;
```

Listing 4: Destructuring

```
var [a, b] = [1, 2];
var {c, d} = {c: 3, d: 4};

var [e, ...rest] = [5, 6, 7];
// rest == [6, 7]
var {foo: f} = {foo: 8};
// f == 8
```

is unfortunate in the latter two cases, as the programmer would ususally like to distinguish those cases more clearly. But for functions, **typeof** will conveniently return `"function"`. Finally, when applied to a variable name which is not declared, **typeof** `undeclaredVariable` will also return `"undefined"`. However, when applied to a variable that is declared, but in the temporal dead zone (see 2.4), it will throw a `ReferenceError`.

Finally, there is special syntax for creating regular expressions, which are mostly Perl-compatible. A regular expression literal is enclosed in single forward slashes: `/ab+c/i`, which creates a `RegExp` instance that can be used to match, search and replace in strings.

2.4 Declarations and Scoping

Variables and functions in JavaScript are lexically scoped. There are three kinds of scope: script/global scope, function scope, and block scope, and six different ways an identifier can be declared.

All declarations in JavaScript are hoisted: while a variable can be declared wherever it is syntactically valid, after parsing the declaration is lifted to the top of the scope. Their initial value, after the scope is entered but before the declaration/initialization is evaluated, depends on the kind of declaration.

A variable declared by **var** has either global scope, when it's declared outside any function, or function scope when it is inside one. In listing 3 *both* `a` and `b` have global scope. Even though `b` is declared in the block of the **if**-consequent, it is visible in the entire script as its declaration is lifted to the global scope. Variable `f` is local to the functions `d` and `h`, and `i` is local only to function `h`. Variables declared by **var** at the global scope are also accessible as properties of the global object, and vice versa: properties added to the global object are visible as if they were declared by **var** in the global scope. After **var** declared variables are hoisted, or when the are declared without an initializing expression, they have

an initial value of `undefined`. A **`var`** declared variable may be re-declared with **`var`**, which will have no effect beside the evaluation of the initializing expression.

Variables declared by **`let`** or **`const`** have block scope. In contrast to variable `b` in the aforementioned listing, `c` is local to the **`if`** block and not visible in the rest of the code. While `j` is at the same level as the global scope, and it is visible to the entire script, it is not added as a property of the global object. Like **`var`** variables, **`let`** and **`const`** variables are hoisted, however they may not be read from or written to before their declaration is evaluated. This is known as the temporal dead zone. A **`let`** declaration without an initializating expression, such as `m` in the code above will be initialized to `undefined` when its declaration is reached. Variables declared by **`const`** may not be reassigned and require an initializing expression.

Function declarations, such as `d` and `h` in listing 3, have the same scope as **`var`** declarations: `d` is visible to the entire script and `h` only inside `d`. Unlike **`var`** declarations, the definition of a function declaration is hoisted with it. That is, a declared function may be called before its declaration is reached. When multiple functions are declared with the same name, only the last one is kept. All of this applies to function *declarations* only. To function *expressions*, such as the initializataion of `j`, the rules of the variable's declaration apply. I.e. `j` may only be called after its declaration statement is evaluated, and may not be re-declared. The named parameters to a function, `e` in the example listing, act as **`var`** declared variables as well, except that no two parameters in the same function may have the same name.

Declarations of **`class`**es follow the same rules as those of **`let`**: they have block scope, may not be referenced before the declaration is evaluated, but can be reassigned. The declaration of `K` in listing 3 could also have been written as `let K = class {};`.

Finally, there is syntax for destructuring declarations and assignments, as shown in listing 4. This makes it more convenient to initialize variables with values taken from objects or arrays, without needing to define temporary values.

2.5 Functions

JavaScript has three kinds of functions: regular, arrow functions, and class constructors. All functions are also objects and can be used as values to be assigned to variables, passed as parameters, or returned from functions. A function can take zero or more parameters, although the JavaScript engine will not enforce that arguments for named parameters are actually passed. Parameters without a value will default to `undefined`. Conversely, a function may be called with more arguments than it has parameters. Values for those arguments can be retrieved via the built-in `arguments` variable. Functions may return a value to the caller via the **`return`** statement, if it is omitted, not reached, or written as an empty return statement, the value `undefined` is returned.

Regular functions can be created by a **`function`** declaration or a function expression. Either way, calling them has the same syntax, which can be seen in listing 5.

Besides their parameters, functions also receive a **`this`** value, which is also

Listing 5: Function parameters and returns

```
// Declare function foo
function foo(a, b){
    console.log('foo called with: a = '+ a + ', b = ' + b);
}

foo(1, 2); // will print "foo called with a = 1, b = 2"
foo(4); // will print "foo called with a = 4, b = undefined"
foo(5, 6, 7); // will print "foo called with a = 5, b = 6"

// Create an anonymous function and assign it to bar
var bar = function(a){
    if(a > 0){
        return a * a;
    }
    if(a < 0){
        return; // empty return statement
    }
};

bar(2); // returns 4
bar(-1); // returns undefined
bar(0); // also returns undefined
```

available outside any function call where it is the value of the global object. Inside a function **this** depends on how the function was called. When called directly, e.g. as `foo()`, **this** will be `undefined`, but when called as a member of an object, e.g. `object.foo()`, the **this** value is set to that object. Note that this is determined at the function's call, not its definition. It is possible to define a function on one object, but call it with a different object as its **this** value. Examples of **this** can be found in listing 6.

A regular function can also be called via the **new** operator, in this case **this** is a new empty object with its prototype (see 2.6) set to the function's **prototype** member. This forms the basis of JavaScript's object orientation system. There is, again, no specific syntax needed to make a function callable as a constructor, although by convention the name of such a function starts with a capital letter. These are called constructor functions, and an example of one is shown in listing 7.

Arrow functions have a more compact notation, as shown in listing 8, but also some limitations: they

1. have no declaration form and can only be created as arrow function expressions,
2. cannot be called with **new**,
3. do not receive their own **this** value, but inherit it from their lexical scope.

That last item is also considered a feature, it allows the programmer to create for

Listing 6: Variations of `this`

```
function foo(){
    return this;
}

var object = {
    value: 42,
    method(){
        return this.value;
    }
};
var other = {value: 24};

foo(); // returns undefined
object.method(); // returns 42

other.method = object.method; // copy a reference to the function
other.method(); // returns 24, the 'value' of other

object.foo = foo;
object.foo(); // returns 'object'
```

Listing 7: Constructor functions

```
function Person(name){
    this.name = name;
}

// This method is available on every object created by new Person
Person.prototype.greet = function(){
    return "Hello " + this.name;
};

var p = new Person("Gordon Freeman"); // a new object with {name: "Gordon
      Freeman"}
p.greet(); // returns "Hello Gordon Freeman"
```

Listing 8: Arrow functions

```
// Simple arrow functions
var approximatePi = () => 355/113; // no arguments
approxmiatePi(); // returns 3.1415929203539825
var squarer = a => a * a;
squarer(2); // returns 4
var summer = (a, b) => a + b;
summer(3, 4); // returns 7

// Arrow function with a more complex body
var normalize = (x, y) => {
    const len = Math.sqrt(x*x + y*y);
    return [x / len, y / len];
};
normalize(3, 4); // returns [0.6, 0.8]

// Demonstrating the effect of 'this' inside an arrow function
var object = {
    count: 10,
    // increments 'this.count' after the given number of milliseconds
    delayedIncrement(delay){
        // setTimeout calls the given function after the delay,
        // passing the global object as this.
        setTimeout(() => { // but the arrow function ignores it
            // and instead keeps the this value of delayedIncrement
            this.count++;
            console.log('incremented', this.count);
        }, delay);
    }
}

object.delayedIncrement();
```

example, a callback function that can refer to the **this** value of the function that creates the callback without needing to use **Function.prototype**.bind or create an alias variable.

The last kind of function is the class constructor. In contrast to arrow functions, class constructors must always be called via **new**. They can be created by a class declaration or class expression for clearer way of defining classes in JavaScript. The constructor function of listing 7 would nowadays be written as in listing 9.

JavaScript functions support a mechanism called closures, where the variables defined outside the function scope are available within it, even outside the surrounding scope.

In listing 10, when privateCounter returns, its count variable has gone out of scope. However, the read method of the returned object can still access it as a reference to it has been captured in the closure. Better yet, the increment method can modify count and its new value is correctly retrieved by read. A new closure is created for every call to privateCounter. When other.read is called,

Listing 9: Classes

```
// Functionally identical to the Person example from before
class Person {
    // Executed when new Person(...) is called
    constructor(name){
        this.name = name;
    }
    // Added to the prototype of Person
    greet(){
        return "Hello " + this.name;
    }
}

var p = new Person("Gordon Freeman");
p.greet(); // returns "Hello Gordon Freeman"
```

Listing 10: Closure

```
function privateCounter(){
    var count = 0;
    return {
        read: function(){
            return count;
        },
        increment: function(){
            count++;
        }
    };
}

var counter = privateCounter();
counter.read(); // 0
counter.increment();
counter.increment();
counter.read(); // 2

var other = privateCounter();
other.read(); // 0
```

Listing 11: Objects

```
var object = {
    value: 2,
    // Called when object.getter is read from
    get getter(){
        return this.value;
    },
    // Called when object.setter is written to
    set setter(newValue){
        this.value = newValue;
    },
    // Allows object.value to be manipulated via object.alias
    get alias(){
        return this.value;
    },
    set alias(newValue){
        this.value = newValue;
    }
};

object.value; // 2
object.getter; // also 2

object.setter = 3;
object.setter; // undefined: no 'get setter' defined
object.value; // now 3

object.getter = 6; // does nothing, no 'set getter' defined
object.getter; // still returns 3

object.value = 4;
object.alias; // also 4
object.alias = 5;
object.value; // now 5
```

it returns 0 because it has its own reference to `count`.

2.6 Objects

The object is the only compound data type in JavaScript. Although arrays have special syntax for manipulating them, they are just objects with special behaviour, just like functions. A JavaScript object is a collection of key/value pairs, called properties. There are two kinds of properties, data properties, and accessor properties, in both cases the key is either a string or a symbol. A data property has a value associated with it, which can be any JavaScript value, including other objects or functions. An accessor property does not contain a value, but allows the programmer to define functions to be executed when that property is read from, or written to, or both.

Beside their key, value or get/set attributes, properties also have a *configurable*

and an *enumerable* attribute, which are both booleans. The *configurable* attribute determines whether the property may be deleted or have its other attributes, including the getter and setter of accessor properties, changed. Though if false, the value of a data property may still be changed. The *enumerable* attribute is used to hide the property from `for-in` loops, the `Object.keys` function, and the spread operator. Data properties also have a *writable* attribute, which prevents re-assigning the value if false, though this attribute can still be changed if the *configurable* attribute is true. By default, all these attributes are true, but they can be changed by the programmer via the `Object.defineProperty/Object.defineProperties` functions. An example of getter and setter methods can be found in listing 11.

Objects themselves also have attributes, the most significant of which is the *prototype* attribute. Note that this is not the same as a `prototype` property, although it is related. The value of an object's *prototype* attribute can either be `null` or another object, which itself has a *prototype* attribute that can be `null` or another object, and so on. When a property is requested on an object, e.g. via `obj.member`, the interpreter first looks at the object itself, if it has a property with that name, its value is returned. If it does not, the interpreter looks at the *prototype* slot, if it is an object, the interpreter looks for the property on that object, and so on up the prototype chain. If the interpreter reaches an object without a prototype, i.e. *prototype* is `null`, without finding the property, it returns `undefined`, indicating there is no property with that name. When a property is directly defined on an object, it is called its own property, other properties accessible on the object are called inherited properties. Listing 12 shows an example of a prototype chain.

The *prototype* attribute is a reference: when a property is changed on an object, all the other objects that inherit that property will see the change. On the other hand, setting a property on an object that inherits a property with the same name will create it as an own property on that object, and not change the object it inherits from.

As mentioned before, arrays are a special kind of object. They have a convenient syntax for creating them: `arr = [1, "two", true]`, which will create an object with numeric property keys, converted to strings, mapping to the values specified, and a `length` property. These objects also have their *prototype* set to `Array.prototype`, which has a number of built-in functions for manipulating arrays, like `fill` and `splice`, finding specific values, like `find` and `indexOf`, and creating new arrays given a predicate function, like `map` and `filter`. The `length` property is a number which is always one greater than the largest numeric property of the array. It updates automatically when the array is changed, and can be assigned to shrink or enlarge the array. The code in listing 13 shows various behaviours of an array.

2.7 Statements

JavaScript has conditional execution via the `if-else` statement, and the `while` and `do-while` loops. Furthermore, there are 3 flavors of for loops.

The regular `for` loop, like Java and C, has an initializer, condition, and update expression, all of which are optional, and will repeat a statement as long as the

Listing 12: Prototype chains

```
// Create a new empty object with a null prototype
var foo = Object.create(null);
foo.a = 1;
// Create a new empty object with foo as its prototype
var bar = Object.create(foo);
bar.b = 2;
var qux = Object.create(bar);
qux.c = 3;

// The prototype chain is: qux -> bar -> foo -> (null)
qux.c; // 3, from its own property
qux.b; // 2, inherited from bar
qux.a; // 1, inherited from foo via bar
qux.d; // undefined, neither qux, nor bar, nor foo have property d

bar.b = 20;
qux.b; // 20, change in bar is reflected in qux

qux.a = 10; // does not modify foo.a, but creates own property on qux
foo.a; // still 1
```

Listing 13: Arrays

```
// Like with regular objects, values need not have the same type
var arr = [1, "two", true];
// Same access syntax as objects
arr[0]; // 1, numeric indices are converted to strings
arr["1"]; // "two"
arr[3]; // true
arr.length; // 3
arr[4] = "four";
arr.length; // 5, length is not necessarily the number of elements
arr[3]; // undefined, properties not set are still undefined
```

condition is true. The initializer can also be a variable declaration, which, if it is declared `let` or `const` will be scoped to the `for` statement.

The `for-in` statement iterates over the enumerable properties of an object, including those inherited from its prototype chain. It can be used to iterate over an array, however the standard does not guarantee the order in which the properties will be visited. Also, since it includes inherited properties, if any code added properties to `Object.prototype` or `Array.prototype`, those may also show up unexpectedly.

The last loop statement is `for-of`. It can be used to iterate over iterable objects: objects that implement the *iterable* protocol. These include `Array`, `String`, and `Map`. The iterable protocol specifies that an object must have a method accessible via the `Symbol.iterator` symbol, which returns an object with a method called `next`. That `next` must return an object with a property called `value`, which is the value that `for-of` receives, and a `done` property, to be set to `true` when the iterator is finished.

2.8 Standard Libarary

It has shown up in many examples already, much of the JavaScript functionality is available through standard built-in objects and functions, forming the standard library. These are mostly regular JavaScript objects and functions, and can to some extent be manipulated by user code. Changing the built-in objects is generally not recommended, as it will affect all code in the context. On a web page which may have many different scripts loaded, this can cause complex bugs. These built-ins are available as global variables and as properties on the global object. Examples of these global built-in objects are `Math`, `Date`, `Object`, `Array`, and `Function`.

Unlike many programming languages, JavaScript lacks any built-in standard way of manipulating files, creating network connections, or running other programs. This is by design, as allowing web pages such control over the visitor's computer would pose great security risks. Any input/output capability is provided by the runtime in which the language interpreter is hosted, like the DOM APIs in web browsers, or the modules bundled with Node.js.

Listing 14: Power by repeated multiplicaton

```
function power(x, n){
   let ret = 1;
   while(n > 0){
      ret = ret * x;
      n--;
   }
   return ret;
}

// raise a run-time random value
      to the power 3
let a = power(Math.random(), 3);
console.log('r**3', a);
```

Listing 15: Power specialized for raising to the third

```
function power3(x){
   let ret = 1;
   {
      ret = ret * x;
      ret = ret * x;
      ret = ret * x;
   }
   return ret;
}

let a = power3(Math.random());
console.log('r**3', a);
```

3 Partial Evaluation

Partial evaluation is the process of transforming a program where certain values are known ahead of time into a more specialized version [6], generally with the intent of creating a program that executes faster at run time. For example, the program from listing 14 can be specialized into the program in listing 15.

In the specialized version, which is called the residual program, the **`while`** loop is eliminated, and the specialized function `power3` only takes 1 parameter. The value 3, passed to the `n` parameter is known as a compile-time or static value, while the `x` parameter is a runtime or dynamic value. Static values are known by the partial evaluator and eliminated from the residual program. An expression that computes a dynamic value on the other hand, cannot be computed ahead of time and thus remains in the output of the evaluator. The specialized function `power3` in listing 15 is also called the Futamura projection of `power` at `n = 3`.

Static, in the context of partial evaluation, does not mean that the value or variable cannot change in the program. However, all computations that involve a static value can be done ahead of time. In listing 14, the `n` parameter is called static, even though it does change during the execution of the function, because all those changes can be computed by the partial evaluator.

Note that partial evaluation is not the same as partial application. Partial application takes a function of one or more parameters and creates a function with fewer parameters, whereas partial evaluation can be applied to an entire program. Also, partial application generally occurs at run-time, while a partial evaluator creates a residual program ahead of time.

It is important that the residual program produces the same results, and has the same effects as the original. The partial evaluator must not evaluate code that has side effects, like manipulating files, or precompute values that are different on each program invocation, like calls to `Math.random()` or **`new`** `Date`.

Partial evaluators are categorized into online and offline partial evaluators [10].

Listing 16: Hybrid Partial Evaluation annotation

```
1 // using the power function from before
2
3 // Specialize power with the first parameter being dynamic, and the
4 // second having the static value 3.
5 power(Math.random(), $HPE.CT(3));
6 // Leave the following call as is.
7 power(Math.random(), $HPE.RT(1000));
```

An offline partial evaluator analyses the code to be optimized, by a process called binding time analysis [11], and then partially evaluates the parts it determines to be static. Online partial evaluators, on the other hand, evaluate the given program as long as they can, optimizing the code using information gathered during the partial evaluation. While online partial evaluators may produce better results than offline ones, they are also harder to implement and it is more difficult to show their correctness [10]. Offline partial evaluators, on the other hand, need to be more conservative in their optimization, as they cannot always fully determine which parts of the program are static.

3.1 Hybrid Partial Evaluation

A large issue with partial evaluation is code explosion. In the `power` example, if the function is specialized for a large exponent, the residual code will have the body of the **`while`** loop repeated that many times. Though it avoids the overhead of the loop, the size of the code starts to have a greater impact on performance. The partial evaluator might be equipped with heuristics to stop specializing a function when its residual code becomes too large, but it is all but impossible to define a general limit on residual code size.

Hybrid Partial Evaluation [16] sidesteps this issue by making the application programmer responsible for deciding what parts of the program to specialize. The programmer adds annotations to the program that direct the hybrid partial evaluator to specialize certain function calls, while leaving the rest of the code untouched.

In listing 16 and beyond, a call to `$HPE.CT` indicates that the value of its argument is static, while `$HPE.RT` means the value it receives is dynamic, and should remain in the residual code as is. These annotations are explained in more detailed in section 5.1.

As Hybrid Partial Evaluation does not use binding time analysis, it is possible to run into situations where a variable that has a static value at first, is later assigned a dynamic value. Either because a later assignment involves a dynamic value directly, or because it occurs within a block that is evaluated at run-time. An example of this can be found in listing 17.

Depending on the run-time value of `b`, the value of `a` can either stay 1, or become 2. Hence, the value of `a` would need to become dynamic as well. Though it is possible to solve this issue in a full partial evaluator [12], it again increases

Listing 17: A variable changes from static to dynamic

```
let a = $HPE.CT(1);
if($HPE.RT(b)){
    a = 2;
}
console.log('a', a);
```

```
// a not declared as it is static
if(b){
    a = 2; // error here
}
console.log('a', ?);
```

the size and complexity of the residual code. Hybrid Partial Evaluation takes a simpler approach to this problem, by disallowing such constructs. A variable that starts out with a compile-time value may only be updated by compile-time computations. For the code of listing 17, a hybrid partial evaluator will detect that the assignment within the consequent of the **if** statement depends on a run-time value and raise an error. It is then up to the programmer to remedy the issue.

In addition to variables, object values are not allowed to move from static to dynamic and vice versa. That is, an object exists either completely at partial evaluation time, or at run-time, and compile-time object values are never residualized. Therefore, in a similar way to variable manipulation, the hybrid partial evaluator will raise an error if a compile-time object ends up being referenced in residual code.

3.2 Control Flow

An issue that the original Hybrid Partial Evaluation paper does not address, but is allowed and commonly used in JavaScript, are interruptions in local control flow. These are caused by the **break**, **continue**, **return**, and **throw** statements.

When a **return** statement ends up in the residual code, any subsequent **return** statement must also be residualized. Otherwise the partial evaluator might return a static value from a function that should return a different value at run-time. An example of this can be found in listing 18. In the first two calls to `foo` with static values, the partial evaluator can evalute the entire function and return a compile-time value. However, in the two calls with a dynamic argument, the first **return** statement is residualized inside the **if** statement. The second **return** statement must therefore also be residualized, or the partial evaluator would return **true** for both run-time calls.

This residualizing of **return** statements must also happen if there is residual code preceding a **return** statement. The call to `bar` in listing 17 cannot be evaluated away despite its return value being static, since the `console.log` call is supposed to happen at run time.

The implementation of this mechanism is further explained in section 5.6. To our knowledge this is the first time that this solution is implemented for Hybrid Partial Evaluation.

The **break** and **continue** statements cause this problem when unrolling loops, which has also been described in [15]. In listing 19 the loop needs to be unrolled because its conditional depends on the compile-time object `arr`. However, inside

Listing 18: Residual `return` statements.

```
function foo(a){
   if(a){
      return false;
   }
   return true;
}
console.log('ct 0', foo(0));
console.log('ct 2', foo(2));
console.log('rt 0', foo($HPE.RT(0)));
console.log('rt 2', foo($HPE.RT(2)));

function bar(a){
    console.log('bar', a);
    return a;
}
bar(2);
```

```
function foo_0(a) {
  if (a) {
    return false;
  }
  return true;
}
console.log("ct 0", true);
console.log("ct 2", false);
console.log("rt 0", foo_0(0));
console.log("rt 2", foo_0(2));

function bar$0() {
  console.log("bar", 2);
  return 2;
}
bar$0();
```

Listing 19: Residual `continue` statement.

```
const arr = $HPE.CT([3, 2, 1]);
for(let i = $HPE.CT(0);
       i < arr.length; i++){
    if($HPE.RT(X)){
        continue;
    }
    console.log('i', i, arr[i]);
}
```

```


{ // for loop unrolled
    if(X){
        continue; // error here
    }
    console.log('i', 0, 3);
    // repeated twice more...
}
```

the loop, the `continue` statement must be residualized because it depends on the dynamic variable `r`. This would cause the `continue` statement to appear outside a loop, which is not allowed. If the loop were contained inside another loop that is not unrolled, it would cause a residual `continue` statement to apply to the outer loop, changing the run-time behaviour.

The solution for a full partial evaluator would be to analyse the loop body in advance and not unroll it in this situation. However, in a hybrid partial evaluator that is not possible because it means residualizing the compile-time object `arr`. Just as in the case of listing 17, the hybrid partial evaluator will throw an error in this situation.

3.3 Classes

Although objects in Hybrid Partial Evaluation cannot be half-static and half-dynamic, it is possible to specialize classes. When a constructor is called, the hybrid partial evaluator checks if all the arguments to it are compile-time, and if so, returns a compile-time object. If there are run-time parameters, the evaluator stores the static values in the partial environment and marks the dynamic fields as such. It then specializes the other methods of the class for the compile-time

fields of the class instance, treating any parameters to the methods themselves as dynamic. The specialized methods and constructor are then added as a specialized class to the residual program. When a class or method is specialized, it is also memoized so that subsequent calls to the same function with the same compile-time arguments can reuse the specialized version. This prevents the evaluator from running into infinite loops when specializing recursive methods.

While compile-time objects are not residualized, it is possible to residualize method calls on a static object, when the method call involves one or more run-time arguments. In this case the method is specialized and added as a static method on the original class.

4 Alternator

Alternator is a meta-circular interpreter of JavaScript, that is: an interpreter of JavaScript, written in JavaScript itself. The interpreter cannot run on its own, but needs to run on top of an existing JavaScript interpreter. The meta-circular interpreter is used as the basis for the partial evaluator.

Alternator implements most of the functionality required by ECMA-262 9th edition [3], with the notable exceptions:

- asynchronous functions via `async function` ... and `await` ...,
- generators via `function*` and `yield` ..., and
- modules via `import` ... and `export` ....

We omitted these features because their implementation would make the interpreter significantly more complex. Additionally, the interpreter only supports strict mode, and lacks the legacy behaviours specified in ECMA-262 annex B. Non-strict mode and the legacy behaviours are only needed to support old code, and are not recommended practice anymore.

4.1 Structure

The ECMAScript specification [3] consists largely of algorithm definitions in pseudo code dictating how a conforming implementation of the language should behave. Nevertheless, for Alternator we implemented only a few of those algorithms literally, opting for simpler implementation strategies most of the time. There are two main reasons for this:

- The specification deals with non-strict mode, legacy behaviours, and other features we deliberately do not implement.
- Being written in JavaScript and running on a conforming implementation, parts of the required behaviour don't need to be re-implemented in Alternator.

Alternator is divided into three components: Scoping, Runtime, and Evaluator. Scoping implements the declaration and scoping rules, matching declarations and identifier references to their appropriate scope. The Runtime component manages built-in objects and the creation of user objects, functions, and classes. Finally, the Evaluator takes care of the actual execution of JavaScript code.

The process of evaluating code is also composed of three phases. First, the code text is turned into an Abstract Syntax Tree (AST) by the parser. Then, the Scoping component is used to gather variable and other declarations, add them to Scope objects, and attach those to their appropriate AST nodes. Finally, the augmented AST passed into the Evaluator, which starts at the top of the tree and processes the nodes by recursive descent.

4.2 Parsing and Preparation

Alternator uses the Shift parser and its AST format [17]. We chose this AST over the more popular ESTree [5] because it provides more specific node types.

Listing 20: Identifier nodes

```
1 let id = foo;
2 bar = 12;
```

Listing 21: A variation of the power example

```
1  function power(x, n){
2      let ret = 1;
3      while(n > 0){
4          ret = ret * x;
5          n--;
6      }
7      return ret;
8  }
9
10 let a = power(2, 3);
11 console.log("2**3 =", a);
```

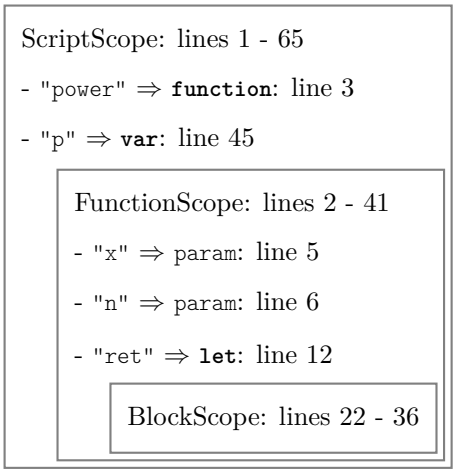


Figure 1: Scope tree of listing 21

For example, for identifiers ESTree provides one node type: `Identifier`, which is used when it's in a binding position or as an expression.

In listing 20, `id`, `foo`, and `bar` are identifiers, but the Shift AST splits these into separate node types: `id` is a *`BindingIdentifier`*, while `foo` is an *`IdentifierExpression`*, and `bar` is an *`AssignmentTargetIdentifier`*. This means the evaluator can determine the role of an identifier from its AST node alone, without needing to examine the context.

Before passing a code string to the parser, the evaluator ensures that it starts with the sequence `"use strict";` to put the parser in strict mode so it can catch certain errors before the code is evaluated. The Shift parser then produces an AST like listing 34 in the appendix, from parsing the code in listing 21.

Note how the three different uses of `ret` within the `power` function of listing 34 are marked with the different node types. As the *binding* of a *`VariableDeclarator`* on line 12, it is marked as a *`BindingIdentifier`*. Then, when it is re-assigned in an *`AssignmentExpression`* on line 24, it takes the role of an *`AssignmentTargetIdentifier`*. And finally, on the expression side, where its value is used to compute `ret * x` on line 26, the variable name is referenced in an *`IdentifierExpression`*.

These different node types make it easy for the Scoping component to find the declaration, writes, and reads of the variables, functions, and classes in the AST.

4.3 Scoping

Before gathering declarations, they need a scope to be limited by. As described in section 2.4, there are 3 different kinds of scope in JavaScript: global, function, and block. These correspond to distinct AST node types as well. The global scope is attached to the *`Script`* node, of which there can only be one. A function scope is attached to every *`FunctionExpression`*, *`FunctionDeclaration`*,

ArrowExpression node, as well as every *Getter*, *Method*, and *Setter*. They are attached to those nodes, rather than to *FunctionBody* because (formal) parameters are also part of the function scope, so the *BindingIdentifier*s in those need to be included in it. Finally, block scopes are attached to *Block* nodes, as well as *CatchClause*, the *For*Statement*, and *SwitchStatement** nodes. Again, because those nodes may include declarations which are part that scope, but in the AST appear above the corresponding block statement.

With the scopes attached to their appropriate AST nodes, the declarations within those scopes can be recorded in them. Once again, we traverse the tree, this time looking for declaration nodes: *ClassDeclaration*, *FunctionDeclaration*, *VariableDeclaration*, *FormalParameters*, *Setter*, and *CatchClause*. Inside those nodes we collect the *BindingIdentifier*s and call `declare` on the nearest enclosing scope of the declaration. The `Scope` class then takes care to match the declaration type with the scope type, or pass the declaration to its parent scope. For example, a **`let`** declaration in a block scope is added to that block scope, but a **`var`** declaration is passed up to the nearest function or script scope. Figure 1 shows the scope tree for the AST of listing 34.

The last ingredient we need before we can start evaluating client code is an implementation of the JavaScript standard library. This is provided by the Runtime component. It creates a shallow copy of the host's (the V8 engine, via Node.js) global object, which contains the standard built-in objects like `Object`, `Array`, and `Math`, with their functions. As it is a shallow copy, this can create issues when the client code modifies those objects, since those changes will affect how Alternator uses those objects as well. For example, if client code does `Array.`**`prototype`**`.map = `**`null`**`;`, the `map` method on array instances in Alternator will become **`null`** too and break the interpreter. Although such modifications of built-ins are allowed in JavaScript, it is strongly discouraged, and we assume new client code won't do it.

While scopes keep track of variable declarations, they do not hold the actual values during evaluation. Like the built-in objects from the Runtime, scope objects are not expected to change much during evaluation, although it is possible, e.g. via `eval("var foo");` which will declare the variable `foo` only when the `eval` call is evaluated.

4.4 Activation Records

Activation Records (ARs) map the variables declared in a scope to their value during evaluation, as well as the **`this`** value, **`new`**`.target` in function calls, and the **`super`** reference in the constructor of a derived class. These are first set to default values, depending on how the variables were declared. Variables declared by **`var`** are initially set to `undefined`. For those declared by **`let`**, **`const`**, or **`class`**, the initial value is a special *invalid* marker, which is removed when their declaration is actually evaluated. This is so the evaluator can throw the mandated `ReferenceError` if such a variable is referenced before its declaration. Finally, **`function`** declarations initialized with a callable function implementing the function definition, since declared functions in JavaScript may be called before their declaration is reached during execution.

The activation record for the global scope has some more special behaviour:

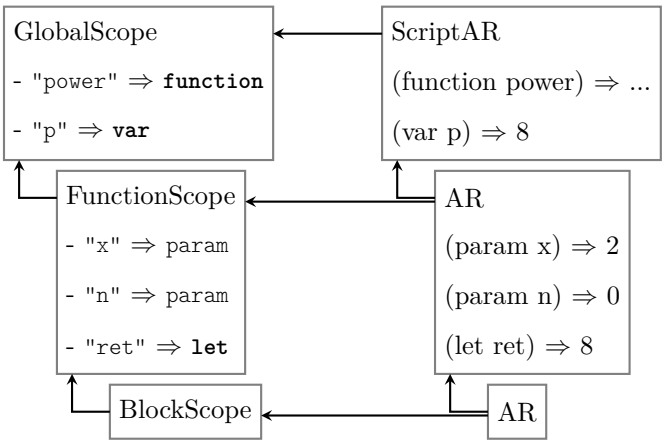


Figure 2: Scopes and activation records

`var`-variables and declared functions are added as properties of the global object, e.g. `var a = 2;` at *Script* level can also be accessed as `this.a;` `/*== 2 */`, and vice versa: properties added to the global object are added as globally declared variables or functions.

Every time Alternator evaluates an AST node with a scope, it creates a new AR for that scope. With the exception of the global one, every AR also has a link to its parent, which is the AR of the parent scope of the ARs scope. The AR is discarded when evaluation for its scope ends, i.e. when the evaluator is done with the AST node of the associated scope. The exception is when a function is defined within an AR that outlives its scope, e.g. when a function is created in, and returned from, another function. This is implemented by capturing the AR in a closure in Alternator: we use JavaScript closures to implement JavaScript closures. It can be seen in listing 23.

To resolve a variable reference, the evaluator queries the scope of the current AR. If it does not have a variable by that name, it asks the parent scope, and so on until the variable is found, or it reaches the global scope without finding it and will then throw a `ReferenceError`. When a variable by the queried name does exist, the evaluator retrieves or sets the value from/in the AR associated with the scope in which its declaration was found.

4.5 Completion Records

The evaluation methods of the evaluator return a completion record, of which there are 5 types: *normal*, *throw*, *return*, *break*, and *continue*. The ECMA specification uses the same system. A *normal* completion indicates evaluation completed without issue and holds the value that resulted from it, e.g. 5 as the value of the binary expression `2 + 3`, or `undefined` from declaration statements. The other 4 completion record types are *abrupt* completions. Most evaluator methods don't deal with these directly, only making the distinction between normal and abrupt results. When a child node returns an abrupt completion the parent immediately returns it up to its parent, until a method that does handle

Listing 22: Completion records

```
1 while(true){
2    let value = +prompt("enter a number");
3    try{
4       if(value <= 41){
5          throw new Error("too low");
6       }else if(value >= 43){
7          throw new Error("too high");
8       }
9       console.log("just right:", value);
10      break;
11   }catch(ex){
12      console.log(ex.message, "try again");
13   }
14 }
```

them.

In listing 22, the `throw` statements on lines 5 and 7 result in a throw completion. Their surrounding `if` statement on line 4 sees these as non-normal and return them back to their parent, which in this example is the `try-catch` statement on lines 3 to 13. This evaluator method does handle the throw completion, by evaluating the `catch` block. If a throw completion reaches a Script evaluator, it is thrown to outside the interpreter as an uncaught exception.

Return completions are created by `return` statements, and handled by the function-body evaluator, which turns it into a normal completion as the result of the function call. A return completion that makes it to the Script evaluator means that there was a `return` statement outside a function body, which is then signaled as an error.

The break and continue completion records are also the result of their similarly named statements, and are handled by the various loop evaluators: `for`, `for-in`, `for-of`, `while`, and `do-while`. Additionally, break completions are also caught by the `switch` statement.

Break and continue completions can have a label value in order to break or continue with a specific labeled loop. If such a completion has a (non-empty) label, it will only be handled by a loop that was prefixed with the same label. On the other hand, a break/continue completion without a label will still be handled by a loop that is prefixed by a label.

4.6 Evaluation

An `Evaluator` instance is initialized with a `Runtime`, with that, it makes a global scope and global AR, and declares and defines the built-in global variables on them. We can then call the `run` method, with a 'raw' AST. The Evaluator will use the Scoping component to add scope information and collect declarations as described in 4.3. With the augmented AST and the global AR, it calls `runScript`, which will actually execute the script.

The statements of the script are evaluated one at a time, in order, by calling `runAny` on the Evaluator. This method inspects the `type` field which every AST node has, and calls the appropriate method on `Evaluator` that follows the name `'run' + ast.type`. If `runAny` is called with a node for which there is no method on `Evaluator`, it means that functionality is not implemented and `runAny` will immediately throw an error and halt the Evaluator.

To illustrate the evaluation process, we will describe how Alternator executes the program in listing 21 after it has been turned into the AST in listing 34. The first statement is the function declaration from lines 1 to 8 turned into a *`FunctionDeclaration`* node. When `runAny` is called with that node it will call `runFunctionDeclaration`.

```
1 function power...
```

The `runFunctionDeclaration` method in fact does nothing, since function declarations are already processed by the Scoping component. It returns a normal completion with a value of `undefined`.

```
1 let a = ...
```

The next statement is the *`VariableDeclarationStatement`* derived from line 10, handled by `runVariableDeclarationStatement`. It is composed of a list of *`VariableDeclarator`*s, which in turn are composed of a `binding` and an optional `init`ializer. The `binding`, in this case a *`BindingIdentifier`* is handled by a different group of methods, headed by `referAny`, which functions similarly to `runAny` calling methods named by `'refer' + ast.type`. These methods return a `Reference` , that is used to assign a value to ultimately a variable identifier or an object's property. The *`BindingIdentifier`* is processed by `referBindingIdentifier` and results in an `IdentifierReference`. They can also create more complex references, by nesting `ArrayReference`s and `ObjectReference`s to implement destructuring.

```
1 ... = power(2, 3);
```

After the Reference has been created, the Evaluator runs the `init` part, on line 46 of listing 34 it is a *`CallExpression`*. The first thing `runCallExpression` does is evaluate the `callee`: an *`IdentifierExpression`* which retrieves the `power` function from the current AR. Should this fail, e.g. if the identifier were misspelled and its declaration not found, the `runIdentifierExpression` method will return an abrupt completion, in that case a *throw* completion with a `ReferenceError` exception. When `runCallExpression` receives such an abrupt completion it will just return it further up the call tree. Most `run*`-methods work like this: evaluate the sub-expressions, if they return a normal completion, use the value in it and continue, otherwise return the abrupt completion further up to a method that does handle it. After the `callee`, the `arguments` are evaluated, in the same way. Provided all of those are normal, the Evaluator can now call the user-defined `power` function.

As will be explained in section 4.7, user-defined functions in Alternator are wrapped in standard JavaScript function values. This wrapper function calls the

implementFunction method on the evaluator. It receives the AST of the function definition and the AR in which it was defined, along with the **this** value and arguments that it is called with.

```
function power(x, n){
...
}
```

The implementFunction method starts by creating a new activation record for the function scope. Then it binds the given arguments to the parameters specified by the *FormalParamters* in the params field. For arrow functions, which are handled by the same implementFunction but can be differentiated by the type field in the AST, it binds the **this** value to that of the parent AR, otherwise it is bound to the given **this** value. In the power example, the parameters x and n are set to 2 and 3 respectively. Since power is not called as a member of an object, **this** is set to undefined. The function body is then evaluated by the runFunctionBody method. This method works much the same as runScript, evaluating the statements one by one, with the exception that it can handle a ReturnCompletion by converting it to a normal one and returning its value as that of the function call.

```
let ret = 1;
while(n > 0){
    ret = ret * x;
    n--;
}
...
```

Inside the *FunctionBody* of listing 34 is another variable declaration. Here we initialize ret to 1, and move on to the *WhileStatement*. The runWhileStatement method is itself implemented with a **while** loop, which evaluates the test field and runs the statement in the body field as long as the test evaluates to a normal truthy value. Evaluating the *Block* is again similar to the evaluation of *Script* and FunctionBody, except that it creates an AR for the block scope. Should the evaluation of the body in the *WhileStatement* result in a *break*, runWhileStatement will handle it by breaking the loop, and a *continue* completion is handled automatically as it halts evaluation of the loop body until runWhileStatement catches it.

```
...
return ret;
```

After the **while** loop in the power function is a return statement, which is turned by runReturnStatement into a *return* completion with the value returned by evaluating the expression in the *ReturnStatement* node. This *return* completion ends up at runFunctionBody, which turns it into the return value of the function call, and is assigned to the a variable.

```
...
console.log("2**3 = ", a);
```

Listing 23: Abridged implementation of makeFunction

```
makeFunction(contextAr, ast){
    const run = this.implementFunction; // implementFunction from
        Evaluator
    return function(){
        return run(ast, contextAr, this, arguments, new.target);
    }
}
```

The last statement in the example script is another *CallExpression*, but this time to a built-in function `log` as a property of the `console` object. In addition to the arguments specified in the `arguments` field of the *CallExpression*, the evaluator also needs to pass the `console` object as the **`this`** value. To achieve this, whenever the evaluator runs a *StaticMemberExpression* or *ComputedMemberExpression*, it stores the value of the `object` field of that node as `lastBaseObject`. The `runCallExpression` method then retrieves this value to pass on to the function call, and sets `lastBaseObject` back to `undefined`. The built-in function is called with the arguments and **`this`** value, the same way as the user-defined function was, except that it does not go through `implementFunction` since we take the implementation from the host engine. Through `console.log` the script will print the result of the computation on the standard output: `2**3 = 8`.

4.7 Objects and Functions

Objects created and manipulated in the interpreted code are obtained straight from the engine, rather than being re-implemented in Alternator. Allocation and garbage collection is therefore also handled by the host engine. Likewise, implementations of built-in objects and functions, such as `Date` and `Math` are also copied from the host.

User-defined functions and classes are wrapped in host-created instances as well. This means calls to those don't need different treatment from built-in or external functions and classes. The function wrapper is very simple: the `makeFunction` method of the Runtime receives a reference to the AR in which the function is defined, the context-AR, and the AST node of the function expression/declaration. It then returns a new **`function`** which calls the `implementFunction` method of the Evaluator with the AST node, the context AR, and the **`this`**, `arguments`, and **`new`**`.target` that the returned client function receives when called. The `implementFunction` method of the Evaluator then evaluates the function body with the given arguments.

Classes are a bit more complicated. To implement the constructor, the evaluator needs access not only to the aforementioned AST, AR, and other arguments, but also a way to use the **`super`**-properties, and in the case of a derived class, a way to call the super constructor. Unlike **`this`**, the **`super`** keyword cannot be used as a variable, so it cannot simply be passed to the evaluator. Additionally, the Shift AST does not differentiate between a super-constructor call and super-property access, treating both as regular property-access/function calls with the

Listing 24: Abridged implementation of makeClass

```
makeClass(classAr, node, ctrNode, base){
    /* ... */
    const ctr = this.implementConstructor; // from Evaluator
    ret = class UserClass extends base {
        constructor(...args){
            // implement the super keyword
            classAr.homeObject = new Proxy(() => {}, {
                apply: (_, __, ...args) => super(...args),
                get: (_, prop) => super[prop],
                set: (_, prop, val) => super[prop] = val
            });

            // cannot pass 'this' as super has not been called yet
            return ctr(ctrNode, classAr, symInvalid, arguments, new.target
                );
        }
    }
    /* ... */
    return ret;
}
```

super keyword as the base object, or callee respectively. Because both require different evaluation, the evaluator needs to either modify or augment the AST so those expressions can be implemented separately, or create a wrapper so the expressions can be handled the same as their normal counterparts. JavaScript's Proxy mechanism provides a convenient way to do the latter. It allows the programmer to intercept and redefine the behaviour of fundamental operations on an object, such as getting/setting properties or calling the object as a function. For the implementation of the **super** keyword, the Runtime creates a proxy which intercepts the *apply*, *get*, and *set* operations on an empty function and redirects them to arrow functions that perform those actions on the **super** keyword. By creating this proxy inside the `constructor` method of the user-defined class, and using arrow functions to implement the super operations, we can sneak this functionality out of the constructor and have it be called by the Evaluator when appropriate.

Listing 25: Partial vs abstract completions

```
var a = $HPE.RT(3),
    b = $HPE.CT(2);
a + b;
```

```
var a = 3;

a + 2;
```

5 HandCrank

HandCrank is a hybrid partial evaluator, based on Alternator. It works in much the same way as the meta-circular interpreter, but allows expressions to be marked as *run-time* so that they are not evaluated directly but turned into residual code. In this chapter we will explain how we implemented Hybrid Partial Evaluation in HandCrank.

5.1 Annotations

To indicate that a value, or the result of an expression is static and should be computed in the partial evaluator, it provides a built-in function accessible via `$HPE.CT`. When this function is called with 1 argument, the partial evaluator will evaluate the expression, throw an error if the result is not compile-time, and return it as a *normal* completion otherwise. It can also be called with more than 1 argument, in which case the first argument is forced to be compile-time only if the remaining arguments are static as well, otherwise it is returned as dynamic. There is also a built-in function to do the opposite: `$HPE.RT` will evaluate its only argument in residual mode (explained in section 5.3), if the result is still static it must be a simple value that can be transformed back into an AST. Otherwise, it returns the run-time expression as a *partial completion*.

5.2 Partial and Abstract Completions

The partial evaluator uses two more completion record types, in addition to those used by Alternator. Those are the *partial completion* and *abstract completion*. An *abstract completion* is used to indicate a run-time value, unknown at partial evaluation time. Partial completions contain residual code, generated by the partial evaluator. As an example of the distinction, consider the following code:

In listing 25, the value stored into `a` is indicated to be dynamic, so it stores an *abstract completion*. The value for `b` is indicated as compile-time and stored as normal. In the evaluation of `a + b`, `runIdentifierExpression` retrieves an *abstract completion* for resolving `a`, which it turns into a partial completion with the *`IdentifierExpression`*. The evaluation of `b` results in a normal completion with the value 2. Finally, `runBinaryExpression` combines the partial completion with `a` and the normal completion with 2 into a partial completion with `a + 2`, which ends up in the residual program. Algorithms 3 and 4 show how expressions and identifiers are partially evaluated.

5.3 Partial ARs and Residual Evaluation

The Partial Activation Record (partial AR) is the partial evaluator's counterpart to the activation record in Alternator. It keeps track of the values of compile-time

Listing 26: Compile-time object

```
const obj = $HPE.CT({a: true,
    b: 2, c: "three"});
console.log('obj.a, .b, .c',
    obj.a, obj.b, obj.c);
obj.a = false;
obj.b = -2;
obj.c = "eerht";
console.log('obj.a, .b, .c',
    obj.a, obj.b, obj.c);
console.log('obj', obj);
```

```

console.log('obj.a, .b, .c',
    true, 2, "three");


console.log('obj.a, .b, .c',
    false, 2, "eerht");
// Error: object residualized
```

variables, and marks run-time variables as being dynamic. Identifier resolution in partial ARs works the same as in the 'standard' ARs, however, each time a variable is written to, the partial AR checks if the store operation is consistent. That is, if a variable was previously compile-time, it may only be re-assigned a static value. Assigning a static value to a dynamic variable is permitted, if it is a simple value, so that it can be turned back into residual code. Objects, arrays, functions, and symbols cannot be generally translated back, and are therefore not allowed to move from the partial evaluator to the residual program. This residualizing of static values is shown in algorithm 1.

While this works well enough for expressions and statements that are partially evaluated directly, we also need to account for assignments in residual blocks. For example, in listing 17, since the conditional of the `if` statement is run-time, the consequent block cannot be partially evaluated directly. Nevertheless, we need to signal that the assignment to `a` in that block is inconsistent. To facilitate this, when we emit a block of residual code, it is partially evaluated in 'residual' mode. In this mode, any assignment to a variable declared outside that block is treated as assigning a run-time value. So, the assignment `a = 3;` in the the `if`-consequent is treated as a dynamic assignment, despite the value itself being compile-time, and the partial evaluator will throw an error. The semantics of residual evaluation and assignment to variables are described in algorithms 2 and 9.

5.4 Objects

Objects in the partial evaluator can either be fully compile-time, in which case they are not residualized, or fully run-time, with one exception, explained in section 5.7. To ensure that a compile-time object is not modified in residual code, the partial evaluator keeps track of its owners, which can be partial ARs, and/or other compile-time objects. These resolve to the partial ARs from which the object is reachable, either because the object is the value of a variable in the AR or it can be reached from an object which is. Whenever a compile-time object is modified, the partial evaluator checks if none of its owners, or its owners' owners, are in residual mode. The algorithms 5, 10 and 11 show how the partial evaluator deals with objects, and listings 26 and 27 contain examples of compile-time objects and possible error scenarios.

Listing 27: Object tracking

```
let outer = $HPE.CT({a: 1,
    inner: {b: 2}});
function mangle(o, k, v){
    o[k] = v;
}
mangle(outer, 'a', 2);
mangle(outer.inner, 'b', 3);
console.log('a', outer.a,
    'b', outer.inner.b);
mangle(outer.inner, $HPE.RT(Y),
    5);
```

```


console.log('a', 2,
    'b', 3);
// Error: inconsistent
//  modification
```

5.5 Specializing Functions

The most important aspect of the hybrid partial evaluator is the specialization of function calls. Before we can do that, we first need to know if the callee is actually a built-in function, which we can't specialize, and second we need to keep track of functions that have already been specialized, and for which arguments. Both of those problems are solved in the `Registry`. It has a list of built-in objects and functions along with information on whether they can be called without side effects, e.g. `Math.sqrt` can, but `Math.random` cannot, and if they modify any arguments passed to them, e.g. `Array.prototype.splice` can add and remove elements from the **`this`** value it is called with. During partial evaluation it keeps track of user-defined functions and their specialized versions.

When a *`CallExpression`* is run in the partial evaluator, it evaluates the callee, arguments, and retrieves the **`this`** value as the regular evaluator does. It then either calls the function, specializes it, and/or re-generates the call expression with the evaluated arguments, according to the algorithm outlined in figure 3. It is implemented in the `runCallExpression` and `partializeCall` methods of the partial evaluator. Actually specializing a user-defined function is implemented in `partializeFunction`, which works very similar to `implementFunction` in Alternator. The main difference is that evaluating the function body may result in a partial completion, that is then wrapped in a *`FunctionDeclaration`* or *`FunctionExpression`* (depending on what the original AST was), to be added to the residual program as the specialized function. The semantics of this can be found in algorithms algorithms 12 to 15. Listing 28 shows a function being specialized once and called multiple times with the same static argument, but different dynamic arguments.

5.6 Non-local Control Flow

To implement the control flow rules for **`return`** statements, as explained in section 3.2, the partial evaluator uses a flag in the partial AR of function bodies. Whenever a **`return`** statement is residualized, it sets the flag, and when subsequent **`return`** statements are encountered while the flag is set, they are also forcibly residualized. If such a **`return`** statement returns a static object, the partial evaluator throws an error that indicating that an object is being moved from

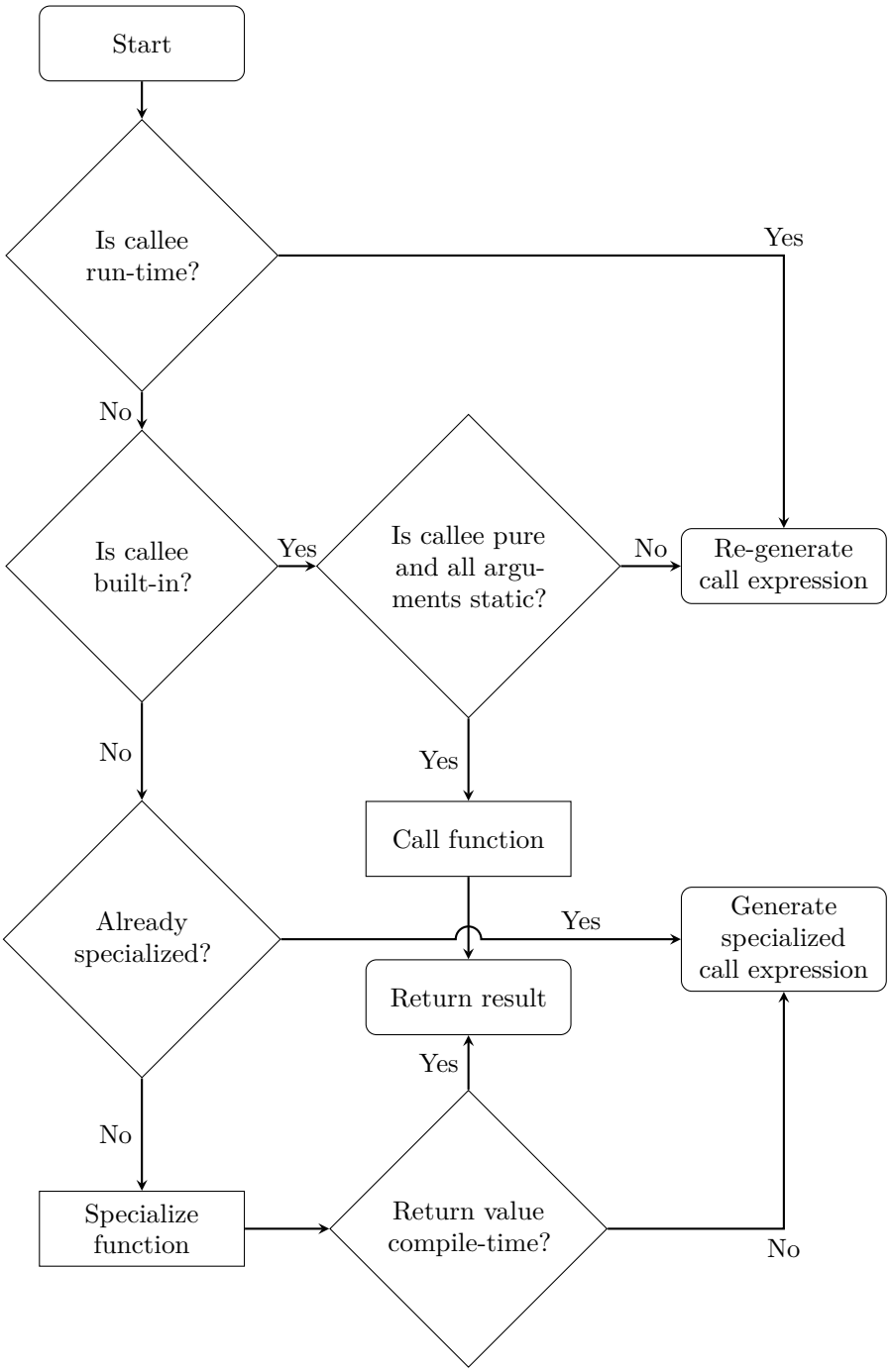


Figure 3: Partially evaluating a call expression

Listing 28: Reusing specialized functions

```
function add(a, b){
    return a + b;
}
console.log('2 + 2', add(2, 2));
console.log('X + 2',
    add($HPE.RT(X), 2));
console.log('Y + 2',
    add($HPE.RT(Y), 2));
```

```
function add$0(a){
    return a + 2;
}
console.log('2 + 2', 4);
console.log('X + 2',
    add$0(X));
console.log('Y + 2',
    add$0(Y));
```

Listing 29: Loop unrolling & `continue`

```
let arr = $HPE.CT(['a', 'b',
    'c']);
for(let i = $HPE.CT(0);
        i < arr.length; i++){
    if(arr[i] === 'b'){
        continue;
    }
    console.log('item', i,
        arr[i]);
}
```

```


{ // loop unrolled
    console.log('item', 0, 'a');
    // 1, 'b' is skipped by
    // continue
    console.log('item', 2, 'c');
}
```

compile-time to run-time. This flag is also set when a `return` statement is evaluated with a residualizable static value while there are residual statements preceding the `return`. When the evaluation of a function ends without the flag set, it means the function was completely evaluated, and the call is eliminated from the residual code and replaced with the return value.

For loops, a similar flag is used when a `break` or `continue` statement is residualized. In this case, the flag is checked after evaluating the loop body and if set, the partial evaluator throws an error. This is because the loop body is only partially evaluated when the loop conditional is compile-time, and thus the evaluator expects the body not to be interrupted at run-time. Algorithms 6 to 8 describe these semantics.

5.7 Classes

Partial evaluation of classes starts off in a similar way as that of function calls. The partial evaluator will first query the registry if the class to be evaluated is indeed a user-defined one, and if it hasn't been evaluated for the given static arguments before. Otherwise, the class' `constructor` method is evaluated first.

Ordinary objects in the partial evaluator must either be completely static, or completely dynamic. However, to facilitate specializing a class, we must allow some assignments to the `this` variable inside the constructor to be compile-time and others run-time. To enable this, we mark the `this` object as partial, allowing mixed compile-time and run-time assignments to its properties, but only for the duration of evaluating the constructor call. After the constructor is evaluated, the partial evaluator checks whether the `this` object has any properties with

Listing 30: Specialized class

```
class Adder {
   constructor(a, b){
      this.a = a;
      this.b = b;
   }
   add(){
      return this.a + this.b;
   }
}
const a = new Adder($HPE.CT(2),
    $HPE.CT(2));
console.log('2 + 2', a.add());
const b = new Adder($HPE.CT(2),
    $HPE.RT(X));
console.log('2 + X', b.add());
```

```
class Adder$0 {
  constructor(b) {

    this.b = b;
  }
  add() {
    return 2 + this.b;
  }
}


console.log("2 + 2", 4);
const b = new Adder$0(X);

console.log("2 + X", b.add());
```

dynamic values. If not, the resulting object is fully compile-time and returned as the result of the **new** call, after being stripped of its partial status.

When the constructor returns a **this** object with run-time properties, the other methods defined on the class are evaluated. Those methods are called with the partial **this** value, but are not allowed to change any compile-time properties on it, though they can of course read static properties. The methods are specialized only on the **this** value, any other parameters they may have are marked as run-time. After the methods are specialized, they are added to the residual class, along with the constructor, and the original **new** call is replaced with one to the specialized class. The semantics of this are shown in algorithms 16 to 18.

Method calls on compile-time objects may also be residualized. In this case, the method is partially evaluated as a regular function, with the **this** argument set to the object. The specialized function is added to the residual program like other residual functions are, rather than added to a residual class as a static method, since JavaScript doesn't require methods to be defined on a class.

5.8 Semantics

In this section we list the various algorithms that we referred to earlier, along with some more detail on how they work. Where the algorithms return an expression in angle brackets: $\langle \ldots \rangle$, the expression is a piece of residual code in a *partial completion*, otherwise the return value is compile-time.

The LITERALIZE function in algorithm 1 receives a completion record that may be *partial* or *normal*, and in the latter case tries to turn it into a *partial completion* with the residual code for the value of that completion. If the value is a compile-time object, the algorithm throws an error since compile-time objects may not be residualized.

Residual evaluation, as mentioned in section 5.3, is shown in algorithm 2. The markResidual and unmarkResidual functions set a flag in the partial AR that is read by the inResidual function used in algorithms 6 and 9.

Algorithm 1 Residualizing a compile-time value

```
function LITERALIZE(v)
    if isPartial(v) then
        return ⟨ v ⟩
    end if
    switch valueType(v)
        boolean, number, string, null:
            return literal expression with v
        undefined:
            return identifier expression with 'undefined'
        object or function:
            throw static object became residual
    end switch
end function
```

Algorithm 2 Residual evaluation

```
function REVAL(a, penv)
    markResidual(penv)
    r ← PEVAL(a, penv)
    unmarkResidual(penv)
    return r
end function
```

Evaluating binary expressions, shown in algorithm 3, is fairly straightforward: if both sub-expressions result in compile-time values, they are combined according to the binary operator, otherwise the result is a *partial completion* with the results of the sub-expressions residualized.

Algorithm 4 shows how identifier expressions are evaluated. As explained in section 5.2, if the value of the variable being referenced is compile-time, that value is returned, otherwise the identifier itself in a *partial completion*

Property access is a bit more complicated: in algorithm 5, we first evaluate the object and property key being read. For run-time objects, the property key is residualized and returned in a residual expression with the original object expression. When the value of a property on a compile-time object is requested, we need to make sure the property key is static, otherwise the partial evaluator won't know what property to read. If that is not the case, an error is thrown.

When both the object and property key can be computed, the value is read. This may still result in a *partial completion*, when the object is a *partial* `this` value in a constructor and the property is marked run-time. When the property value is also compile-time it is returned, but not before setting the lastBaseObject. This value is used when this property access is part of a call expression, in order to pass along the `this` value, which is the object.

The semantics for evaluating a sequence of statements, in algorithm 6, deal with the various completion types produced by other evaluation methods. A *normal completion* is the easiest: just continue with the next statement.

When a statement is residualized and returned as a *partial completion*, the

Algorithm 3 Partial evalution of expressions

```
function PEVAL(e1 op e2, penv)
    r1 ← PEVAL(e1, penv)
    r2 ← PEVAL(e2, penv)
    if isConcrete(r1) ∧ isConcrete(r2) then
        return op(r1, r2)
    else
        v1 ← LITERALIZE(r1)
        v2 ← LITERALIZE(r2)
        return ⟨ v1 op v2 ⟩
    end if
end function
```

Algorithm 4 Partial evalution of identifier expressions

```
function PEVAL(id, penv)
    v ← getValue(id, penv)
    if isAbstract(v) then
        return ⟨ id ⟩
    else
        return v
    end if
end function
```

evaluator sets the markResidualReturn flag to indicate that any `return` statements after this need to be residualized as well.

This flag is also set if the evaluator handles a *return completion*. In the case of a *return completion*, when the aforementioned flag is set or the partial AR is in residual mode, the `return` statement is residualized (and the flag is also set), otherwise it returns the value in the completion and does not evaluate the next statements. If the statement is residualized, but the return value is a compile-time object, the LITERALIZE function will throw an error.

Finally, the *break* and *continue completion*s set the markLoopInterrupt flag, if the statements were evaluated in residual mode. This flag is checked by algorithm 8 to detect when a `break` or `continue` statement is residualized in a loop that is supposed to be unrolled at compile-time.

For `if` statements, the evaluator first evaluates the conditional expression, as shown in algorithm 7. If its value is static then either the consequent or alternate branch of the statement is further evaluated. When the test expression is compile-time, both branches are evaluated, but in residual mode. This results in two *partial completions* which are combined into a residual `if` statement.

The handling of `while` statements, which can be found in algorithm 8, is similar when the conditional is a run-time expression: in that case the body of the loop is evaluated in residual mode and combined with the test expression into a residual `while` statement.

When the condition of the loop is compile-time, the loop is unrolled. As long as the condition evaluates to a truthy value, the body of the loop is partially

Algorithm 5 Partial evaluation of object property access

```
function PEVAL(obj[prop], penv)
    o ← PEVAL(obj, penv)
    p ← PEVAL(prop, penv)
    if isConcrete(o) then
        if isPartial(p) then
            throw inconsistent access
        else
            v ← o[p]
            if isPartial(v) then
                return ⟨ v ⟩
            else
                lastBaseObject(penv) ← o
                return v
            end if
        end if
    else
        p_r ← LITERALIZE(p)
        return ⟨ o[p_r] ⟩
    end if
end function
```

evaluated and any *partial completion*s from that evaluation are concatenated into the residual code. If the partial evaluation of the loop body results in a residual `break` or `continue` statement, the loopInterrupted flag will be set. This is detected here, and an error is raised in that case.

Variable assignments, described in algorithm 9, is where the semantics start to get more complicated, since this is where a lot of the HPE techniques come together. These semantics apply to assignment statements, as well as variable declaration statements.

A variable can either be uninitialized, compile-time, or run-time. This is checked by isConcrete(id) and isAbstract(id) respectively. If the variable is already marked as run-time, the value being assigned is residualized by LITERALIZE, and the assignment expression/statement along with it.

For compile-time variables, we check if the new value being assigned is static as well, and if the partial AR in which the variable is declared is not in residual mode. When the latter case is true, it means the code is trying to assign to a variable within code that is being residualized: moving the variable from compile-time to run-time. In either case, we throw an error.

When the value being set is compile-time as well, we update the partial AR with the new value. If that value is an object, we also add the partial AR as an owner of that object, which is needed to deal with object modifications properly.

Finally, if the variable has not been assigned a value before, it becomes compile-time or run-time depending on the status of the initial value. When that value is static and an object we also add the owner just like the previous case.

Algorithm 6 Partial evalution of statement list

```
function PEVAL(s1; s2, penv)
    r1 ← PEVAL(s1, penv)
    switch completionType(r1)
        concrete: return PEVAL(s2, penv)
        partial:
            markResidualReturn(penv)      ▷ Subsequent returns will be residualized
            return ⟨ r1; PEVAL(s2, penv) ⟩

        return:
            if inResidual(penv) ∨ hasResidualReturn(penv) then
                markResidualReturn(penv)
                return ⟨ LITERALIZE(r1) ⟩
            else
                return r1
            end if

        break or continue:
            if inResidual(penv) then
                markLoopInterrupt(penv)
                return ⟨ LITERALIZE(r1) ⟩
            else
                return r1
            end if

    end switch
end function
```

Algorithm 10 deals with object property assignments. When the object expression itself results in a *partial completion*, the assignment expression is residualized, similar to variable assignments.

If the object is static, but the expression of property key cannot be computed at compile-time, the evaluator raises an inconsistent modification error.

Finally, when the value being assigned is run-time, we need to raise an error as well, unless the object is marked as *partial*. This is done by the SPECIALIZECLASS function in algorithm 18, to allow mixed run-time and compile-time fields in a class constructor. When none of the previous checks raise an error, the object is passed on to the UPDATEOBJECT function of algorithm 11.

The UPDATEOBJECT function in algorithm 11 first checks if a modification of the object is allowed by its owners. That is, if none of the partial ARs from which the object is reachable is in residual mode. If one of them is, an error is raised.

Otherwise, the object itself is updated with the new property value, which may be run-time, in which case the assignment expression is also residualized. If, however, the new value is compile-time, and also an object, its owners are updated with the object that is being modified.

Algorithm 7 Partial evaluation of `if` statements

```
function PEVAL(if(test) consequent else alternate, penv)
    t ← PEVAL(test, penv)
    if isConcrete(t) then
        if isTruthy(t) then
            return PEVAL(consequent, penv)
        else
            return PEVAL(alternate, penv)
        end if
    else
        c_r ← REVAL(consequent, penv)
        a_r ← REVAL(alternate, penv)
        return ⟨ if(t) c_r else a_r ⟩
    end if
end function
```

Algorithm 8 Partial evaluation of `while` statements

```
function PEVAL(while(test) body, penv)
    t ← PEVAL(test, penv)
    if isPartial(t) then
        b_r ← REVAL(body, penv)
        return ⟨ while(t) b_r ⟩
    else if isTruthy(t) then
        b ← PEVAL(body, penv)
        if loopInterrupted(penv) then
            throw loop interrupted
        end if
        r ← PEVAL(while(test) body, penv)
        return ⟨ b ; r ⟩
    else
        return ⟨  ⟩
    end if
end function
```

Algorithm 12 shows the three forms of call expressions: a standard function call, a method call on an object, and a `new` call. In all cases, we start by evaluating the callee and arguments, the latter of which is a list of possibly mixed compile-time and run-time completions. For method calls, this is also where we read the lastBaseObject, set by algorithm 5. If the callee itself is a run-time expression, the call expression is re-generated with the partially evaluated arguments. Otherwise, the call is handled by the CALL or NEW functions in algorithms 13 and 16 respectively. An overview of this mechanism can also be found in figure 3.

The CALL function in algorithm 13 receives the callee, a function, the `this` value, an object or `undefined`, the evaluated arguments, an array of *partial* and/or *normal completion*s, and the partial AR of the call expression.

If the callee is a built-in function, it checks if all arguments are compile-time values, and if so, calls the function and returns the compile-time result. If not,

Algorithm 9 Partial evalution of variable assignments

```
function PEVAL(id = exp, penv)
    ownerEnv ← resolve(id)
    e ← PEVAL(exp, penv)
    if isAbstract(id) then
        e_r ← LITERALIZE(e)
        return ⟨ id = e_r ⟩
    else if isConcrete(id) then
        if isPartial(e) ∨ inResidual(ownerEnv) then
            throw abstract value assigned to concrete variable
        else
            if isObject(e) then
                update(owners(e) ← ownerEnv)        ▷ Add ownerEnv as owner of e
            end if
            setValue(id ← e, ownerEnv)
        end if
    else                                                    ▷ id is uninitialized
        setValue(id ← e, ownerEnv)
                                  ▷ id becomes abstract or concrete, depending on e
        if isConcrete(e) then
            if isObject(e) then
                update(owners(e) ← ownerEnv)        ▷ Add ownerEnv as owner of e
            end if
            return e
        else
            return ⟨ id = e ⟩
        end if
    end if
end function
```

the call expression is residualized.

When the callee is a user-defined function, it is passed on to SPECIALIZECALL, which can return either a compile-time value or a residual call expression.

SPECIALIZECALL, found in algorithm 14, first looks in the function Registry, mentioned in section 5.5, which has a table of user-defined functions and the compile-time arguments they have been specialized for. If there is already an entry with the same callee and static arguments in that table, the function has already been specialized, and we can call the specialized version with just the run-time arguments.

Otherwise, we look up the original function definition and pass it to SPECIALIZEFUNCTION. It will return either a compile-time value, in which case the call expression is evaluated away entirely, or a new function definition. This definition is added to the table of specialized functions, and its name is used to generate a new residual call expression to it.

Finally, SPECIALIZEFUNCTION in algorithm 15, takes care of actually specializing a user-defined function. It receives the function definition and the `this`

Algorithm 10 Partial evaluation of object property assignments

```
function PEVAL(obj[prop] = exp, penv)
    o ← PEVAL(obj, penv)
    p ← PEVAL(prop, penv)
    e ← PEVAL(exp, penv)
    if isPartial(o) then
        e_r ← LITERALIZE(e)
        p_r ← LITERALIZE(p)
        return ⟨ o[p_r] = e_r ⟩
    else if isPartial(p) then          ▷ Object is concrete, property key is not
        throw inconsistent object modification
    else if isPartial(e) then          ▷ Object & property key concrete, value is not
        if ¬ allowResidualModification(o) then
            throw inconsistent object modification
        end if
    end if
                    ▷ Object, property key, and value concrete, or modification allowed
    return UPDATEOBJECT(o, p, e)
end function
```

and arguments to specialize it for. The algorithm first makes a new partial AR based on the Scope of the function and the partial AR in which it was defined. It then sets the `this` value in that environment, and binds the parameters to the given arguments. These parameters become compile-time or run-time based on the status of their corresponding arguments. Finally, the function body is evaluated as a sequence of statements, which can result in a residual block of code or a compile-time return value.

The NEW and SPECIALIZENEW functions in algorithms 16 and 17 are almost identical to CALL and SPECIALIZECALL in algorithms 13 and 14, except that they end up in at SPECIALIZECLASS in algorithm18 when a user-defined class needs to be specialized.

The last algorithm, SPECIALIZECLASS in 18, takes care of specializing a user-defined class. It receives the class definition and the constructor arguments to specialize for. First, it makes a new object based on the `prototype` of the class. This object is marked as *partial*, allowing mixed compile-time and run-time properties. Then, the `constructor` method of the class is specialized with the *partial* object and the given constructor arguments. Afterwards, the object is unmarked as *partial*.

If specializing the constructor method resulted in a compile-time return value, and all properties in the new `this` object are also compile-time, the `this` object is returned as a compile-time value, and the `new` call is evaluated away.

When neither of the abovementioned conditions are true, the other methods of the class are specialized. They only receive the `this` object as compile-time value, their other parameters are all marked as run-time. Finally, the specialized constructor and methods are combined into a specialized class definition. This definition is registered (by algorithm 17), and the original `new` expression is

Algorithm 11 Updating a concrete object

```
function UPDATEOBJECT(o, p, e)
    for all w ∈ ownersOf(o) do
        if ¬ allowAssignment(w) then
            throw inconsistent object modification
        end if
    end for
    update(o[p] ← e)                    ▷ Allow run-time property values in partial objects.
    if isPartial(e) then
        e_r ← LITERALIZE(e)
        return ⟨ o[p] = e_r ⟩
    else
        if isObject(e) then
            update(owners(e) ← o)                    ▷ Add o as owner of e
        end if
        return e
    end if
end function
```

residualized as a call to the specialized version.

Algorithm 12 Partial evaluation of call expressions

```
function PEVAL(callee(arguments), penv)
    c ← PEVAL(callee, penv)
    A ← map(arguments, PEVAL(·, penv))
    if isPartial(c) then
        return ⟨ c(A) ⟩
    else
        return CALL(c, undefined, A, penv)
    end if
end function

function PEVAL(obj.method(arguments), penv)
    lastBaseObject(penv) ← undefined
    c ← PEVAL(obj.method, penv)
    o ← lastBaseObject(penv)
    A ← map(arguments, PEVAL(·, penv))
    if isPartial(c) then
        return ⟨ c(A) ⟩
    else
        return CALL(c, o, A, penv)
    end if
end function

function PEVAL(new Class(arguments), penv)
    c ← PEVAL(Class, penv)
    A ← map(arguments, PEVAL(·, penv))
    if isPartial(c) then
        return ⟨ new c(A) ⟩
    else
        return NEW(c, A, penv)
    end if
end function
```

Algorithm 13 Partial evalution of function calls

```
function CALL(c, t, A, penv)
    if isBuiltIn(c) then
        if ∀a ∈ A : isConcrete(a) then
            return call(c, t, A)
        else
            return ⟨ c(A) ⟩
        end if
    else
        c_r ← SPECIALIZECALL(c, t, A)
        if isConcrete(c_r) then
            return c_r
        else
            return ⟨ c_r ⟩                    ▷ Residual call from SPECIALIZECALL
        end if
    end if
end function
```

Algorithm 14 Specializing function calls

```
function SPECIALIZECALL(callee, t, A)
    m ← findMemoCall(callee, t, A)
    A_r ← filter(A, isPartial(·))
    if m then                    ▷ Already specialized for the given concrete arguments
        c_r ← name(m)
        return ⟨ c_r(A_r) ⟩
    else
        f(P) ← findDefinition(callee)
        r ← SPECIALIZEFUNCTION(f(P), t, A)
        if isConcrete(r) then
            return r                                  ▷ Call is evaluated away
        else
            registerDefinition(r, A)
            c_r ← name(r)
            return ⟨ c_r(A_r) ⟩
        end if
    end if
end function
```

Algorithm 15 Specializing functions

```
function SPECIALIZEFUNCTION(f(P), t, A)
    upenv ← environmentOf(f)
    penv ← newEnvironment(scopeOf(f), upenv)
    thisOf(penv) ← t
    for all p_i ∈ P, a_i ∈ A do
        PEVAL(p_i = a_i, penv)
                ▷ Set p_i static or dynamic, depending on its corresponding argument
    end for
    return PEVAL(bodyOf(f), penv)
                                ▷ Can be a residual function, or concrete result value
end function
```

Algorithm 16 Partial evaluation of `new` calls

```
function NEW(c, A)
    if isBuiltIn(c) then
        if ∀a ∈ A : isConcrete(a) then
            return construct(c, A)
        else
            return ⟨ new c(A) ⟩
        end if
    else
        c_r ← SPECIALIZENEW(c, A)
        if isConcrete(c_r) then
            return c_r
        else
            return ⟨ c_r ⟩                    ▷ Residual new from SPECIALIZENEW
        end if
    end if
end function
```

Algorithm 17 Specializing `new` calls

```
function specializeNew(c, A)
    m ← findMemoClass(c, A)
    A_r ← filter(A, isPartial(·))
    if m then
        c_r ← name(m)
        return ⟨ c_r(A_r) ⟩
    else
        C(P) ← findDefinition(c)
        r ← specializeClass(C(P), A)
        if isConcrete(r) then
            return r
        else
            registerDefinition(r)
            C_r ← name(r)
            return ⟨ new C_r(A_r) ⟩
        end if
    end if
end function
```

Algorithm 18 Partial evaluation of constructors

```
function specializeClass(c, A)
    C ← constructorMethod(c)
    t ← makeObject(c.prototype)
    markPartialObject(t)
                ▷ Allow this to receive run-time properties during the constructor call
    C_r ← specializeFunction(C, t, A)
    unmarkPartialObject(t)
    if isConcrete(C_r) then
        return t                                  ▷ Completely concrete object
    end if
    R ← {C_r}
    for all m(P_m) ← methodsOf(c) do
        if m = C then
            skip                                  ▷ Constructor already done
        end if
        A_m ← map(P_m, ⊤)
                        ▷ Specialize other methods with their parameters run-time
        m_r ← specializeFunction(m, t, A_m)
        R ← {R; m_r}
    end for
    return ⟨ class R ⟩
end function
```

	# of tests	% of previous
Total	36759	100%
Language section	19961	54%
Relevant to Alternator	7465	37%
Failures	318	4%

Table 1: Test results

6 Testing

We cannot, at this time, provide a formal proof of correctness of either Alternator or HandCrank. However, there is a comprehensive test suite for validating JavaScript interpreters, called Test262 [4], provided by the ECMAScript standards committee itself.

The Test262 set is meant to test a complete JavaScript interpreter, including non-strict mode and web compatibility mode. Since those are not implemented in our meta-circular interpreter, along with several other features, we skip the tests involving those in our validation. There are 19961 tests in the 'language' section of the test set, which is where the language features itself are tested, and not for example, the behaviour of built-in functions. Of those, 7465 remain after filtering out the behaviours intentionally not implemented. Alternator fails on 318 tests. These numbers are also broken down in table 1.

Some failures are due to the test case changing global objects that the meta-circular interpreter shares with interpreted code, breaking the behaviour of the interpreter. Other cases test the return value of statements, which is only observable via `eval` and not implemented completely as there is almost no use for them in modern code. In a similar vein, there are test cases for assigning a name to an anonymous function from a function expression, which depends on how the function expression appears in the right-hand side of the assignment: another behaviour that modern code is not expected to make use of. Finally, there are some tests that fail due to bugs in Alternator. However, they are not impactful enough to prevent execution of most well-formed programs. All-in-all, Alternator successfully passes 96% of the relevant test cases.

For the partial evaluator, our primary means of verifying its functionality are the benchmark programs in section 7. We check the correctness of the partially evaluated code by comparing its output to the non-specialized versions of the benchmarks. In addition, we use a number of smaller programs with special annotations to verify that the partial evaluator throws errors in the cases where the programmer uses concrete and abstract values in inconsistent ways, as explained in the sections 3.1 and 3.2.

7 Benchmarking

To test if Hybrid Partial Evaluation does indeed result in program speed-up, we benchmarked three example programs: matrix multiplication (mulmat), fast Fourier transform (fft), and state machine (fsm). These benchmarks were run on three different machines, one running Windows on an Intel processor, one running Linux on an Intel processor, and one running Linux on an ARM processor. We ran the benchmarks in Node.js on all three machines, and found that they all had the same performance trends: they have different actual running times, but the same relative speed-ups/slow-downs. On only one machine, Linux on Intel, we also have access to a standalone version of the Spidermonkey engine, called JS78, and found it has different performance characteristics to the V8 engine of Node.js. We shall, therefore, compare the results of our three main benchmarks running in Node.js and JS78 on that one machine.

For each benchmark we defined a number of cases to specialize on, e.g. matrix size for matrix multiplication. We then ran both the original version and the partially evaluated version in a script for a number of iterations. This script measures the total time it takes to run those iterations in the original and in the specialized version. Finally, that script was run 7 times as normal and 7 times with Just-In-Time compilation (JIT) disabled. We present the median of those run-times in the selected graphs below.

7.1 Just-In-Time Compilation

A common technique for optimizing interpreted programming language is Just-In-Time compilation (JIT) [1]. While the source code for a language like C is compiled ahead of time into machine instructions to be executed by the processor directly, an interpreter for a language like JavaScript parses the source code and performs the behaviour itself. This interpretation step usually incurs a performance penalty at run-time that compiled languages pay for at compile time. A JIT interpreter does both: it starts off interpreting the program as a regular interpreter would, but at the same time tries to identify sections of the program that are executed often. These sections are then compiled to machine code and executed that way. Since the compilation itself also has a cost, now at runtime, it is only beneficial for relatively small but frequently executed code. Nevertheless, for those sections it can provide great speed-up.

Most modern JavaScript engines use JIT, including V8 and Spidermonkey, but the standalone interpreters we use to run the benchmarks provide options to disable it.

7.2 Matrix Multiplication

The matrix multiplication benchmark measures the performance of the function in listing 31. The partially evaluated versions are specialized to the matrix size and a static matrix A. An example of a specialized version can be found in listing 35 in the appendix. For the benchmark we specialized to matrix sizes ranging from 2×2 to 16×16. Both matrices being multiplied are the same each iteration, with the first being static and the second marked as dynamic. To prevent the interpreter from simply caching the result, the `x` parameter is

Listing 31: Matrix multiplication

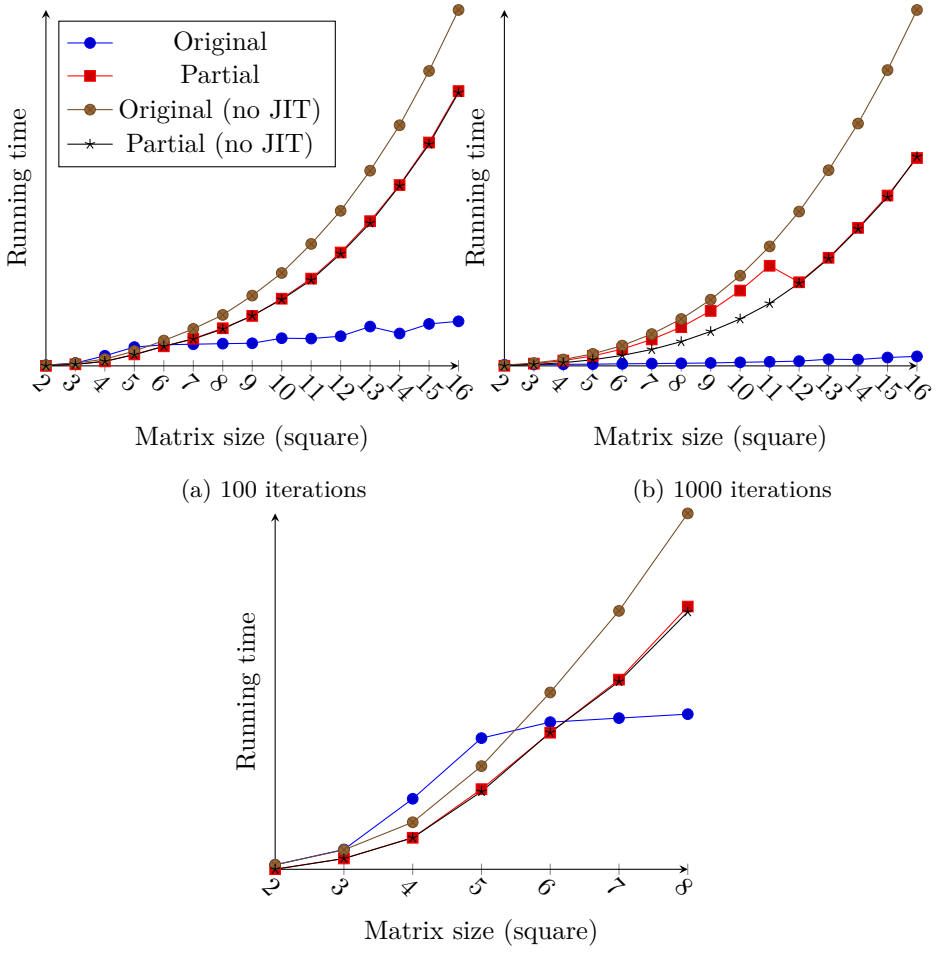
```
// multiply matrix A * B, where a has ac columns and b has bc columns,
    each
// cell in the resulting matrix is multiplied by x
function mul(A, B, ac, bc, x){
   const ar = A.length / ac,
      len = ar * bc,
      ret = $HPE.RT(new Array(len));

   ret.fill(0);

   for(let i = $HPE.CT(0, ar); i < ar; i++){
      for(let j = $HPE.CT(0, bc); j < bc; j++){
         for(let k = $HPE.CT(0, ar); k < ar; k++){
            ret[i * bc + j] += A[i * ac + k] * B[k * bc + j];
         }
         ret[i * bc + j] *= x;
      }
   }

   return ret;
}
```

different each iteration, resulting in a different output matrix. The output of the partially evaluated code is verified by comparing the resulting matrix against the one produced by the original code.

Figure 4 shows the performance trends of the matrix multiplication benchmark running on Node.js. The actual running times are elided from the graph, since we are only interested in the relative performance of the specialized code versus the original. Overall, we can see that for increasing matrix sizes, and/or increasing iteration counts, the original code performs better than the specialized versions, when the JIT is enabled, but worse without the JIT.

However, as highlighted in figure 4c, with only 100 iterations, the specialized versions for matrix sizes up to 7×7 are slightly faster than the original code. With the small iteration count, the specialized function performs the same with or without JIT, but is faster than the original code without JIT. When we run Node.js with the `--trace-opt` argument, it indicates that for the matrix sizes 2×2 and 3×3 at 100 iterations, it does not try to optimize either the original or the specialized function. Only at the larger matrix sizes does it compile the non-specialized function to native code, but up to 7×7 matrices, this compilation takes longer than the specialized function needs to run under regular interpretation. The engine does not optimize the specialized function for any matrix size, at 100 iterations, which is why the running times for partial and partial without JIT are the same.

For the 1000 iterations run, the specialized versions for matrices up to 11×11 are slower with JIT than without, and after that those running times are the same. Tracing the optimizer again tells us that up to the 11×11 matrix case,

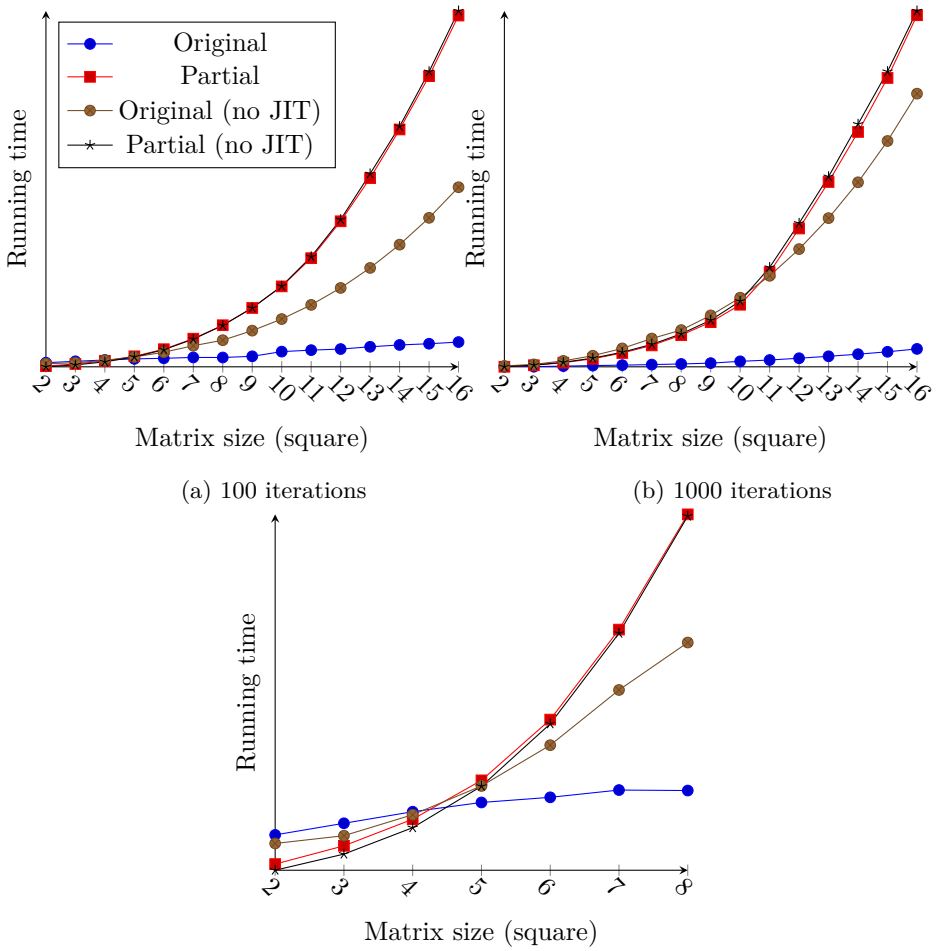


(a) 100 iterations

(b) 1000 iterations

(c) 100 iterations, zoomed in

Figure 4: Matrix multiplication with Node.js



(a) 100 iterations

(b) 1000 iterations

(c) 100 iterations, zoomed in

Figure 5: Matrix multiplication with JS78

the engine attempts to optimize the specialized function, but in later cases it stops trying with the reason that the function is too big.

The running time trends for JS78, shown in figure 5, are quite different in this benchmark, to those of Node.js. The non-specialized function is still generally faster when executed with the JIT enabled, but the specialized version running with JIT is even slower than the original version without, for larger matrix sizes. Unfortunately, JS78 does not offer as extensive tracing tools as V8 does. But the obtained timings indicate that the increase in code size resulting from partial evaluation does more harm than good here.

7.3 Fast Fourier Transform

The Fast Fourier transform (FFT) benchmark is based on the code described in [2], translated to JavaScript, which can be found in listing 32. We specialized the algorithm only for its radix size, unrolling the outer loop of the `perform`

Listing 32: FFT

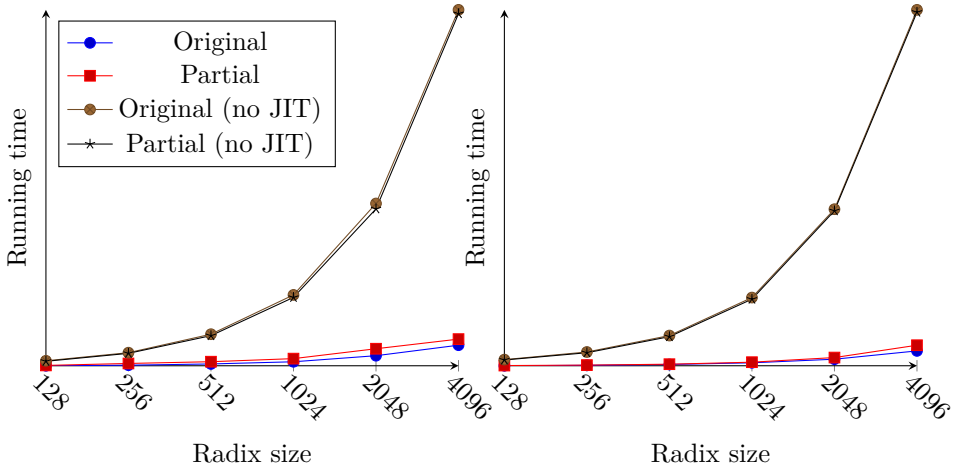
```
function perform(data, N){
   for(let step = $HPE.CT(1, N); step < N; step <<= 1){
      const jump = step << 1,
         delta = Math.PI / step,
         sine = Math.sin(delta / 2),
         mult = [-2 * sine * sine, Math.sin(delta)];
      let fact = $HPE.RT([1, 0]);
      for(let group = $HPE.RT(0); group < step; group++){
         for(let pair = group; pair < N; pair += jump){
            const match = pair + step,
               prod = mulComplex(fact, data[match]);
            data[match] = subComplex(data[pair], prod);
            data[pair] = addComplex(data[pair], prod);
         }
         fact = addComplex(mulComplex(mult, fact), fact);
      }
   }
   return data;
}
```

function, and benchmarked radix sizes $128, 256, ..., 4096$. The code for the `perform` function specialized to a radix of 128 can be found in listing 36 in the appendix. For each iteration, the benchmark generates a sine-wave signal with a different frequency, computes the FFT via the original or specialized function, and finds the maximum value in the frequency domain. The partially evaluated code is verified by checking if this maximum value is the same as that computed by the original function.

The running times for the FFT benchmark are even less flattering: for all cases, on both tested engines, the specialized functions perform no better and often worse than the original one. This is the same for 100 and 1000 iterations, and JS78 exhibits this behaviour as well. The most likely reason for this is that unrolling the outer loop in `perform` is just not a big performance gain, as the inner loops are much 'hotter'. However, if we were to unroll those as well, the code grows exponentially, and the JIT wouldn't touch it. As the benchmark runs without JIT show both specialized and original versions as equally slow, unrolling additional loops will not have any benefit.

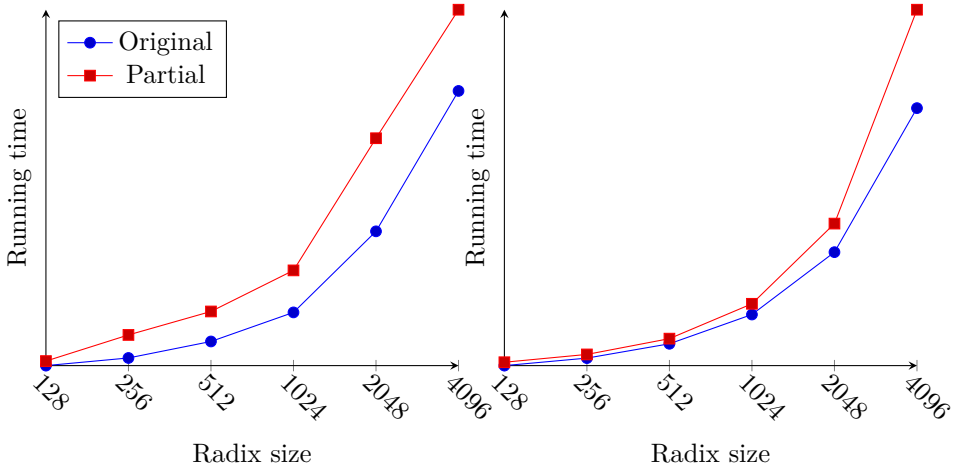
7.4 Finite State Machine

For the last benchmark, the Finite state machine (FSM), we created two variants: a small state machine, and a larger one. The FSM represents a door that can be in one of the states *closed*, *opened*, or *locked*, and transition between these states via the *open*, *close*, *lock*, and *unlock* events. The larger FSM has 16 states and 24 transitions, with between 1 and 3 outgoing transitions per state. Listing 33 contains the definition of the small FSM (created by `makeSmallFSM`), and the FSM evaluator: `function fsm`. This function processes a sequence of transition events and recursively traverses the FSM to build up a list of the



(a) 100 iterations

(b) 1000 iterations



(c) 100 iterations, JIT only

(d) 1000 iterations, JIT only

Figure 6: FFT on Node.js

Benchmark	Specialization	LoC in	LoC out	Best performance
MulMat	Size, left matrix	18	31 – 4903	195%
FFT	Size, outer loop	19	68 – 113	77%
FSM (small)	State machine	12	33	168%
FSM (large)	State machine	12	184	160%

Table 2: Summary of benchmark results

states that those transitions lead to. The partially evaluated version eliminates the definition of the state machine itself, and specializes the `fsm` function for each state individually. The residual code for the specialized FSM is in listing 37, in the appendix.

The finite state machine benchmark paints a better picture, in figure 7: as we increase the number of events or iterations, the specialized versions perform better than the non-specialized one. This holds when we run with JIT and without. Interestingly, the large FSM shows exactly the same trend, but this may just indicate that the large FSM isn't significantly larger. One caveat that should be mentioned is that, due to the recursive nature of the FSM evaluator, the total number of events it can process is limited by the stack size available to the JavaScript code. When trying to process 8192 events or more, the code crashed with a stack overflow error. Though it is possible to increase the stack size for Node.js, this is not the case for JS78 and not something a script running in a web browser could do either. Nevertheless, this benchmark shows that Hybrid Partial Evaluation can indeed produce a performance increase that is not limited to small iteration counts.

7.5 Summary

Table 2 shows a summary of our benchmark results. The 'Best performance' column shows the ratio of $\frac{T_{orig}}{T_{spec}}$ of the best case measured for that benchmark. A value greater than 100% means the specialized version is faster than the original. Of course, these percentages are not representative of the entire benchmark, but show what *can* be achieved in the ideal case.

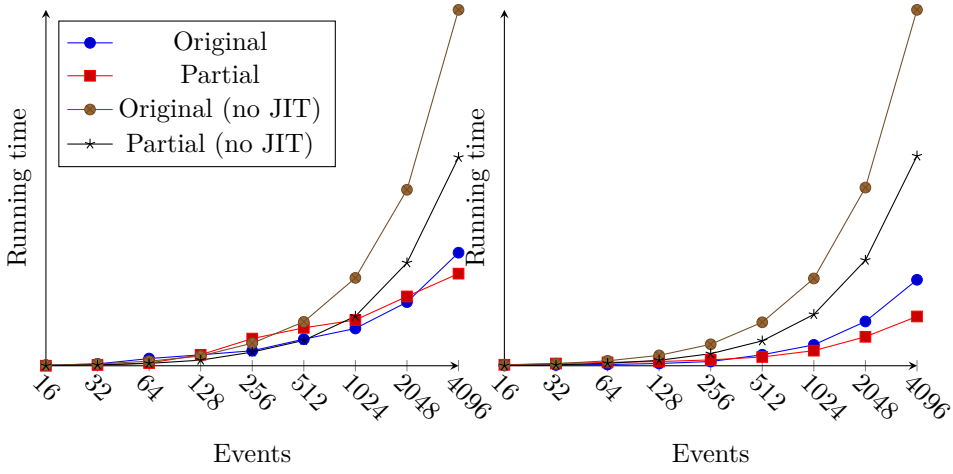
As mentioned, the Matrix Multiplication showed some speedup, but only for small matrices. In those cases the JIT takes longer to optimize the code than the interpreter takes to run it, and there the specialized version makes it run faster. However, as the size of the input grows, the original program does perform better.

In the Fast Fourier Transform benchmark we found no speedup whatsoever. In the table, the best performance shows the relative performance of the least worst result.

Finally, the Finite State Machine benchmarks do show improvement from HPE. And what's more, their performance appears to improve as the input grows, meaning that HPE may provide reliable optimization for this program.

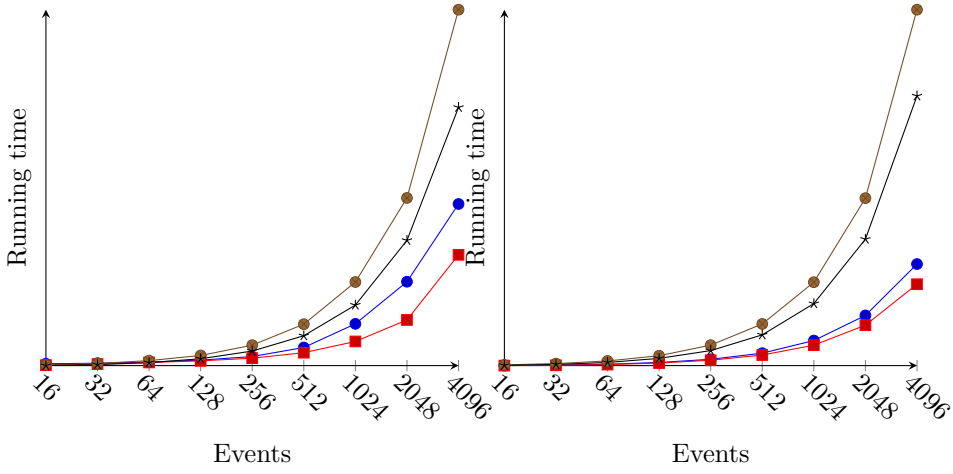
Listing 33: Small state machine and evaluator

```
class State {
   constructor(name){
      this.name = name;
      this.outs = $HPE.CT([], name);
   }
}

class Transition {
   constructor(event, to){
      this.event = event;
      this.to = to;
   }
}

function makeSmallFSM(ct){
   const stOpened = new State($HPE.CT('opened', ct)),
      stClosed = new State($HPE.CT('closed', ct)),
      stLocked = new State($HPE.CT('locked', ct)),
      trClose = new Transition($HPE.CT('close', ct), stClosed),
      trOpen = new Transition($HPE.CT('open', ct), stOpened),
      trLock = new Transition($HPE.CT('lock', ct), stLocked),
      trUnlock = new Transition($HPE.CT('unlock', ct), stClosed);

   stOpened.outs = [trClose];
   stClosed.outs = [trOpen, trLock];
   stLocked.outs = [trUnlock];

   return stClosed;
}

function fsm(current, ret, getEvent){
   ret.push(current.name);

   const event = getEvent();
   for(let i = $HPE.CT(0, current); i < current.outs.length; i++){
      const next = current.outs[i];
      if(event === next.event){
         return fsm(next.to, ret, getEvent);
      }
   }
   return ret;
}
```



(a) Node.js, 100 iterations

(b) Node.js, 1000 iterations



(c) JS78, 100 iterations

(d) JS78, 1000 iterations

Figure 7: Small FSM

8 Discussion and Future Work

While neither of our implementations, Alternator and HandCrank, are complete, we believe they can serve as inspiration for more production-ready programs. Some of the features not fully implemented in Alternator were left out to keep it simple, and can be added in a more complete implementation. However, other features are not so easy, if not downright impossible, to implement within the current architecture of the interpreter. Should one want to build a feature-complete meta-circular interpreter and hybrid partial evaluator for JavaScript, it may be advisable to use our implementation as an inspiration, but choose a different structural design. For a hybrid partial evaluator, the semantics described in this thesis should be useful for any future implementation.

In our benchmarking we found that, while it is possible to speed programs up with Hybrid Partial Evaluation, there are also situations where HPE actually slows things down. The biggest confounding factor here is the JIT. In places where the JIT can optimize non-partially evaluated code very well, partially evaluating it results in worse performance. On the other hand, for code where HPE can eliminate object instances from the code and replace them with simple primitive values, the resulting program may perform much better. It is imperative though, that anyone who wishes to use HPE to optimize their code, tests it and keeps testing it as JavaScript engines are updated.

We have only run a relatively small number of benchmarks. Enough to show that performance can be improved with HPE, but a greater variety of benchmarks may be able to paint a clearer picture where the strengths and weaknesses of Hybrid Partial Evaluation for JavaScript can be found.

An interesting contrast is the earlier research in HPE for Ruby [15], which did find more performance improvements for Ruby programs when they were specialized by HPE. There are a myriad of possible explanations for this, but one is that in the time between their research and ours, JIT compilers have seen a great deal of development and improvement. This development of JavaScript engines is still very much ongoing, and with each improvement of an engine, the results of our benchmarks should be re-evaluated. In future work, rather than treating HPE as a technique separate from the optimizing JIT compilers found in modern JavaScript engines, it may be fruitful to combine the two. Using the techniques from Hybrid Partial Evaluation inside the JIT may be able to give us the best of both worlds.

9 Conclusion

At the start of this research, we asked two questions:

- How can Hybrid Partial Evaluation be implemented for JavaScript?
- Does Hybrid Partial Evaluation of JavaScript result in measurable performance gains?

In section 5 we described an implementation of Hybrid Partial Evaluation for JavaScript. Although our partial evaluator does not implement all aspects of JavaScript, it can deal with the basics of the language, and serve as a groundwork for future projects in this area. There are however, inherent limitations due to the architecture of the partial evaluator that make implementing features like generator and async functions infeasible. Nevertheless, the semantics of Hybrid Partial Evaluation described in this thesis should be helpful for any future implementations. In particular, the methods for object tracking and handling of early `return` statements have not been described for Hybrid Partial Evaluation before.

The second question is addressed in section 7, where we show the results of testing our hybrid partial evaluator on three benchmark programs. In the first benchmark, matrix multiplication, we saw that the partially evaluated program does perform faster, but in a very narrow set of circumstances. The Fast Fourier Transform benchmark showed no improved performance after Hybrid Partial Evaluation, in fact, the specialized version performed worse in all tested cases. However, in the final benchmark, the finite state machine, we found that the specialized program performs better as the size of the input increases. In conclusion: while it depends on the program, and under which circumstances it is executed, Hybrid Partial Evaluation can produce measurable performance gains.

Glossary

abrupt completion A non-normal completion record: *throw*, *break*, *continue*, or *return*, that is only handled by a few specific Evaluator methods. 32

accessor property A property of an object that defines a getter, setter, or both. 18

AR Activation Record 29–34, 37

arrow function A function created by an arrow function expression, which cannot be called via `new` and inherits its `this` value from its lexical context. 14, 67

AST Abstract Syntax Tree 27–36, 38

class constructor A function created by a `class` declaration or class expression, which can only be called via the `new` operator. 16, 66, 67

closure A record of a function environment mapping the free variables in that function to the values defined in the scope surrounding it. 16

constructor function A function intented to be called via the `new` operator, which is not necessarily a class constructor. 14, 16

data property A property of an object that contains a value. 18

DSL Domain-Specific Language 1, 5

dynamic value Also called a runtime value. A value that cannot be computed by the partial evaluator and remains in the residual code. 22

Ecma International A standards organization for information and communication systems, formerly known as European Computer Manufacturers Association (ECMA). Responsible for managing the ECMAScript standard which forms the basis of JavaScript. 7

FFT Fast Fourier transform 58, 59

FSM Finite state machine 59, 61

Futamura projection The specialization of a function (by partial evaluation) for a given static parameter. For example the Futamura projection of $f(k, b) = k + b$ at $k = 2$ is $f_2(b) = 2 + b$. 22

variable hoisting Lifting a variable declaration to the top of its scope. 12

HPE Hybrid Partial Evaluation 1, 5, 44, 61, 64

inherited property A property of an object that is not an own property, but defined on one of the objects up its prototype chain. 19

iterable An object that implements the *iterable* protocol, which allows it to be iterated over via `for-of` loops. 21

JIT Just-In-Time compilation 55, 56, 58, 59, 61, 64

LoC Lines of code 61

normal completion A completion record indicating evaluation completed without issue. It contains a value that is the result of the evaluation. 32

NPM Node Package Manager 8

own property A property of an object that is defined on the object itself. 19, 66

partial application The creation of a specialized function of an existing function by setting one or more of its parameters to fixed values. 22

partial AR Partial Activation Record 36–38, 41, 43–46, 48

property A key/value or key/accessor pair member of an object. 18, 66

prototype chain The chain of objects formed by following the *prototype* property of an object recursively until reaching an object with no prototype. 19, 21, 66

regular function A function created by a `function` declaration or function expression which is not an arrow function or class constructor. 13

residual program The program code resulting from (hybrid) partial evaluation. 22

Shift AST The AST format produced by the Shift parser. 28, 34, *see* AST

simple value A JavaScript boolean, number, string, or the value `null` or `undefined`. A simple value is immutable, and can be translated between internal representation and code without issue. Contrast with objects, arrays, functions, and symbols. 36, 37

static value Also called a compile-time value. A value that is known or computed by the partial evaluator and is eventually eliminated from the residual code. 22

strict mode An evaluation mode of JavaScript where otherwise silently ignored mistakes are thrown as errors, poorly optimizable behaviours are changed, and the syntax is restricted to be more future proof. Strict mode is not a subset of JavaScript, but a variant, and is not backwards compatible with old style code. 7, 27

temporal dead zone The region of code between the start of a variable scope and its declaration, where it may not be accessed. Only applicable to `let`, `const`, and `class` declarations. 12, 13

W3C World Wide Web Consortium 7, *Glossary:* World Wide Web Consortium

World Wide Web Consortium The standards organization that manages the various protocols and APIs of the world wide web. 7, 68

References

[1] John Aycock. 'A Brief History of Just-in-Time'. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[2] Sergey Chernenko. *Fast Fourier Transform in C++*. URL: https://web.archive.org/web/20201123211905/http://www.librow.com/articles/article-10 (visited on 23/11/2020).

[3] Ecma International. *ECMA-262: ECMAScript Language Specification*. Ninth. June 2018. URL: https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262%209th%20edition%20June%202018.pdf.

[4] Ecma International, Technical Committee 39. *Test262: ECMAScript Test Suite*. URL: https://github.com/tc39/test262.

[5] *ESTree*. URL: https://github.com/estree/estree.

[6] Yoshihiko Futamura. 'Partial computation of programs'. In: *RIMS Symposia on Software Science and Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 1–35. ISBN: 978-3-540-39442-6.

[7] Paul Hudak. 'Modular Domain Specific Languages and Tools'. In: *in Proceedings of Fifth International Conference on Software Reuse*. IEEE Computer Society Press, 1998, pp. 134–142.

[8] Sebastian Hunt and David Sands. 'Binding Time Analysis: A New PERspective.' In: Jan. 1991, pp. 154–165. DOI: 10.1145/115865.115881.

[9] IEEE. 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: 10.1109/IEEESTD.2008.4610935.

[10] Neil D. Jones. 'An Introduction to Partial Evaluation'. In: *ACM Comput. Surv.* 28.3 (Sept. 1996), pp. 480–503. ISSN: 0360-0300. DOI: 10.1145/243439.243447.

[11] Neil Jones, Peter Sestoft and Harald Søndergaard. 'An Experiment in Partial Evaluation: The Generation of a Compiler Generator.' In: vol. 202. May 1985, pp. 124–140. ISBN: 978-3-540-15976-6. DOI: 10.1007/3-540-15976-2_6.

[12] Uwe Meyer. 'Correctness of on-line partial evaluation for a Pascal-like language'. In: *Science of Computer Programming* 34.1 (1999), pp. 55–73. ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(98)00015-X. URL: https://www.sciencedirect.com/science/article/pii/S016764239800015X.

[13] Netscape Communications. *INDUSTRY LEADERS TO ADVANCE STANDARDIZATION OF NETSCAPE'S JAVASCRIPT AT STANDARDS BODY MEETING*. Nov. 1996. URL: https://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html (visited on 03/12/1998).

[14] Netscape Communications. *NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET*. Dec. 1995. URL: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (visited on 16/09/2007).

[15] Arnoud Roo. 'Hybrid Partial Evaluation of Ruby'. MA thesis. Universiteit van Amsterdam, June 2012.

[16] Amin Shali and William Cook. 'Hybrid Partial Evaluation'. In: vol. 46. Oct. 2011, pp. 375–390. DOI: 10.1145/2076021.2048098.

[17] Shape Security, Inc. *Shift AST*. URL: https://shift-ast.org/.

A AST

Listing 34: AST of listing 21. Some fields have been elided for brevity.

```
Script { statements: [
  FunctionDeclaration { // function power(x, n){ ..
    name: BindingIdentifier { "power" },
    params: FormalParameters { [
      BindingIdentifier { "x" },
      BindingIdentifier { "n" }
    ] },
    body: FunctionBody { [
      VariableDeclarationStatement { VariableDeclaration { // let ret =
          1;
        kind: "let",
        declarators: [ VariableDeclarator {
          binding: BindingIdentifier { "ret" },
          init: LiteralNumericExpression { 1 }
        } ]
      } }, // VariableDeclaration, VariableDeclarationStatement
      WhileStatement { // while(n > 0){ ..
        test: BinaryExpression {
          left: IdentifierExpression { "n" },
          operator: ">",
          right: LiteralNumericExpression { 0 }
        }, // BinaryExpression
        body: BlockStatement { Block { [ // ret = ret * x;
          ExpressionStatement { AssignmentExpression {
            binding: AssignmentTargetIdentifier { "ret" },
            expression: BinaryExpression {
              left: IdentifierExpression { "ret" },
              operator: "*",
              right: IdentifierExpression { "x" }
            }
          } }, // AssignmentExpression, ExpressionStatement
          ExpressionStatement { UpdateExpression { // n--;
            binding: AssignmentTargetIdentifier { "n" },
            operator: "--",
            isPrefix: false,
          } } // UpdateExpression, ExpressionStatement
        ] } } // Block, BlockStatement
      }, // WhileStatement
      // return ret;
      ReturnStatement { IdentifierExpression { "ret" } }
    ] } // FunctionBody
  }, // FunctionDeclaration
  VariableDeclarationStatement { VariableDeclaration { // let a = power
      (2, 3)
    kind: "let",
    declarators: [ VariableDeclarator {
      binding: BindingIdentifier { "a" },
      init: CallExpression {
        callee: IdentifierExpression { "power" },
        arguments: [
```

```
49          LiteralNumericExpression { 2 },
50          LiteralNumericExpression { 3 }
51        ]
52      } // CallExpression
53    } ] // VariableDeclarator
54  } }, // VariableDeclaration, VariableDeclarationStatement
55  ExpressionStatement { CallExpression { // console.log('2**3', a)
56    callee: StaticMemberExpression { // console.log
57      object: IdentifierExpression { "console" },
58      property: "log"
59    },
60    arguments: [ // ("2**3", a)
61      LiteralStringExpression { "2**3 =" },
62      IdentifierExpression { "a" }
63    ]
64  } } // CallExpression, ExpressionStatement
65 ] } // Script
```

B Residual code

Listing 35: Matrix multiplication, specialized for 3x3 matrices with a fixed left-hand side matrix

```
function mul$0$2$3(b, x) {
  const ret = new Array(9);
  ret.fill(0);
  {
    {
      {
        ret[0] += 0 * b[0];
        ret[0] += .8414709848078965 * b[3];
        ret[0] += .9092974268256817 * b[6];
      }
      ret[0] *= x;
      {
        ret[1] += 0 * b[1];
        ret[1] += .8414709848078965 * b[4];
        ret[1] += .9092974268256817 * b[7];
      }
      ret[1] *= x;
      {
        ret[2] += 0 * b[2];
        ret[2] += .8414709848078965 * b[5];
        ret[2] += .9092974268256817 * b[8];
      }
      ret[2] *= x;
    }
    {
      {
        ret[3] += .1411200080598672 * b[0];
        ret[3] += -0.7568024953079282 * b[3];
        ret[3] += -0.9589242746631385 * b[6];
      }
      ret[3] *= x;
      {
        ret[4] += .1411200080598672 * b[1];
        ret[4] += -0.7568024953079282 * b[4];
        ret[4] += -0.9589242746631385 * b[7];
      }
      ret[4] *= x;
      {
        ret[5] += .1411200080598672 * b[2];
        ret[5] += -0.7568024953079282 * b[5];
        ret[5] += -0.9589242746631385 * b[8];
      }
      ret[5] *= x;
    }
    {
      {
        ret[6] += -0.27941549819892586 * b[0];
        ret[6] += .6569865987187891 * b[3];
        ret[6] += .9893582466233818 * b[6];
      }
```

```
51      ret[6] *= x;
52      {
53        ret[7] += -0.27941549819892586 * b[1];
54        ret[7] += .6569865987187891 * b[4];
55        ret[7] += .9893582466233818 * b[7];
56      }
57      ret[7] *= x;
58      {
59        ret[8] += -0.27941549819892586 * b[2];
60        ret[8] += .6569865987187891 * b[5];
61        ret[8] += .9893582466233818 * b[8];
62      }
63      ret[8] *= x;
64    }
65  }
66  return ret;
67 }
```

Listing 36: FFT, specialized for a radix size of 128

```
function perform$1(data) {
  {
    let fact = [1, 0];
    for (let group = 0; group < 1; group++) {
      for (let pair = group; pair < 128; pair += 2) {
        const match = pair + 1, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
        data[pair] = addComplex_0(data[pair], prod);
      }
      fact = addComplex_0(mulComplex$0(fact), fact);
    }
    fact = [1, 0];
    for (let group = 0; group < 2; group++) {
      for (let pair = group; pair < 128; pair += 4) {
        const match = pair + 2, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
        data[pair] = addComplex_0(data[pair], prod);
      }
      fact = addComplex_0(mulComplex$0_0(fact), fact);
    }
    fact = [1, 0];
    for (let group = 0; group < 4; group++) {
      for (let pair = group; pair < 128; pair += 8) {
        const match = pair + 4, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
        data[pair] = addComplex_0(data[pair], prod);
      }
      fact = addComplex_0(mulComplex$0_1(fact), fact);
    }
    fact = [1, 0];
    for (let group = 0; group < 8; group++) {
      for (let pair = group; pair < 128; pair += 16) {
        const match = pair + 8, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
        data[pair] = addComplex_0(data[pair], prod);
      }
      fact = addComplex_0(mulComplex$0_2(fact), fact);
    }
    fact = [1, 0];
    for (let group = 0; group < 16; group++) {
      for (let pair = group; pair < 128; pair += 32) {
        const match = pair + 16, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
        data[pair] = addComplex_0(data[pair], prod);
      }
      fact = addComplex_0(mulComplex$0_3(fact), fact);
    }
    fact = [1, 0];
    for (let group = 0; group < 32; group++) {
      for (let pair = group; pair < 128; pair += 64) {
        const match = pair + 32, prod = mulComplex_0(fact, data[match]);
        data[match] = subComplex_0(data[pair], prod);
```

```
53        data[pair] = addComplex_0(data[pair], prod);
54      }
55      fact = addComplex_0(mulComplex$0_4(fact), fact);
56    }
57    fact = [1, 0];
58    for (let group = 0; group < 64; group++) {
59      for (let pair = group; pair < 128; pair += 128) {
60        const match = pair + 64, prod = mulComplex_0(fact, data[match]);
61        data[match] = subComplex_0(data[pair], prod);
62        data[pair] = addComplex_0(data[pair], prod);
63      }
64      fact = addComplex_0(mulComplex$0_5(fact), fact);
65    }
66  }
67  return data;
68 }
69 function mulComplex$0_5(b) {
70  return [-0.0012045437948276074 * b[0] - .049067674327418015 * b[1],
        -0.0012045437948276074 * b[1] + .049067674327418015 * b[0]];
71 }
72 function mulComplex$0_4(b) {
73  return [-0.004815273327803114 * b[0] - .0980171403295606 * b[1],
        -0.004815273327803114 * b[1] + .0980171403295606 * b[0]];
74 }
75 function mulComplex$0_3(b) {
76  return [-0.019214719596769552 * b[0] - .19509032201612825 * b[1],
        -0.019214719596769552 * b[1] + .19509032201612825 * b[0]];
77 }
78 function mulComplex$0_2(b) {
79  return [-0.07612046748871323 * b[0] - .3826834323650898 * b[1],
        -0.07612046748871323 * b[1] + .3826834323650898 * b[0]];
80 }
81 function mulComplex$0_1(b) {
82  return [-0.2928932188134525 * b[0] - .7071067811865475 * b[1],
        -0.2928932188134525 * b[1] + .7071067811865475 * b[0]];
83 }
84 function mulComplex$0_0(b) {
85  return [-0.9999999999999998 * b[0] - b[1], -0.9999999999999998 * b[1] +
         b[0]];
86 }
87 function mulComplex$0(b) {
88  return [-2 * b[0] - 1.2246467991473532e-16 * b[1], -2 * b[1] +
        1.2246467991473532e-16 * b[0]];
89 }
```

Listing 37: Specialized small finite state machine

```
function fsm$0(ret, getEvent) {
  ret.push("closed");
  const event = getEvent();
  {
    if (event === "open") {
      return fsm$0_0(ret, getEvent);
    }
    if (event === "lock") {
      return fsm$0_1(ret, getEvent);
    }
  }
  return ret;
}
function fsm$0_1(ret, getEvent) {
  ret.push("locked");
  const event = getEvent();
  {
    if (event === "unlock") {
      return fsm$0(ret, getEvent);
    }
  }
  return ret;
}
function fsm$0_0(ret, getEvent) {
  ret.push("opened");
  const event = getEvent();
  {
    if (event === "close") {
      return fsm$0(ret, getEvent);
    }
  }
  return ret;
}
```