Interpretation of Dynamic Languages in Hardware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the academic degree

Diplom-Ingenieur

in the Master's Program

COMPUTER SCIENCE

Submitted by:
  Alwin W. Zulehner

At the:
  Institute for Computer Architecture

Advisor:
  Univ.-Prof. Dr. Volker Strumpen

Linz, June 2015
Kurzfassung


Wir haben einen Interpreter Prototypen in programmierbarer Hardware auf einem Altera FPGA implementiert, und Funktionalität und Optimierungspotential mittels Benchmarks evaluiert. Unser Prototype ist funktional aber nicht optimiert. Wir haben wesentliche Optimierungen indentifiziert, deren Implementierung jedoch den Rahmen dieser Arbeit sprengen würde. Wir erwartern, dass unser Hardware Interpreter mit diesen Optimierungen 2.65 mal weniger Clock Zyklen zum ausführen userer Benchmarks benötigt als der General Purpose Prozessor auf dem die Software Interpreter V8 und SpiderMonkey laufen. Diese Differenz in der Anzahl der Clock Zyklen kann dazu genutzt werden um die Performanz zu erhöhen oder um den Energieverbrauch zu reduzieren.
Abstract

We investigate the applicability of hardware accelerators for the interpretation of dynamic languages, with the goal to increase performance, and to reduce power consumption. This thesis is based on JavaScript, one of the most commonly used dynamic programming languages in web browsers. ECMA International provides an informal language specification for JavaScript, ECMAScript, but the software implementations, V8 of Google and SpiderMonkey of Mozilla disregard this specification in parts. Therefore, we believe that a formal semantics of JavaScript should be a helpful alternative. We develop a formal semantics for JavaScript, which serves also as reference for our hardware interpreter.

We have implemented a prototype of a JavaScript subset interpreter in programmable hardware on an Altera FPGA, and have evaluated its functionality and potential for optimizations with benchmarks. Our prototype is functional, but not optimized. We identified essential optimizations, but their implementation is beyond the scope of this thesis. With these optimizations, we expect that our hardware interpreter requires 2.65 times fewer clock cycles than the software interpreters V8 and SpiderMonkey, running on a general purpose processor, to execute our benchmark programs. We can use the difference in the number of clock cycles to increase the performance, or to reduce power consumption of executing JavaScript programs.
# Contents

## 1 Introduction

## 2 The Language JSub

2.1 JSub Syntax ................................................. 3
2.2 JSub Object Model ........................................... 5
  2.2.1 Objects .................................................. 6
  2.2.2 Functions and Constructors .............................. 7
  2.2.3 Prototype-Based Inheritance ............................. 7

## 3 Formal Semantics of JSub

3.1 Big-Step Structural Operational Semantics .................. 11
3.2 The semantic framework $K$ ................................ 12
3.3 Evaluation of JavaScript programs ............................ 14

## 4 Machine Language

4.1 Tagged Pointers ............................................. 19
4.2 JSub Program Representation ................................ 22
4.3 JSub Object Representation .................................. 23

## 5 Microarchitecture

5.1 Datapath ..................................................... 25
5.2 Control Unit ................................................ 26
5.3 Implementation in $K$ ......................................... 29

## 6 Prototype Implementation

6.1 Bit Format of Tagged Pointers ................................. 32
6.2 FPGA implementation ....................................... 32
6.3 Runtime Environment ....................................... 34

## 7 Optimizations

7.1 Lexical Addressing .......................................... 37
7.2 Optimizations for ALU operations ............................ 39
7.3 Hardware Support for Strings ................................ 40
7.4 Additional Buses in the Datapath ............................. 41
7.5 Hardware Support for Loops .................................. 42
7.6 Property Lookup ................................................... 43

8 Conclusion and Future Work ................................. 45

A Formal Semantics of JavaScript ............................ 48
A.1 Semantic Domains ........................................... 48
A.2 Judgements .................................................... 50
A.2.1 Program .................................................... 51
A.2.2 Body ....................................................... 51
A.2.3 Declarations .............................................. 51
A.2.4 Statements ................................................ 53
A.2.5 Expressions ................................................ 57
A.2.6 Argument Lists ......................................... 77
A.2.7 Auxiliary Functions .................................... 78

B Tagged Pointers .................................................... 85
B.1 Member Expression .......................................... 85
B.2 Call Expression .............................................. 85
B.3 New Expression .............................................. 86
B.4 Function Expression ....................................... 86
B.5 Function Declaration ..................................... 86
B.6 ALU Operation ................................................ 87
B.7 Assign Expression .......................................... 87
B.8 Add Expression .............................................. 87
B.9 Less than Expression ...................................... 87
B.10 Instanceof Expression ................................... 88
B.11 In Expression ................................................ 88
B.12 Equals Expression ....................................... 88
B.13 Logical Expression ....................................... 88
B.14 Typeof Expression ....................................... 89
B.15 Delete Expression ....................................... 89
B.16 If Statement ................................................ 89
B.17 Statement List ............................................. 89
B.18 Return Statement ......................................... 90
B.19 Throw Statement .......................................... 90
B.20 Labelled Statement ...................................... 90
B.21 Try Statement ............................................. 90
B.22 Environment ................................................. 91
B.23 Object ........................................................ 91
B.24 Function ...................................................... 91
B.25 Reference Pointer ........................................ 92

C FSM of the Control Unit ........................................ 93
Contents

Bibliography 133
List of Figures

2.1 Abstract syntax of JSub in BNF .............................................. 4
2.2 Desugaring property accesses in JavaScript ............................. 5
2.3 Desugaring of JavaScript object literals ................................. 5
2.4 Graphical representation for JavaScript objects ......................... 6
2.5 Graphical representation of JSub functions ............................. 8
2.6 Prototype-based inheritance in JavaScript ............................... 9

3.1 JavaScript example that yields different result on V8 and SpiderMonkey 10
3.2 Abstract syntax of EXP .......................................................... 11
3.3 Big-step SOS rules for EXP ...................................................... 12
3.4 Proof tree for evaluating $(1 + x) + 3$ .................................... 12
3.5 Specification of EXP in $\mathcal{K}$ ............................................. 13
3.6 Big-step SOS rule for evaluating member expressions .................. 15
3.7 Big-step SOS rule for evaluating assign expressions .................. 16
3.8 Proof tree for evaluating $1["a"] = 1$ ..................................... 17
3.9 JavaScript programs that yield different results on V8 and SpiderMonkey, and $\sigma$JSub ......................................................... 17

4.1 Tagged graphs in a list memory ............................................. 20
4.2 Tagged graph representation for $a + 1$ .................................. 20
4.3 Tagged graph representation for if statements ......................... 20
4.4 Tagged graph representation of a JavaScript program .................. 22
4.5 JavaScript program in a list memory ....................................... 23
4.6 Tagged graph representation for JavaScript objects .................... 23

5.1 Datapath of a hardware interpreter for JavaScript .................... 26
5.2 FSM for the control unit ....................................................... 27
5.3 FSM for evaluating statement lists ........................................ 27
5.4 FSM for evaluating member expressions ............................... 28
5.5 $\mathcal{K}$ configuration for the hardware interpreter ................... 29
5.6 $\mathcal{K}$ rules for the hardware interpreter ............................... 30
5.7 $\mathcal{K}$ configuration for representing pairs ............................. 30

6.1 Prototype of a hardware interpreter for JavaScript ................... 31
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Bit format for IEEE 754 64-bit floating point numbers</td>
<td>32</td>
</tr>
<tr>
<td>6.3</td>
<td>Tagged pointers in IEEE 754 64-bit floating point numbers</td>
<td>32</td>
</tr>
<tr>
<td>7.1</td>
<td>Tagged graph for environments in $\mu$JSub</td>
<td>38</td>
</tr>
<tr>
<td>7.2</td>
<td>Tagged graph representation for environments with lexical addressing</td>
<td>39</td>
</tr>
<tr>
<td>7.3</td>
<td>Tagged graph representation for loops</td>
<td>43</td>
</tr>
<tr>
<td>7.4</td>
<td>Tagged graph representation for simple loops</td>
<td>43</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Tags for representing JavaScript programs and JavaScript objects</td>
<td>21</td>
</tr>
<tr>
<td>6.1</td>
<td>Delay of Floating Point Operations</td>
<td>33</td>
</tr>
<tr>
<td>7.1</td>
<td>Estimated execution time of $\mu$JSub if all optimizations were implemented $(\mu$JSub opt)</td>
<td>37</td>
</tr>
<tr>
<td>7.2</td>
<td>Execution time on software interpreters</td>
<td>37</td>
</tr>
<tr>
<td>7.3</td>
<td>Estimated execution of $\mu$JSub with lexical addressing $(mu$JSub la)</td>
<td>38</td>
</tr>
<tr>
<td>7.4</td>
<td>Estimated execution time of $\mu$JSub with optimized ALU operations $(\mu$JSub alu)</td>
<td>40</td>
</tr>
<tr>
<td>7.5</td>
<td>Estimated execution time of $\mu$JSub with hardware support for strings $(\mu$JSub hws)</td>
<td>41</td>
</tr>
<tr>
<td>7.6</td>
<td>Estimated execution time of $\mu$JSub with additional buses in the datapath $(\mu$JSub buses)</td>
<td>41</td>
</tr>
<tr>
<td>7.7</td>
<td>Estimated execution time of $\mu$JSub with hardware support for loops $(\mu$JSub loop)</td>
<td>42</td>
</tr>
<tr>
<td>7.8</td>
<td>Estimated execution time of $\mu$JSub with move-to-front heuristic for property accesses $(\mu$JSub MTF)</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

JavaScript is one of the most commonly used programming languages. Not only does JavaScript enable dynamic web pages, it also runs on servers and in mobile applications. Currently, software interpreters evaluate JavaScript programs or compile them into machine code with just-in-time (JIT) compilers. We investigate the applicability of a hardware accelerator for JavaScript, with the goal to reduce the power consumption and to increase the performance. We base our approach on the seminal work of Gerald J. Sussman and Guy L. Steele, who implemented and interpreter for the functional language Scheme in hardware [10][13]. Our goal is to investigate whether we can extend the approach of Sussman and Steele to develop a hardware accelerator for the modern language JavaScript. In contrast to the functional language Scheme, we also have to consider an object model when we develop a hardware interpreter for JavaScript. Our approach is not only applicable to JavaScript, but we choose JavaScript as demonstration language because it covers functional, imperative, and object oriented aspects of programming languages.

There exists an informal language specification for JavaScript, ECMAScript, but the software implementations, V8 of Google and SpiderMonkey disregard this specification in parts. Therefore, we develop a formal semantics for JavaScript, to serve a reference implementation. We can prove results of program evaluation with the formal semantics and use it as reference for our hardware interpreter. We implemented a prototype of the hardware interpreter on programmable hardware, an Altera FPGA. The prototype is not a complete implementation of JavaScript, because we do not implement all built-in functions and built-in objects. We use benchmark programs to evaluate the functionality and the optimization potential of our prototype.

This thesis is structured as follows. In Chapter 2 we define our source language, a subset of JavaScript. We give an overview of supported language constructs and discuss the object model. In chapter 3, we formalize the semantics of our JavaScript subset
and demonstrate how to prove bugs in JavaScript engines. Chapter 4 introduces the machine language of our hardware interpreter. We discuss how to represent JavaScript programs and objects in machine readable form. In Chapter 5 we show how to translate the formal description of JavaScript into a hardware interpreter which evaluates JavaScript programs recursively. We discuss our prototype implementation of the interpreter on an FPGA in Chapter 6. In Chapter 7 we discuss performance optimization for our hardware interpreter and analyze their impact on the runtime with benchmark programs.
In this chapter, we define a subset of the strict mode of the ECMAScript Language Specification 5.1 [7], suitable for realization in hardware. We call this subset JSub. In section 2.1, we introduce the abstract syntax of JSub, which is a subset of the ECMAScript syntax [7, Appendix B]. We discuss the reduction of the original JavaScript syntax by means of examples and explain how to desugar JavaScript programs to JSub programs. In section 2.2 we describe a simplified version of the JavaScript object model. In particular, we discuss objects, functions and prototype based inheritance with examples.

2.1 JSub Syntax

In this section, we present the abstract syntax of JSub in BNF (figure 2.1). We omit the concrete syntax in this thesis, because the precedence of the operators is the same as in the ECMAScript Language Specification[7, Appendix B]. Our syntax reduction does not affect the capabilities of JavaScript, because we can desugar all missing syntactic JavaScript constructs to JSub syntax. In the remainder of this section, we discuss differences between JavaScript and JSub with examples.

In JSub, we demand an order for declarations and statements inside function bodies to reduce the effort of our compiler. Function declarations are followed by variable declarations and finally by statements. This order of declarations and statements does not affect the semantics of JavaScript programs, because each time we enter a function body, JavaScript evaluates all declarations, independent of their location in the function body, before executing the statements. First, JavaScript executes all function declarations, followed by function declarations, and finally the statements.
The Language JSub

\[ P ::= \langle BODY \rangle \]

\[ BODY ::= \langle FD \rangle \langle VD \rangle \langle S \rangle \]

\[ FD ::= \langle empty \rangle \]
\[ | \quad \text{function}(I) \{ \langle IDENTS \rangle \} \{ \langle BODY \rangle \} \langle FD \rangle \]

\[ VD ::= \langle empty \rangle \]
\[ | \quad \text{var}(I) ; \langle VD \rangle \]

\[ S ::= \text{empty} \]
\[ | \quad \langle E \rangle ; \]
\[ | \quad \text{if}(\langle E \rangle) \langle S \rangle \text{else} \langle S \rangle \]
\[ | \quad \text{while}(\langle E \rangle) \langle S \rangle \]
\[ | \quad \text{continue}(I) ; \]
\[ | \quad \text{break}(I) ; \]
\[ | \quad \text{return}(E) ; \]
\[ | \quad (I) : (S) \]
\[ | \quad \text{throw}(E) ; \]
\[ | \quad \{ \langle S \rangle \} \quad \text{catch}(\langle I \rangle) \{ \langle S \rangle \} \quad \text{finally} \{ \langle S \rangle \} \]
\[ | \quad (S) \langle S \rangle \]

\[ IDENTS ::= \text{empty} \]
\[ | \quad \langle IL \rangle \]

\[ IL ::= \langle I \rangle \]
\[ | \quad (I), \langle IL \rangle \]

\[ E ::= \text{this} \]
\[ | \quad (I) \]
\[ | \quad (L) \]
\[ | \quad \text{function}(I) \{ \langle IDENTS \rangle \} \{ \langle BODY \rangle \} \]
\[ | \quad \langle E \rangle \[ \langle E \rangle \]
\[ | \quad \langle E \rangle (\langle AL \rangle) \]
\[ | \quad \text{new}(\langle E \rangle)(\langle ARGS \rangle) \]
\[ | \quad \langle E \rangle = \langle E \rangle \]
\[ | \quad \langle E \rangle (\langle BinOp \rangle) \langle E \rangle \]
\[ | \quad \langle UnaryOp \rangle \langle E \rangle \]
\[ | \quad \text{delete}(\langle E \rangle) \]

\[ ARGS ::= \text{empty} \]
\[ | \quad \langle AL \rangle \]

\[ AL ::= \langle E \rangle \]
\[ | \quad (E), \langle AL \rangle \]

\[ L ::= \text{String} | \text{Number} | \text{Boolean} | \text{null} \]

\[ BinOp ::= \ast | / | + | - | < | > | \langle | \rangle | \text{instanceof} | \text{in} | == | === | \& | | | \&\& | || \]

\[ UnaryOp ::= \text{typeof} | + | - | ~ | ! \]

Figure 2.1: Abstract syntax of JSub in BNF
We access properties of JSub objects with the bracket notation. JavaScript also allows property accesses with the dot-notation. However, JSub does not support this notation, because the bracket notation subsumes the dot-notation. Figure 2.2 shows a small JavaScript code sample were we show two semantically equivalent property accesses, one with the dot-notation and one with the bracket notation.

```javascript
//obj.a;
obj["a"];  
```

Figure 2.2: Desugaring property accesses in JavaScript

In JSub, we generate new objects with the `Object` constructor. The `Object` constructor is a built-in function, which returns a new object. JavaScript also allows to instantiate objects with an object literal. An object literal contains all properties as a list, enclosed in curly braces. Object literals are syntactic sugar for generating a new object with the `Object` constructor and setting the properties individually. Figure 2.3 shows how to desugar an object literal. The object `x` has two properties, data property `a` and accessor property `b`. A data property stores data directly, whereas accessor properties define a `get` and/or a `set` function. The accessor property `b` defines only a `get` function to read its value. When we read the value of `b`, the `get` function is called without arguments. We desugar accessor property `b` to the function `getB`.

```javascript
//x = {a:1, get b() {return 5; }};
x = new Object();
x["a"] = 1;
x["getB"] = function f() {return 5;};
//x.b;
x["getB"]();
```

Figure 2.3: Desugaring of JavaScript object literals

### 2.2 JSub Object Model

In this section, we describe the object model of JSub subset. The object model of JSub is a subset of the JavaScript object model, because JSub does not support accessor properties for objects (Section 2.1). In Section 2.2.1, we discuss the structure of objects. In Section 2.2.2 we discuss JSub functions, because they are callable objects. We conclude this section with an example to explain prototype based inheritance in JavaScript.
2.2.1 Objects

In JSub, objects are associations with strings as keys and data properties as values. Data properties are quadruples, consisting of a value and three Boolean flags, writable, enumerable and configurable. The writable flag determines whether we can change the value of the data property. If the enumerable flag is set, the property is enumerated in an iteration over all properties of an object. The third flag, configurable, determines whether we can change any flag of the data property or delete the whole data property. Therefore, setting the configurable flag of a data property to false is irreversible.

Besides data properties, JSub objects have a fixed set of internal properties, which are invisible to the programmer. All objects have internal properties named class, extensible and prototype. Class determines the class name, the extensible flag determines whether we can add new data properties to the object and prototype points to the prototype of the object. The internal property prototype points always to an object, except for the Object prototype object (OPO) where it is a null pointer. The Object prototype object is the root object in JavaScript.

Figure 2.4 illustrates the object model by means of a small example. The program instantiates a new object x and adds a data property "a" with the value 1. Figure 2.4 also contains a graphical representation of the object x. We represent objects with boxes. We list the internal properties in the upper part and the data properties in the lower part. Object x has only one data property, named "a". According to the ECMAScript language specification [7], all flags of a user defined data property are initially set to true (T). This means, that we can change or delete data property "a". The internal property prototype points to the OPO. All objects linked through internal property prototype form a prototype chain. The OPO is the only object in JavaScript with a null reference as internal property prototype and is therefore the tail end of all prototype chains.

```javascript
1 var x;
2 x = new Object();
3 x["a"] = 1;
```

Figure 2.4: Graphical representation for JavaScript objects
2.2.2 Functions and Constructors

JSub functions are callable objects with additional internal properties. Constructors are functions we can use to instantiate new objects. All user defined functions are constructors, but some JSub built-in functions are not. For example, the toString function of the OPO is not a constructor. We discuss JSub functions with the code sample in Figure 2.5.

Figure 2.5 contains a JavaScript example and represents the objects we obtain after its evaluation. Function object \( f \) has four additional internal properties compared to non-function objects. Flag \( isConstructor \) determines, whether \( f \) is a constructor. List \( parameter \) stores the parameter names. Internal property \( parameter \) of \( f \) is a list with the identifiers \( x \) and \( y \). Internal property \( body \) stores the JSub code of the function body and internal property \( env \) points to the environment, in which \( f \) is defined. A reference to the environment is necessary, because JavaScript is statically scoped. The internal property \( prototype \) of all function objects points to the Function prototype object (FPO). The Function prototype object is a built-in object of JavaScript.

Every function object contains at least two data properties. Data property \( length \) stores the number of specified arguments, which is 2 for function \( f \). If we call a JavaScript function with less arguments than specified, the missing ones are set to the value \( undefined \). Data property \( length \) can not be manipulated, which is indicated by setting all three flags to false. For each function declaration, we create two objects, the function object and the object \( proto \), serves as prototype for all objects created the function. Data property \( prototype \) of \( f \) references \( proto \). We can’t delete data property \( prototype \) of \( f \), but we can change its value, as indicated by the writable flag set to true.

2.2.3 Prototype-Based Inheritance

JavaScript implements a unique form of object inheritance called prototype-based inheritance. This means, that an object inherits the properties from all objects in its prototype chain. We discuss prototype-based inheritance in JavaScript with the code example in Figure 2.6.

The JavaScript example in Figure 2.6, defines a function \( f \), and invokes \( f \) to instantiate two new objects, \( x \) and \( y \). Function object \( f \) resembles as described in Section 2.2.2. Internal property \( prototype \) of \( x \) and \( y \) point to the \( proto \) object. Therefore both objects, \( x \) and \( y \) share the data property \( a \) of the \( proto \) object. Line 11 reassigns \( a \) in object \( x \). JavaScript does not change the value of \( a \) in the \( proto \) object, because this would also
function f(x, y) {
    return x+y;
}

f["prototype"]["a"] = 1;

Figure 2.5: Graphical representation of JSub functions

affect y. Instead, JavaScript creates a new data property in x which shadows property a of the proto object.

Line 2 accesses this. The value of this depends on how function f is invoked. In this example, f is called as constructor in a new expression. Before the function body of f is executed, JavaScript binds this to a new object with internal property prototype set to the proto object. Therefore, x and y have a data property with name b. Line 12 looks up the property a in object y, which results in the value 1. Object y’s set of data properties does not include a data property with name a. Therefore, JavaScript continues the lookup along the prototype chain and finds it in the proto object.
We define JSub as a subset of the strict variant of JavaScript, including the JavaScript object model. We choose JSub in a way, that it has a simpler syntax than JavaScript and that we can transform all JavaScript programs straightforward into JSub programs. We choose JSub for implementation in hardware, because it has a simpler syntax than JavaScript. From the simpler syntax of JSub results a slightly simpler object model than the object model of JavaScript, because the JSub syntax does not include accessor properties for objects. JSub provides a small subset of the JavaScript built-in functions, sufficiently many to demonstrate the execution of JavaScript programs on our hardware interpreter.
Chapter 3

Formal Semantics of JSub

This chapter we formalizes the semantics of JSub. The ECMAScript Language Specification 5.1 [7] specifies ECMAScript informally in natural language, but we prefer formal semantics. Consider the code sample in Figure 3.1. When we execute the code sample on Google’s V8 (version 3.31.1) and Mozilla’s SpiderMonkey (version JavaScript-C40.0a1), we get different results. V8 throws a TypeError exception, whereas SpiderMonkey exits without an error. With an informal semantics, it is cumbersome to prove discrepancies between the software implementations and the ECMAScript Language Specification [7]. We use an implementation of the formal semantics in the semantic framework \( K \) to prove this discrepancies. Furthermore, formal semantics remove the ambiguities in the natural language of informal semantics and can be used for automatic test case generation.

This chapter is structured as follows. In Section 3.1, we introduce big-step structural operational semantics (big-step SOS) and show how we can describe the semantics of a simple programming language. In Section 3.2, we introduce \( K \), a semantic framework we use to implement the formal semantics of JavaScript. In Section 3.3, we show how to describe the formal semantics of JSub with big-step SOS and how to implement them in \( K \). Furthermore, we determine the correct result, according to the ECMAScript Language Specification [7], of the code sample in figure 3.1 with our formal semantics. Appendix A contains the complete semantic description of JSub.

```
1 "use strict"
2 1["a"] = 1;
```

Figure 3.1: JavaScript example that yields different result on V8 and SpiderMonkey
3.1 Big-Step Structural Operational Semantics

Big-step structural operational semantics (big-step SOS). Big-step SOS were introduced by Gilles Kahn in 1987 [11] under the name of natural semantics. Big-step SOS models a recursive interpreter for a programming language with inference rules. The inference rules are called structural, because they operate on the structure of programs, the syntax. In Chapter 5 of this thesis, we show how generate a FSM from the big-step SOS rules which evaluates JavaScript programs recursively (chapter 5). The inference rules of big-step SOS contain judgments, relations on so-called configurations. Judgments are written as \( C \downarrow R \) and are read as "\( C \) evaluates to \( R \)". \( C \) is the input configuration, a tuple that contains the expression to evaluate and the current state of the interpreter. \( R \) is the result configuration, a tuple that contains the result of the evaluation and the state of the interpreter. The state of the interpreter can be changed by the evaluation. The inference rules of big-step SOS describe evaluation steps from \( C \) to \( R \) without intermediate results.

In the remainder of this section, we give an introductory example to big-step SOS. We introduce EXP, a simple language to evaluate arithmetic expressions. Figure 3.2 shows the abstract syntax of EXP in BNF.

\[
\langle EXP \rangle ::= \langle Int \rangle \\
| \langle Id \rangle \\
| \langle EXP \rangle + \langle EXP \rangle
\]

Figure 3.2: Abstract syntax of EXP

An EXP expression is either an integer or an identifier, or the addition of two EXP expressions. In EXP, an identifier shall reference an integer in a store. In EXP, we define a store as partial function from identifiers to integer numbers. Judgments of the big-step SOS rules for EXP have the form:

\[
< e : Exp, \sigma : Store > \downarrow < i : Int >
\]

The input configurations for judgments are pairs, consisting of an expression \( e \) and a store \( \sigma \). The result configuration consists only of an integer number. The result configuration does not include a store, because the expressions of EXP do not have side effects.

We use big-step SOS rules to describe the semantics of EXP. The rules in Figure 3.3, follow the convention that \( n \in Int, i \in Id, e \in Exp \) and \( \sigma \in Store \). Rule \( ExpN \) describes the evaluation of number literals. The result of evaluating a number \( n \) with
store $\sigma$, is number $n$. Rule $\text{ExpId}$ describes the evaluation on identifier $i$ in store $\sigma$. Partial function $\sigma$ maps identifiers to numbers, so that identifier $i$ evaluates to $\sigma(i)$. The inference rules for evaluating number literals and identifiers are axioms, because they do not have any premises.

Inference rule $\text{ExpAdd}$ in Figure 3.3 has two premises. We can only apply rule $\text{ExpAdd}$ if both premises are fulfilled. We evaluate an addition only if expressions $e_1$ and $e_2$ evaluate to the number $n_1$ and $n_2$ in store $\sigma$, respectively. If so, the addition evaluates to the sum of $n_1$ and $n_2$. The operator $\text{+}_{\text{Int}}$ indicates an addition of two integer numbers.

\[
\begin{align*}
\text{EXPN} & \quad \langle n, \sigma \rangle \Downarrow \langle n \rangle \\
\text{ExpId} & \quad \langle i, \sigma \rangle \Downarrow \langle \sigma(i) \rangle \\
\text{ExpAdd} & \quad \langle e_1, \sigma \rangle \Downarrow \langle n_1 \rangle \quad \langle e_2, \sigma \rangle \Downarrow \langle n_2 \rangle \\
& \quad \langle e_1 + e_2, \sigma \rangle \Downarrow \langle n_1 + \text{Int} n_2 \rangle
\end{align*}
\]

Figure 3.3: Big-step SOS rules for EXP

The big-step SOS rules specify how to evaluate EXP expressions. We can derive the result of an EXP expression with a proof tree. We start with an EXP expression and apply the inference rules recursively. In figure 3.4, we show the proof tree for evaluating EXP expression $(1 + x) + 3$ in store $\sigma = \{x \rightarrow 2\}$.

\[
\begin{align*}
\langle 1, \{x \rightarrow 2\} \rangle & \Downarrow \langle 1 \rangle & \langle x, \{x \rightarrow 2\} \rangle & \Downarrow \langle 2 \rangle \\
\langle 1 + x, \{x \rightarrow 2\} \rangle & \Downarrow \langle 3 \rangle & \langle 3, \{x \rightarrow 2\} \rangle & \Downarrow \langle 3 \rangle \\
\langle (1 + x) + 3, \{x \rightarrow 2\} \rangle & \Downarrow \langle 6 \rangle
\end{align*}
\]

Figure 3.4: Proof tree for evaluating $(1 + x) + 3$

3.2 The semantic framework $\mathbb{K}$

$\mathbb{K}$ is a rewrite-based executable semantic framework, developed by Grigore Rosu in 2003 [12]. The $\mathbb{K}$ framework automatically generates an interpreter from semantic rules. We use the generated interpreter as reference implementation and compare the
evaluation results to the results of the JavaScript engines of V8 and SpiderMonkey. \( K \) also enables us to specify the structure of the program state. Therefore In the remainder of this section, we explain how to specify the simple language EXP in \( K \).

A semantic description in \( K \) consists of three parts. The syntax of the language, a configuration, and rules. The configuration describes the structure of the program state. \( K \) uses rewrite rules which can include parts of the configuration. Figure 3.5 shows the \( K \) specification of EXP.

```plaintext
module EXP
  syntax Exp ::= Int
      | Id
      | Exp "+" Exp [strict, left]
  syntax KResult ::= Int
  configuration
    <k>$PGM:Exp </k>
    <store> (String2Id("x") |-> 2):MyMap </store>
    rule <k>I:Id => N ...</k>
    <store>... I |-> N:Int ... </store>
    rule I1:Int + I2:Int => I1 +Int I2
endmodule
```

Figure 3.5: Specification of EXP in \( K \)

\( K \) uses and notation, similar to BNF, to specify the semantics of EXP. Productions rules in \( K \) can specify attributes in square brackets. In Figure 3.5, only the production rule for an addition has attributes. Attribute \textit{left} indicates, that the operator + is left associative. Attribute \textit{strict} indicates, that both subexpressions have to be evaluated before we evaluate the addition. \( K \) evaluates the subexpressions of the addition to \textit{results}, which we specify to be of type integer. The result has the same type as the result configuration of our big-step SOS rules for EXP. The attribute \textit{strict} reduces number of rules we have to specify, because \( K \) automatically adds rules which evaluate the subexpressions of an addition.

\( K \) configurations describe the state of a program. Therefore, \( K \) configurations are the equivalent to input configurations of big-step SOS judgments. In \( K \), we describe configurations with an XML like notation. The configuration contains a cell with name \( k \). The attribute \textit{strict} reduces number of rules we have to specify, because \( K \) automatically adds rules which evaluate the subexpressions of an addition. Variable \$PGM\ references the parsed program in the automatically generated interpreter for
EXP. We specify another cell with name store (line 10) for EXP, namely the store. The store is initially a map of identifier "x" to integer 2. The configuration can change during program evaluation.

We specify two rewrite rules for EXP. In line 13, we specify a rewrite rule for evaluating an identifier. The rule contains cells from the configuration and reads as follows: if identifier I maps to integer N in store, I evaluates to N. The rule in line 16 describes the evaluation of the addition. The operands of the addition are of type integer, because the attribute strict in the syntax definition reduces the operands to results of type integer. We use the built-in function +Int of K to add two integer numbers.

3.3 Evaluation of JavaScript programs

In this section, we show how to use big-step SOS and K to specify the semantics of JSub. We demonstrate how to prove the result of the example in Figure 3.1 with our formal description, because the example yields different results on the JavaScript engines of Google (V8) and Mozilla (SpiderMonkey). According to the ECMA-Script Language Specification [7], V8 yields the correct result, a TypeError exception. The program results in a TypeError exception, because it contains an attempt to add a property to a transient object. In the remainder of this section, we show how to describe this behavior with big-step SOS. In particular, we discuss the configurations for JavaScript expressions and the rules to evaluate the sample program.

The big-step SOS rules for JSub, have different judgments. We distinguish judgments for programs, function bodies, declarations, statements, expressions, and auxiliary functions. We write judgments as $C \Downarrow_n R$, where $n$ determines the type of the judgment, to increase the readability of complex rules. All judgments for JSub expressions are of the form

\[
< e : Expr, env : Loc, this : Value, \sigma : Store, eVal : ExprValue > \\
\Downarrow_n < \sigma_2 : Store, eVal_2 : ExprValue >
\]

The input configuration is a quintuple, where $e$ is the expression we wish to evaluate, $env$ is a pointer to the current environment in store $\sigma$ and $this$ holds the current this value. Store $\sigma$ contains JavaScript objects and environments. We add $eVal$ to the input configuration to indicate a thrown exception. With $eVal$ in the input configuration, we need fewer rules to describe the semantics of JavaScript, because we do not have to check for an exception after each sub-expression we evaluate. The result configuration of the judgment consists of the updated store $\sigma_2$ and $e_2$, which holds the result of
the evaluation. The formal definition of all types used in the semantic description of JavaScript is part of Appendix A.

The sample program in Figure 3.1 contains an assignment expression. The left hand side of the assignment expression is a member expression which accesses the property "a" of 1. We discuss the big-step SOS rules, needed to evaluate the assignment expression and the member expression in detail.

Figure 3.6 shows the big-step SOS rule we need to evaluate the member expression of the sample program in Figure 3.1. First, we evaluate $e_1$, which results in $exprVal_1$ and an updated store $\sigma_1$ because evaluating $e_1$ might cause side effects. The evaluation of $e_1$ might cause side effects, because $e_1$ is an arbitrary expression and can contain function calls or assignments. According to the language specification, we call the auxiliary function $GetValue$ to dereference $exprVal_1$. This is necessary because evaluating $e_1$ can result in a reference to a property of an object, or to a variable in an environment.

We introduce big-step SOS rules to describe the behavior of auxiliary functions and indicate them with $\Downarrow_6$. In our semantic description of JavaScript, exceptions and values are of the same type. Therefore we introduce type tags to distinguish them, where $val(v)$ indicates that $v$ is a value and not an exception. We evaluate $e_2$ and apply auxiliary function $GetValue$ to the result. Auxiliary function $ToString$ converts the value $v_2$ to string $s$. Furthermore, $v$ must not be null or undefined, which would lead to a TypeError exception. If all premises hold, the member expression $e_1[e_2]$ evaluates to a reference with the base $v$ and the name $s$, and the store $\sigma_5$.

\[\text{MemberExpr2} \]
\[
< e_1, env, this, \sigma, exprVal > \Downarrow_4 < \sigma_1, exprVal_1 >
\]
\[
< \text{GetValue}, exprVal_1, \sigma_1 > \Downarrow_6 < \sigma_2, val(v) >
\]
\[
< e_2, env, this, \sigma_2, val(v) > \Downarrow_4 < \sigma_3, exprVal_3 >
\]
\[
< \text{GetValue}, exprVal_3, \sigma_3 > \Downarrow_6 < \sigma_4, val(v_2) >
\]
\[
< \text{ToString}, val(v_2), \sigma_4 > \Downarrow_6 < \sigma_5, val(s) > \quad v \notin \{\text{null, undefined}\}
\]
\[
< e_1[e_2], env, this, \sigma, exprVal > \Downarrow_4 < \sigma_5, \text{ref}(v, s) >
\]

Figure 3.6: Big-step SOS rule for evaluating member expressions

Figure 3.7, shows the big-step SOS rule for evaluating the assign expression of the sample code in Figure 3.1. First, we evaluate $e_1$, which has to result in a reference and an updated store $\sigma_1$. A reference is a pair, consisting of a base value and a string, the name. We label references with the tag $\text{ref}$. Like in the rule for the member expression, we first evaluate $e_1$ and $e_2$ afterwards. The only difference is, that we do not call auxiliary function $GetValue$ after evaluating $e_1$, because we need a reference and not a value when we perform an assignment. We evaluate $e_2$ and call auxiliary function $GetValue$ to dereference the result of $e_2$. According to the language specification,
we have to throw a TypeError exception if the base of the reference, obtained by evaluating $e_1$, has the type Boolean, Number or String. We instantiate a new error object, \textit{error}. An object is a quintuple, consisting of a set of data properties and the internal properties \textit{prototype}, \textit{class}, \textit{extensible} and \textit{primitive value}. For more detail about the JSub object model and its formal definition, see Section 2.2 and Appendix A. The premise $\sigma_3 = (map, top)$ splits store $\sigma_3$ into its two components. Location $top$ is the next free location in $map$, which stores objects and environments. We define $map[top/error]$ as an update function of $map$ which stores $error$ at the location $top$. We define the update function formally in Appendix A. We create an updated store $\sigma_4$, with the updated map and $top + 1$ as next free location of $map$. If all premises hold, the assign expression $e_1 = e_2$ evaluates to the updated store $\sigma_4$ and the exception $error$, references by the location $top$.

\[< e_1, env, this, \sigma, exprVal > \downarrow_4 < \sigma_1, ref(base, name) > \]
\[< e_2, env, this, \sigma_1, ref(base, name)) > \downarrow_4 < \sigma_2, exprVal_2 > \]
\[< getValue, exprVal_2, \sigma_2 > \downarrow_6 < \sigma_3, val(v) > \]
\[base \in \text{Boolean} \cup \text{Number} \cup \text{String} \quad (\emptyset, \text{TEPO}, 'Error', \text{true}, \text{null}) = \text{error} \]
\[\sigma_3 = (map, top) \quad (map[top/error], top + 1) = \sigma_4 \]
\[< e_1 = e_2, env, this, \sigma, exprVal > \downarrow_4 < \sigma_4, exc(loc(top)) > \]

Figure 3.7: Big-step SOS rule for evaluating assign expressions

We use our big-step SOS description of JSub to determine the result of the sample program in Figure 3.1. Figure 3.8 shows the proof tree, we use to derive the result of the assign expression $1["a"] = 1$. We split the tree into two parts to maintain readability. The second inference rule in Figure 3.8 derives the result of the member expression $1["a"]$, which is the left hand side of the assign expression. The member expression results in a reference $r$ with the base 1 and the name 'a'. The assign expression results in a TypeError exception, because the base of $r$ is a Number.

We have shown how to prove, that V8 yields the same result as specified in the ECMAScript Language Specification [7], when we execute the example in figure 3.1. Of course, a manual proof is not suitable for large programs. Therefore, we implement the semantics of our subset of JavaScript in the semantic framework $\mathbb{K}$. $\mathbb{K}$ generates a software interpreter from the semantic rules, $\sigma$JSub, which we use as reference implementation.

With $\sigma$JSub, we found additional discrepancies between the JavaScript engines V8 and SpiderMonkey, and the ECMAScript Language Specification [7]. Consider the JavaScript examples in Figure 3.9, where V8 and SpiderMonkey yield different results
Formal Semantics of JSub

< 1["a"], env, this, σ, val(null) > ⊄ 4 < σ₁, ref(1, "a") >
< 1, env, this, σ₁, ref(1, "a") > ⊄ 4 < σ₂, val(1) >
< GetValue, val(1), σ₂ > ⊄ 4 < σ₃, val(1) >

1 ∈ Boolean ∪ Number ∪ String  (null, TEPO, "Error", true, null) = error
σ₃ = (map, 100) (map[100/error], 101) = σ₄

< 1["a"] = 1, env, this, σ, val(null) > ⊄ 4 < σ₄, exc(loc(100)) >
< 1, env, this, σ, val(null) > ⊄ 4 < σ₁, val(1) >
< GetValue, val(1), σ₁ > ⊄ 4 < σ₂, val(1) >
< "a", env, this, σ₂, val(1) > ⊄ 4 < σ₃, val("a") >
< GetValue, exprVal₃, val("a") > ⊄ 6 < σ₄, val("a") >
< ToString, val("a"), σ₄ > ⊄ 6 < σ₅, val("a") >
1 ∉ {null, undefined}
< 1["a"], env, this, σ, val(null) > ⊄ 4 < σ₅, ref(1, "a") >

Figure 3.8: Proof tree for evaluating 1["a"] = 1

than σJSub. In the example on the left hand side, V8 and SpiderMonkey throw a TypeError exception after executing \( f \), but σJSub throws the exception without executing \( f \).
In the example on the right hand side, V8 and SpiderMonkey throw a ReferenceError exception, but σJSub exits without exception.

```javascript
"use strict"

function f() {
  print("1");
}

undefined["a"] = f();
```

Figure 3.9: JavaScript programs that yield different results on V8 and SpiderMonkey, and σJSub

Summary

We introduce the big-step structural operational semantics (big-step SOS) of JSub. The complete semantics of JSub is described in Appendix A. Big-step SOS describe the execution of JSub on an abstract interpreter. In chapter 5, we translate the big-step SOS rule manually into a FSM, the control unit of our hardware interpreter. We implement the formal specification of JSub in the semantic framework \( \mathcal{K} \). \( \mathcal{K} \) generates a software interpreter for JSub, σJSub, which serves as a reference for our hardware interpreter. We compare the result of the program execution on σJSub to the result of the program execution on V8 and SpiderMonkey. We found discrepancies, which indicate
Chapter 4

Machine Language

In this chapter, we describe the machine language for our hardware interpreter. The machine language for our hardware interpreter contains high-level instructions, rather than low-level instructions like a RISC architecture. We represent JSub programs as tagged graphs in a list memory. Our approach is based on the representation of functional Scheme/Lisp program, developed by Gerald J. Sussman and Guy L. Steele [13] [5]. We extend their approach to represent JSub including JSub objects. This chapter is structured as follows. In section 4.1, we introduce tagged graphs and list memories. We discuss examples of JavaScript programs and JavaScript objects, represented as tagged graphs in Sections 4.2 and 4.3.

4.1 Tagged Pointers

In this section, we describe how to represent tagged graphs in a list memory. Each memory address of a list memory stores a pair of tagged pointers. We call these pointers CAR and CDR. Each pointer consists of a type tag, a datum and flags (Figure 4.1). We need one type tag for each language construct of JSub. Table 4.1 lists all type tags, we need to represent JSub programs and JSub objects. The datum field of a tagged pointer stores a reference to another pair. If a tagged pointer references a primitive value, we store the value directly in the datum field. Some tagged pointers store additional data in their flag field. Pointers, that use the flag field are marked with an asterisk in Table 4.1. In the remainder of this section, we discuss tagged graphs by means of examples. In Appendix B, we describe the tagged graphs for all pointer types.

For better readability, we introduce the box-and-pointer notation for tagged graphs. We represent pairs as double boxes and label the pointers with the type tag and the flags where applicable. As a first example, Figure 4.2 shows the tagged graph for addition
$a + 1$. We use the type tag `add` for a pointer to an addition. We represent and addition with the tag `add` instead of the tag `alu`, because an addition in JavaScript has other semantics than an ALU operation. For an ALU operation, we convert all operands to a number, whereas we convert all operands to a primitive value for an addition and determine the result with the type of the operands. The addition points to a pair holding the left and the right operand. The CAR cell of the reference pair holds a pointer to the symbol $a$ (tag `sym`). The CDR cell holds a tagged pointer with the tag `n` (number) and the value 1 as datum. To keep the graph representation smaller, we directly write 1 instead of a pointer to the value 1.

<table>
<thead>
<tr>
<th>CAR</th>
<th>CAR tag</th>
<th>CAR datum</th>
<th>CAR flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CDR</th>
<th>CDR tag</th>
<th>CDR datum</th>
<th>CDR flags</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: Tagged graphs in a list memory

Figure 4.2: Tagged graph representation for $a + 1$

Figure 4.3 shows the graph representation of an if statement. The CAR cell of the referenced pair contains a pointer to the predicate, an equality check of value of $a$ and number 2. The CDR cell of the referenced pair references another pair, which holds the consequent and the alternative. We augment the pointer in the CDR cell with the tag `p`, which is an abbreviation for `pair`. The consequent and the alternative are both an assignment expression `as`, where we assign number 1 and 2 to $a$, respectively.

```plaintext
1 if(a == 2) {
2     a = 1;
3 } else{
4     a = 2;
5 }
```

Figure 4.3: Tagged graph representation for if statements
<table>
<thead>
<tr>
<th>tag</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>null pointer</td>
</tr>
<tr>
<td>add</td>
<td>add expression</td>
</tr>
<tr>
<td>alu</td>
<td>ALU operation</td>
</tr>
<tr>
<td>as</td>
<td>assign expression</td>
</tr>
<tr>
<td>b</td>
<td>Boolean</td>
</tr>
<tr>
<td>bi</td>
<td>binding</td>
</tr>
<tr>
<td>br</td>
<td>break statement</td>
</tr>
<tr>
<td>call</td>
<td>function call</td>
</tr>
<tr>
<td>co</td>
<td>continue statement</td>
</tr>
<tr>
<td>del</td>
<td>delete expression</td>
</tr>
<tr>
<td>env</td>
<td>environment</td>
</tr>
<tr>
<td>eq</td>
<td>equals expression</td>
</tr>
<tr>
<td>f</td>
<td>function</td>
</tr>
<tr>
<td>fd</td>
<td>function declaration</td>
</tr>
<tr>
<td>fe</td>
<td>function expression</td>
</tr>
<tr>
<td>gc</td>
<td>garbage collected pointer</td>
</tr>
<tr>
<td>if</td>
<td>if statement</td>
</tr>
<tr>
<td>in</td>
<td>in expression</td>
</tr>
<tr>
<td>inst</td>
<td>instanceof expression</td>
</tr>
<tr>
<td>internal</td>
<td>pointer to controller state</td>
</tr>
<tr>
<td>log</td>
<td>logical expression</td>
</tr>
<tr>
<td>la</td>
<td>labeled statement</td>
</tr>
<tr>
<td>lt</td>
<td>less than expression</td>
</tr>
<tr>
<td>mem</td>
<td>member expression</td>
</tr>
<tr>
<td>mem_{ref}</td>
<td>member reference</td>
</tr>
<tr>
<td>n</td>
<td>number</td>
</tr>
<tr>
<td>new</td>
<td>create new object</td>
</tr>
<tr>
<td>nl</td>
<td>null</td>
</tr>
<tr>
<td>o</td>
<td>object</td>
</tr>
<tr>
<td>op</td>
<td>object pair</td>
</tr>
<tr>
<td>p</td>
<td>pair</td>
</tr>
<tr>
<td>pr</td>
<td>property</td>
</tr>
<tr>
<td>ref</td>
<td>reference</td>
</tr>
<tr>
<td>ret</td>
<td>return statement</td>
</tr>
<tr>
<td>s</td>
<td>string</td>
</tr>
<tr>
<td>sl</td>
<td>statement list</td>
</tr>
<tr>
<td>sym</td>
<td>symbol</td>
</tr>
<tr>
<td>sym_{ref}</td>
<td>symbol reference</td>
</tr>
<tr>
<td>th</td>
<td>throw statement</td>
</tr>
<tr>
<td>this</td>
<td>this expression</td>
</tr>
<tr>
<td>try</td>
<td>try statement</td>
</tr>
<tr>
<td>type</td>
<td>typeof expression</td>
</tr>
<tr>
<td>u</td>
<td>undefined</td>
</tr>
<tr>
<td>vd</td>
<td>variable declaration</td>
</tr>
</tbody>
</table>

Table 4.1: Tags for representing JavaScript programs and JavaScript objects
4.2 JSub Program Representation

In this section, we show how to represent a complete JavaScript program as tagged graph in a list memory. Figure 4.4 shows the tagged graph of a simple JavaScript program.

```
1 var x;
2 x = new Object();
3 x["a"] = 1;
```

![Figure 4.4: Tagged graph representation of a JavaScript program](image)

The code sample in Figure 4.4 consists of three statements. The tagged graph represents these statements in a statement list (tag `sl`). We represent each node of the list with a pair. The CAR cell of a list node points to a statement, and the CDR cell points to the next list node. This enables us to execute all three statements sequentially. The CDR cell of the last list node contains a null pointer (`⊥`). The first statement in the statement list is a variable declaration (`vd`) for the name `x`. The other two statements are assignment expressions (`as`). An assignment expression points to a pair with the left hand side and the right hand side. The left hand side is always a reference to a symbol (`sym_ref`) or a reference to a member of an object (`mem_ref`).

A `new` pointer indicates a new expression and references a pair with the constructor and the arguments. Figure 4.4 references the constructor with the symbol `Object`. The arguments are a pair holding the number of arguments and the list of arguments. In Figure 4.4, the argument list is empty (`⊥`) because we call the `Object` constructor without any arguments.

A pointer to a member expression (`mem_ref`) references a pair with the object and the name of the property. In our example, the object is referenced with the symbol `x` and the name is the string "a".
The difference between the \textit{sym} and \textit{sym}_{\textit{ref}} is that \textit{sym} determines the value of a symbol, whereas \textit{sym}_{\textit{ref}} determines a reference to the symbol. The same difference applies to \textit{mem} and \textit{mem}_{\textit{ref}}.

Before mapping a tagged graph into list memory, we flatten the graph. Figure 4.5 shows a flattened version of the graph of the JavaScript program in Figure 4.4. The pointers in the box-and-pointer notation of the tagged graph become cell addresses.

<table>
<thead>
<tr>
<th>car</th>
<th>cdr</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{cd} \text{'x'}</td>
<td>\text{\textit{as} 2}</td>
</tr>
<tr>
<td>\textit{sl 1}</td>
<td>\textit{sl 5}</td>
</tr>
</tbody>
</table>

Figure 4.5: JavaScript program in a list memory

### 4.3 JSub Object Representation

We represent JSub objects analogously to JSub programs, with tagged graphs. Figure 4.6 shows a JavaScript program which instantiates an object \textit{x}, and how we represent object \textit{x} as tagged graph.

```
1 var x;
2 x = \textbf{new} Object();
3 x["a"] = 1;
```

Figure 4.6: Tagged graph representation for JavaScript objects

The pointer to the object (\textit{o}) references a pair where the CAR cell points to the list of properties. The CDR cell references another pair, that holds the internal property prototype and an additional value. The type of the pointer in CDR cell is \textit{op} (object pair). Pointers with the type tag \textit{op}, store additional information in their five flags ("00001"). The first four flags encode the class name of the object, which is "Object" if all are set to false. We can encode the class name with flags, because JavaScript has a fixed set of 12 different class names. The last flag in the \textit{op} pointer encodes the
extensibility of the object. In our case, we can add further properties to the object \( x \). The internal property prototype points to the Object prototype object (OPO, see Section 2.2). We do not need the additional value in this example. Therefore we set it to \( \bot \).

We encode the properties of an object with a list. The list consists of pairs, where the CAR cell holds a pointer to a property \( (pr) \) and the CDR cell points to the next pair in the list \( (p) \). In our case, the list contains only one property. Therefore the CDR cell of the first list element is \( \bot \). A \( pr \) pointer encodes the three flags \textit{writable}, \textit{enumerable} and \textit{configurable}, which are all set to true in our case. The pair, referenced by the \( pr \) pointer, stores the name of the property and the current value.

We decided to encode flags in the pointers to save memory. Otherwise we would need additional pairs to represent the class name, the extensibility of the object and the flags for properties. Each of these pairs would only store a few bits, but occupy a complete memory cell in memory.

**Summary**

We specify the machine language of a hardware interpreter for JSub. We represent JSub programs and JSub objects as tagged graphs in a list memory. Because of the common representation for JSub programs and JSub objects, we can use the same memory routines to traverse programs, access objects, and perform variable lookup. Furthermore, garbage collection is automatically applicable to JSub programs.

We have not discussed lexical addressing in this chapter. Lexical addressing is more efficient for variable access than lists, but demands a different representation. We aim for a functional and simple representation in this chapter, but we discuss lexical addressing in Chapter 7.
Chapter 5

Microarchitecture

In this chapter, we describe the microarchitecture of our hardware interpreter for JavaScript. The microarchitecture is based on the hardware interpreter for Scheme/Lisp, developed Gerald J. Sussman and Guy L. Steele [10] [13]. We extend their design to handle the language constructs of our JavaScript subset including JavaScript objects. In Section 5.1, we describe the datapath of our hardware interpreter. The control unit is a finite state machine that implements our formal semantics of JSub and controls the register transfers in the datapath. We show how we can generate the controller FSM from our formal semantics of JavaScript and demonstrate parts of the control unit with program examples in Section 5.2. In section 5.3, we show how we use the semantic framework $\mathcal{K}$ to implement a software emulator of our hardware interpreter.

5.1 Datapath

In this section, we describe the datapath of our hardware interpreter. The datapath consists of 8 special purpose registers, 5 stacks, and a list memory that holds the JavaScript program and JavaScript objects. Each register stores one tagged pointer. All components of the datapath are connected via a bus. The control unit coordinates the data transfers between registers, stacks, and list memory. Figure 5.1 shows the datapath of the hardware interpreter. In the remainder of this section, we discuss parts of the datapath in detail.

Some registers of the datapath are special purpose registers when evaluating a JavaScript program. Register EXP holds a tagged pointer to the expression we want to evaluate. Depending on the type tag, the control unit transitions into the next controller state. The register THIS holds a pointer to the this value. Register ENV references the environment. An environment stores bindings for variable names. After evaluating the
expression in EXP, register VAL contains the result. We use registers ARGS, UNEV, PROC and A as general purpose registers.

The stacks store intermediate values of the recursive evaluation of JavaScript programs. This differs to RISC architectures, where we execute programs iteratively with a program counter. The data stack (DStack) saves tagged pointers and the continuation stack (CStack) stores the controller state we visit after returning from a recursive evaluation. The other stacks are used to store values for function calls (CallStack), exceptions (XStack) and labels (LabelStack), that we access with break and continue statements.

The datapath supports type conversions of the primitive data types null, undefined, Number, String and Boolean. For example, the operation Boolean(pv) converts the primitive value pv into a Boolean value. Furthermore, the datapath provides three memory operations. The operations car(p) and cdr(p) access the CAR and the CDR cell of the pair, referenced by the tagged pointer p, respectively. The memory operation cons(p_1, p_2, t, f) constructs a new pair with p_1 and p_2 as CAR and CDR, and returns the a tagged pointer with type tag t and flags f.

5.2 Control Unit

In this section, we describe the control unit of our hardware interpreter. We translate the big-step SOS rules of formal semantics of our subset of JavaScript into a finite state machine, the control unit. This translation is almost straightforward, because big-step SOS rules model interpreter that evaluates JavaScript programs recursively. In the remainder of this section, we show parts of the FSM and demonstrate how we
can evaluate JavaScript programs on our hardware interpreter. Appendix C contains the complete control unit.

Figure 5.2 shows a small part of the FSM with start state S0. The next state depends on the type tag of the pointer in EXP. When we evaluate a literal, i.e. a pointer with the type tag nl (null), u (undefined), n (number), b (Boolean), s (string), or o (object), we copy EXP to VAL and continue with the state at the top of the continuation stack. The evaluation of a this expression in EXP is similar, except that we copy the tagged pointer of register THIS into register VAL.

Figure 5.3 shows the evaluation of a statement list. A pointer with type tag sl refers to a statement list, a pair CAR and CDR hold a pointer to statements s1 and s2. When the tagged pointer in EXP has tag sl, we push s2 on the data stack, to evaluate it later. Furthermore, we remember the S1 on the continuation stack. We store statement s2 in register EXP with the statement s1 and return to state S0 to evaluate it. After evaluating statement s1, the FSM returns to state S1. The FSM pops s2 from the data stack, stores it in EXP and continues with state S0 to evaluate s2.

We demonstrate how to transform the big-step SOS rule for evaluating a member expression $e_1[e_2]$ into the FSM in Figure 5.4. This rule is described in detail in Section 3.3. To evaluate a member expression $e_1[e_2]$, we first evaluate $e_1$ and $e_2$ first. This is similar
to the evaluation of a statement list, except that we store the result of evaluating $e_1$, $r_1$, on the data stack before we evaluate $e_2$ to $r_2$. Furthermore, we store state **S15** on the continuation stack. In the big-step SOS rule for evaluating a member expression, we call the auxiliary function *GetValue* to dereference a reference. This call is not necessary in the FSM, because we use pointers with the tags *sym* and *mem* instead of *sym_{ref}* and *mem_{ref}* if we want the dereferenced value (see Section 4.2). After evaluation of $e_1$ and $e_2$, $r_1$ is pushed onto the data stack and the $r_2$ is stored in VAL.

State **S15** pops $r_1$ from the data stack, transfers it to EXP, and check the type tag. If $r_1$ is undefined or null, we throw a TypeError exception. Otherwise, we push $r_1$ back to the data stack, push state **S16** onto the continuation stack and continue in state **toString**. State **toString** converts the value $r_2$ in register VAL into string $s$ and returns to state **S16**. State **S16** creates a *ref* pointer to a pair with $s$ in CAR and $r_1$ in CDR. With \( \text{cons}(p_1, p_2, t) \), the FSM creates a new pair with $p_1$ in CAR and $p_2$ in

---

**Figure 5.4: FSM for evaluating member expressions**

- **S0**: dispatch tag(EXP)
  - **mem_{ref}**: push(S15, CStack)
  - push(cdr(EXP), Stack)
  - EXP = car(EXP)
  - push(EvalOpd2, CStack)
  - next-state = S0

- **S15: Member Expression**
  - EXP = pop(Stack)
  - tag(EXP) = nl, u
  - next-state = throwTypeError
  - tag(EXP) \neq nl, u
    - push(EXP, Stack)
    - push(S16, CStack)
    - next-state = toString

- **S16: Member Expression**
  - VAL = cons(VAL, pop(Stack), ref)
  - next-state = pop(CStack)

- **toString**: dispatch tag(VAL)
  - S
  - next-state = pop(CStack)
CDR in the list memory, and returns a pointer with type tag $t$ to the pair. We store this pointer in register VAL and with the next state on the continuation stack.

### 5.3 Implementation in $\mathcal{K}$

We use the $\mathcal{K}$ framework, see Section 3.2 to specify a software emulator of our hardware interpreter. We use the software emulator to check the functionality of our JavaScript interpreter before we implement it in hardware. $\mathcal{K}$ configurations represent the datapath and $\mathcal{K}$ rules to describe the control unit.

Figure 5.5 shows a small part of the $\mathcal{K}$ configuration for the software emulator of our hardware interpreter. The first cell of the configuration represents the register EXP. We represent tagged pointer as pairs of the tag and the datum. In this example, EXP holds a tagged pointer with the tag $\bot$. We implement a stack with two cells. One cell contains the memory, which is represented as a List. The other cell of the stack holds the stack pointer and is used to determine the stack size. When push an element on the stack, we add it to the start of the list and increase the stack pointer. When we pop an element from the stack, we remove the first element from the list and decrease the stack pointer.

```plaintext
1 <EXP> (@nil, 0) </EXP>
2 <stack>
3   <mem> .List </mem>
4     <SP> 0 </SP>
5 </stack>
```

Figure 5.5: $\mathcal{K}$ configuration for the hardware interpreter

We use $\mathcal{K}$ rules to implement state transitions in the FSM of the controls unit. The advantage of $\mathcal{K}$ rules is, that we can omit all cells of the configuration, that we do not need in the current rule. Figure 5.6 shows how to implement the FSM of the control unit with $\mathcal{K}$ rules. When we evaluate a number in EXP, we copy EXP to VAL and pop the next state from the continuation stack. The rewrite rule overrides the value in VAL with the value of EXP. Furthermore, we delete the top of the continuation stack (C) and decrease the stack pointer by one. Finally, we rewrite the state S0 to C.

Before we can execute JavaScript programs on our hardware interpreter, we have to translate them into tagged graphs. We also use $\mathcal{K}$ to describe this assembler. We start with the same syntax specification as for our big-step SOS rules and use the rewrite capabilities of $\mathcal{K}$ to translate JavaScript programs into tagged graphs. Figure 5.7 shows
Figure 5.6: \( \mathbb{K} \) rules for the hardware interpreter

how we store pairs in a \( \mathbb{K} \) configuration. Our memory is a collection of pairs. Each pair consists of its location (the memory address) and two cells for CAR and CDR.

Figure 5.7: \( \mathbb{K} \) configuration for representing pairs

Summary

We choose a simple bus-based datapath in our hardware interpreter because functionality is most important for our first prototype. The datapath supports arithmetic operations, type conversions of primitive data types, memory operations, and garbage collection. The data transfers on the datapath are controlled by the control unit. The control unit is a manual translation of the big-step SOS rules of JSub into a FSM. We specify the datapath and the control unit of the hardware interpreter in \( \mathbb{K} \) to create a software emulator, \( \epsilon \)JSub. The input for \( \epsilon \)JSub is a tagged graph, in contrast to \( \sigma \)JSub, which interprets a JSub source program. In chapter 6, we show how to transliterate \( \epsilon \)JSub into a Verilog design to get a hardware implementation.
We implement a prototype of the hardware interpreter on programmable hardware, an Altera Stratix IV FPGA. We call this prototype \( \mu \)JSub. We implement the prototype to prove functionality, but know that we have to apply optimization to be competitive to software interpreters. The JavaScript interpreter on the FPGA communicates with a runtime environment on the host machine over a PCIe interface. We use DMA to load programs into the list memory on the FPGA and programmed I/O to communicate with the JavaScript interpreter during program execution. The runtime environment on the host system assembles JSub programs into tagged graphs and loads them into the list memory of the interpreter. Furthermore, the runtime environment provides system calls to the JavaScript interpreter because we do not implement all operations in hardware. Figure 6.1 shows a block diagram of our prototype. In Section 6.1, we describe the bit format of tagged pointers for our prototype. We describe the hardware design on the FPGA, and the runtime environment in Sections 6.2 and 6.3.

![Figure 6.1: Prototype of a hardware interpreter for JavaScript](image-url)
6.1 Bit Format of Tagged Pointers

In this section, we describe the bit format for tagged pointers in our prototype. We use a 64 bit representation for tagged pointers to simplify the communication over the PCIe interface with the runtime environment. JavaScript represents numbers as double precision floating-point numbers, defined in the standard IEEE 754 [6] (figure 6.2). The IEEE 754 standard specifies multiple bit patterns that represent NaN values. Different NaN values are not distinguishable in JavaScript. Therefore, we can use a subset of the NaN representations, defined in the standard IEEE 754, to encode our tagged pointers.

Each double precision floating point number that represents NaN has an exponent of 2047 and a mantissa not equal to zero. We set the most significant bit in the mantissa of a NaN value to indicate a tagged pointer and use the remaining 51 bits of the mantissa to encode it (Figure 6.3). As described in Section 4.1, a tagged pointer consists of three fields. We use 6 bits to encode the the type tag, because we need 44 different types to represent all language constructs of JavaScript and the JavaScript object model. We use 5 bits to store flag bits, because a tagged pointer encodes at maximum 5 flags. The remaining 40 bits store the datum of the pointer.

6.2 FPGA implementation

In this section, we discuss the implementation of our hardware interpreter on an Altera DE4 Development and Education board with a Altera Stratix IV FPGA. The JavaScript interpreter communicates with the runtime environment on the host machine over an interrupt register, connected to the PCIe interface. In the remainder of this section, we describe the components on the FPGA in detail.
We decide to implement string operations and number to string conversion in the runtime environment to reduce the effort of implementation. The JavaScript interpreter represents strings as a tagged pointer with the type tag \( s \). The datum of the pointer stores the index of the string in a symbol table on the runtime environment on the host machine. The JavaScript interpreter compares the indices of two strings to perform an equality check, because equal strings are always represented with the same index. We need a system call to the runtime environment for all other string operations. To perform a system call, the JavaScript interpreter writes a bit pattern, representing the requested function into the interrupt register and provides the operands in registers which can be accessed by the runtime environment. The runtime environment system performs the requested operation, writes the result back to the FPGA and clears the interrupt register. On average, a system call to the runtime environment causes 1200 idle clock cycles on our JavaScript interpreter.

We implement the list memory (see Section 4.1) of the JavaScript interpreter with the memory cells of the FPGA to reach faster memory access than on external memory. On our FPGA, this limits the memory size to 256KByte. We implement a stop-and-copy garbage [8] collector to reclaim unreferenced pairs in the list memory. In future implementations we want to use external memory to store JavaScript programs and JavaScript objects, and use the memory cells on the FPGA for caching. We implement the list memory memory as interleaved memory with two banks to access CAR and CDR of a pair concurrently. The memory has a delay of one clock cycle for read and write operations.

Our prototype represents all numbers as double precision floating-point numbers. We implement floating point operations with IP cores provided by Altera. The IP cores for floating point operations require multiple clock cycles to compute the result. For bitwise operations, JavaScript requires a conversion from floating point numbers to 32 bit integers. We use Altera IP cores to convert floating point number to integer numbers and vice versa. Table 6.1 shows the delay for floating point operations in clock cycles.

<table>
<thead>
<tr>
<th>operation</th>
<th>delay [clock cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mult</td>
<td>5</td>
</tr>
<tr>
<td>div</td>
<td>10</td>
</tr>
<tr>
<td>add</td>
<td>7</td>
</tr>
<tr>
<td>sub</td>
<td>7</td>
</tr>
<tr>
<td>compare</td>
<td>3</td>
</tr>
<tr>
<td>convert</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.1: Delay of Floating Point Operations
We choose a bus-based datapath (see Section 5.1) for our prototype on the FPGA. This datapath is not optimal, because the control unit can only perform one register transfer in one clock cycle. For example, the interleaved memory reads CAR and CDR of one pair in the list memory concurrently, but the control unit writes CAR and CDR into two different registers, it needs two clock cycles because the datapath has only one bus. With a second bus, the control unit could update both registers concurrently.

In this chapter, we described the latency in our prototype by means of clock cycles. We provide the same clock frequency for all components on the FPGA. Therefore, we are limited 125MHz, the clock frequency of the PCIe interface. In future implementations, we want to use multiple clock domains to increase the clock frequency of the JavaScript interpreter.

6.3 Runtime Environment

We need a runtime environment on the host machine for our hardware interpreter. The runtime environment translates JavaScript a program into a tagged graph and loads it into the list memory on the FPGA via the PCIe interface. Furthermore, the runtime environment provides system calls to the JavaScript interpreter and stores a symbol table with the strings of the program.

The runtime environment contains an assembler, that translates JavaScript a program into a tagged graph. The assembler writes the tagged graph directly to memory and does not generate an intermediate representation. An intermediate representation is not necessary, because the assembler does not perform any optimization of the JavaScript programs. Furthermore, the assembler generates a symbol table with all strings in the source code and labels them with a unique reference. We use these references to represent strings as tagged pointer in the JavaScript interpreter.

The Altera University Program [1] provides access to a C driver for the PCIe interface of the FPGA. The loader of the runtime environment uses this driver to load the tagged graph into the list memory on the FPGA with direct memory access (DMA). We use DMA to reduce the effort of the processor on the host machine.

During program execution on the JavaScript interpreter, the runtime environment on the host machine polls the value of the interrupt register on the FPGA. We use programmed I/O to read the value of the interrupt register, because it has only a size of one byte. The JavaScript interpreter changes the value of the interrupt register to request a system call. The runtime environment provides system calls for string op-
erations, number to string conversions and I/O. The runtime environment reads the operands of the system call from the FPGA and writes the result back to the FPGA. The operands and the result are tagged pointers. If the result of a system call is a string, we perform a lookup in the symbol table. If the symbol table already contains the result string, we return its index. Otherwise, we insert the result string into the symbol table and generate a new index. As last step, the runtime environment clears the interrupt register to transfer the control back to the JavaScript interpreter.

Summary

We implement a functional prototype, $\mu$JSub, that executes JSub program on an FPGA. $\mu$JSub consists of a JSub interpreter on the FPGA, a runtime environment on the host machine, and a PCIe interface as communication layer. We transilterate $\epsilon$JSub into a Verilog implementation of the JSub interpreter. For reasons of time, we did not implement string operations in hardware, but we provide them as system calls to the runtime environment on the host machine. We provide these system calls to demonstrate the functionality of $\mu$JSub. We know, that string operations in hardware would be more efficient and analyze their impact on the execution time in Chapter 7.
Chapter 7

Optimizations

Our prototype of a hardware interpreter for JavaScript, $\mu$JSub (Chapter 6), is not optimized in terms of performance. We investigate the optimization potential of $\mu$JSub with benchmark programs from Google’s Octane benchmark suite [3], Mozilla’s SunSpider [4] benchmark suite and the Computer Languages Benchmarks Game [2]. The implementation of the optimizations is beyond the scope of this thesis, because we estimate 6 man-month as development cost. However, we estimate the optimization potential of $\mu$JSub by counting clock cycles. We determine which states of the control unit of $\mu$JSub would be removed by the optimizations and accumulate their frequency while program execution to estimate the optimization potential. In this chapter, we discuss optimizations for $\mu$JSub regarding:

- Lexical addressing
- Optimizations for ALU operations
- Hardware Support for strings
- Additional buses in the datapath
- Hardware support for loops
- Property lookup

Table 7.1 shows the execution time of our benchmark programs on $\mu$JSub and the estimated execution time of $\mu$JSub, assuming that all optimizations discussed in this chapter were implemented. We estimate to reduce the execution time of our benchmark programs on $\mu$JSub by a factor of 11.8 if we implement the optimizations.
## Optimizations

Table 7.1: Estimated execution time of µJSub if all optimizations were implemented (µJSub opt)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>253.9</td>
<td>16.0</td>
<td>237.9</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>152.0</td>
<td>11.5</td>
<td>140.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>159.4</td>
<td>17.4</td>
<td>139.3</td>
</tr>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>58.2</td>
<td>17.0</td>
<td>41.2</td>
</tr>
<tr>
<td>SunSpider</td>
<td>controlflow-recursive.js</td>
<td>78.0</td>
<td>15.7</td>
<td>62.3</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>33.1</td>
<td>694.3</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js</td>
<td>385.2</td>
<td>16.8</td>
<td>368.4</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>8.1</td>
<td>25.9</td>
</tr>
</tbody>
</table>

Table 7.2 shows the execution time of our benchmark programs on the software interpreters V8 (version 3.31.1) and SpiderMonkey (version JavaScript-C40.0a1) in clock cycles. We execute the software interpreters on an Intel(R) Core(TM) i7-3667U CPU @ 2.00GHz, and 8GByte DDR3 RAM. The operating system of the PC is Ubuntu 14.04 with the Linux kernel version 3.13. During the execution of JavaScript programs, the CPU has run with a clock frequency of 2.50GHz.

With our estimated performance improvement, µJSub executes the benchmark programs faster than V8 and SpiderMonkey by a factor of 2.65 on average.

Table 7.2: Execution time on software interpreters

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>V8 [M cycles]</th>
<th>SpiderMonkey [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>40.0</td>
<td>60.0</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>32.5</td>
<td>57.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>32.5</td>
<td>55.0</td>
</tr>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>37.5</td>
<td>65.0</td>
</tr>
<tr>
<td>SunSpider</td>
<td>controlflow-recursive.js</td>
<td>27.5</td>
<td>42.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>32.5</td>
<td>40.0</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js (n=100)</td>
<td>32.5</td>
<td>42.5</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>47.5</td>
<td>75.0</td>
</tr>
</tbody>
</table>

### 7.1 Lexical Addressing

In this section, we describe lexical addressing for variable lookup. Lexical addressing is possible for JavaScript variables, because JavaScript is statically scoped, i.e. the offsets of variables in their environments are known at compile time. Table 7.3 shows the estimated performance improvement of µJSub, by implementing lexical addressing. In the
In the remainder of this section, we explain why lexical addressing improves the performance of \( \mu \text{JSub} \).

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>( \mu \text{JSub} ) [M cycles]</th>
<th>( \mu \text{JSub la} ) [M cycles]</th>
<th>( \Delta ) [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>253.9</td>
<td>152.8</td>
<td>105.4</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>152.0</td>
<td>91.2</td>
<td>60.8</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>159.4</td>
<td>98.3</td>
<td>63.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>58.2</td>
<td>35.2</td>
<td>23.0</td>
</tr>
<tr>
<td>SunSpider</td>
<td>controlflow-recursive.js</td>
<td>78.0</td>
<td>41.1</td>
<td>36.9</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>496.1</td>
<td>234.6</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js</td>
<td>385.2</td>
<td>117.3</td>
<td>267.9</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>30.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 7.3: Estimated execution of \( \mu \text{JSub} \) with lexical addressing (\( \mu \text{JSub la} \))

\( \mu \text{JSub} \) stores variables in environments. Each environment stores a list of bindings and a reference to its outer environment. For variable lookup, we traverse these lists of bindings. Figure 7.1 shows a JavaScript sample program and the simplified structure of the environment while evaluating \( a \) inside the function body of \( f \). The lookup of \( a \) requires 10 memory accesses, because we have to traverse two complete environments until we find the binding for \( a \).

![Figure 7.1: Tagged graph for environments in \( \mu \text{JSub} \)](image)

A lexical address is a pair of numbers. The first number determines the offset in the list of environments, and the second number determines the offset inside the environment. If we consider the example in Figure 7.1, the variables \( a \), \( b \) and \( c \) have the lexical addresses \( (1,1) \), \( (0,1) \) and \( (0,0) \) inside the function body of \( f \). With lexical addressing, there is no need to store the variable names. We only store the bindings in an array like structure without pointers. Figure 7.2 shows the representation for environments, when using lexical addressing. An \( env \) pointer references a pair which holds a reference to the
outer environment in CAR and the number of bindings in CDR. The number of bindings is needed for garbage collection, because we store the bindings in consecutive pairs of the list memory without referencing them. With this representation of environments, we access $a$ inside the function body of $f$ with $cdr(car(ENV) + 1)$, where ENV holds a pointer to the environment.

![Figure 7.2: Tagged graph representation for environments with lexical addressing](image)

Besides the faster variable lookup, lexical addressing also reduces the number of garbage collector runs, because $\mu$JS creates a new pair of type reference, each time we change the value of a variable. With lexical addressing, we avoid the creation of these new pairs, because a lexical address is already a reference. For example, the program bitops-bitwise-and.js from the SunSpider benchmark suite causes 352 garbage collector runs on $\mu$JS. Lexical addressing reduces the number of garbage collector runs to zero.

### 7.2 Optimizations for ALU operations

We present three optimizations for ALU operations. These optimizations include the number representation, the removal of unnecessary type conversions and ALU operations with one constant operand. Table 7.4 shows the estimated performance improvement of $\mu$JS by optimizing ALU operations. In the remainder of this section, we discuss the optimizations for ALU operations in detail.

$\mu$JS stores all numbers as double precision floating point numbers (standard IEEE 754 [6]). Floating point operations are quite costly on $\mu$JS, because they require multiple clock cycles (Table 6.1). Furthermore, the ALU of $\mu$JS converts floating point numbers to 32 bit integers whenever it performs bitwise operations or shift operations. After calculating the result, the ALU converts the result back to a floating point number. Each conversion requires 6 clock cycles. We introduce a new type tag for 32 bit integers to optimize $\mu$JS. This is possible without changing the bit format of tagged pointers, because the datum field has a bit width of 40 bits (see Section 6.1). We store all numbers as 32 bit integers, if applicable. This reduces the execution time of JavaScript...
### Optimizations

#### Table 7.4: Estimated execution time of $\mu$JSub with optimized ALU operations ($\mu$JSub alu)

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>$\mu$JSub [M cycles]</th>
<th>$\mu$JSub alu [M cycles]</th>
<th>$\Delta$ [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>253.9</td>
<td>177.5</td>
<td>76.4</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>152.0</td>
<td>114.8</td>
<td>37.2</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>159.4</td>
<td>105.7</td>
<td>53.7</td>
</tr>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>58.2</td>
<td>52.5</td>
<td>5.7</td>
</tr>
<tr>
<td>SunSpider</td>
<td>controlflow-recursive.js</td>
<td>78.0</td>
<td>66.2</td>
<td>11.8</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>665.4</td>
<td>62.0</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js</td>
<td>385.2</td>
<td>325.5</td>
<td>59.7</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>31.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Before the ALU performs an operation, $\mu$JSub converts all operands to numbers, even if the operands are already numbers. Each type conversion causes a delay of 5 clock cycles. We can avoid these unnecessary type conversions with more branches in the control unit.

In our benchmark programs, we discovered many ALU operations with one constant operand. $\mu$JSub evaluates all operands before we perform an ALU operation. The evaluation of a constant requires two clock cycles in $\mu$JSub. We introduce two new type tags, one for binary ALU operation with a constant as first operand, and one for an binary ALU operation with a constant as second operand. With this additional type tags, we save two clock cycles for each binary ALU operation with one constant operand, because we omit the evaluation of the constant and pass it directly to the ALU.

#### 7.3 Hardware Support for Strings

$\mu$JSub performs a system call to the runtime environment on the host machine to request string operations or a number to string conversion (see Section 6.2). We can increase the performance of $\mu$JSub if we perform these operations in hardware, because each system call has an average delay of 1200 clock cycles.

In the programs richards.js from Google’s Octane benchmark suite and math-cordic.js from Mozilla’s SunSpider benchmark suite, $\mu$JSub performs number to string conversions, because we access properties of objects with numeric indices. Each number to
string conversion causes a system call with an average delay of 1200 clock cycles. Table 7.5 shows the estimated performance improvement of $\mu$JSub by adding hardware support for strings.

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>$\mu$JSub [M cycles]</th>
<th>$\mu$JSub hws [M cycles]</th>
<th>$\Delta$ [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>368.5</td>
<td>358.9</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>27.8</td>
<td>6.2</td>
</tr>
</tbody>
</table>

Table 7.5: Estimated execution time of $\mu$JSub with hardware support for strings ($\mu$JSub hws)

### 7.4 Additional Buses in the Datapath

In this section, we discuss optimizations in the datapath of our prototype. $\mu$JSub performs only one register transfer per clock cycle, because all components are connected with a single bus. With additional connections between the registers and the stacks, $\mu$JSub could perform multiple register transfers concurrently. Table 7.6 shows the estimated performance improvement of $\mu$JSub, if we added buses buses to its datapath.

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>$\mu$JSub [M cycles]</th>
<th>$\mu$JSub buses [M cycles]</th>
<th>$\Delta$ [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>253.9</td>
<td>226.0</td>
<td>27.9</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>152.0</td>
<td>130.4</td>
<td>21.6</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>159.4</td>
<td>141.7</td>
<td>17.7</td>
</tr>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>58.2</td>
<td>45.7</td>
<td>12.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>controlflow-recursive.js</td>
<td>78.0</td>
<td>64.5</td>
<td>13.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>700.1</td>
<td>27.3</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js</td>
<td>385.2</td>
<td>362.5</td>
<td>22.7</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>27.0</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Table 7.6: Estimated execution time of $\mu$JSub with additional buses in the datapath ($\mu$JSub buses)

$\mu$JSub needs three clock cycles to construct a new pair, because all components are connected with a single bus. Consider the state S16 of the control unit in Figure 5.4. State S16 creates a new pair of with VAL as CAR and the top of the data stack as CDR, and store the ref pointer to this pair in VAL. $\mu$JSub needs two clock cycles to transfer VAL and the top of the data stack to the list memory, because both transfers occupy the bus for one clock cycle. In the third clock cycle, we transfer the ref pointer to VAL over the bus. If we add direct connections to the datapath, from VAL and from the data stack to the list memory, we can perform all three register transfers...
concurrently and save two clock cycles. We analyzed the control unit to identify useful additional connections in the datapath and estimate the performance improvement of \( \mu \text{JSub} \).

### 7.5 Hardware Support for Loops

We add hardware support for loops to our datapath, to speed up simple loops. We consider a loop to be simple, if it has one loop counter, the loop conditions is a comparison to a constant, and a constant increment of the loop counter. Table 7.7 show the estimated performance improvement of \( \mu \text{JSub} \) by adding hardware support for loops.

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>( \mu \text{JSub} ) [M cycles]</th>
<th>( \mu \text{JSub} ) loop [M cycles]</th>
<th>( \Delta ) [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>bitops-bits-in-byte.js</td>
<td>253.9</td>
<td>225.7</td>
<td>28.2</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-bitwise-and.js</td>
<td>152.0</td>
<td>131.0</td>
<td>21.0</td>
</tr>
<tr>
<td>SunSpider</td>
<td>bitops-3bit-bits-in-byte.js</td>
<td>159.4</td>
<td>154.9</td>
<td>4.5</td>
</tr>
<tr>
<td>SunSpider</td>
<td>math-cordic.js</td>
<td>727.4</td>
<td>716.0</td>
<td>11.4</td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>mandelbrot.js</td>
<td>385.2</td>
<td>367.2</td>
<td>18.0</td>
</tr>
</tbody>
</table>

Table 7.7: Estimated execution time of \( \mu \text{JSub} \) with hardware support for loops (\( \mu \text{JSub} \) loop)

Figure 7.3 shows a JavaScript example of a simple loop and its representation as tagged graph. We represent a loop with a cyclic if statement (see Section 4.1). The consequent of the if statement contains a statement list (sl) where the CAR cell contains the loop body and the CDR cell contains a cyclic reference to the if statement.

We add a second ALU and two new registers to our datapath, one for the loop counter and one for the constant in the loop condition. Before we execute a loop body, we load the loop counter and the constant of the loop condition into their registers. While executing the loop body, we can calculate the new value of the loop counter and compare it to the constant with the second ALU. We introduce two new type tags to represent simple loops. We use type tag loop to indicate a tagged pointer to a simple loop and type tag lc to access the loop counter in the loop body. Figure 7.4 shows the tagged graph for a simple loop. The representation of a simple loop is more compact and its execution is faster, because we omit three statements in each loop iteration, one assign expression that updates the loop counter, one if statement and one comparison expression. For each iteration of the loop, we can save 35 clock cycles in our FSM when we add hardware support for simple loops to \( \mu \text{JSub} \).
7.6 Property Lookup

In this section, we describe how to speedup property lookup on $\mu$JSub. $\mu$JSub store properties of objects in lists. The latency of a property lookup depends on the position of the property in the list. We can optimize the property lookup with a move-to-front heuristic (MTF) [9]. MTF changes the list of properties dynamically by moving a property to the start of the list after accessing it.

We analyze the impact of the MTF on the programs richards.js form the Octane benchmark suite and access-binary-trees.js form the SunSpider benchmark suite. Table 7.8 shows the estimated performance improvement of $\mu$JSub, if we implement MTF for property accesses. MTF does not speedup the program access-binary-trees on $\mu$JSub because the properties of each object are only accessed once in the program.
Optimizations

<table>
<thead>
<tr>
<th>benchmark suite</th>
<th>program</th>
<th>(\mu)JSub [M cycles]</th>
<th>(\mu)JSub MTF [M cycles]</th>
<th>(\Delta) [M cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>SunSpider</td>
<td>access-binary-trees.js</td>
<td>58.2</td>
<td>0.0</td>
<td>58.2</td>
</tr>
<tr>
<td>Octane</td>
<td>richards.js</td>
<td>34.0</td>
<td>28.6</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Table 7.8: Estimated execution time of \(\mu\)JSub with move-to-front heuristic for property accesses \((\mu\)JSub MTF\)

**Summary**

We analyze the performance of \(\mu\)JSub with benchmark programs and identify optimizations. The optimizations regard the program representation, the control unit, and the datapath. Their implementation is beyond the scope of this thesis, because from the experience of implementing \(\mu\)JSub, we estimate at least one man-month as development costs for each optimization. However, we can estimate the optimization potential, because we know which states of the FSM are removed by the optimizations. We accumulate the frequency of these states during program execution to estimate the optimization potential accurately. We execute our benchmark programs on V8 and SpiderMonkey and determine their execution time in clock cycles. With the identified optimizations, we expect that \(\mu\)JSub executes our benchmark programs 2.65 times faster than V8 and SpiderMonkey on average.
Chapter 8

Conclusion and Future Work

Hardware interpreters are applicable for dynamic languages to improve the performance, and to reduce the energy consumption. More succinctly, we draw these conclusions:

1. We have implemented $\mu$JSub, a prototype of a hardware interpreter for JavaScript. $\mu$JSub is a functional prototype, but it is not optimized. We identify optimizations regarding the control unit, the program representation, and the datapath to improve the performance of $\mu$JSub. The implementation of these optimizations is beyond the scope of this thesis, because with our experience from the implementation of $\mu$JSub, we estimate at least one man-month as development costs for each optimization. However, we estimate the optimization potential of $\mu$JSub accurately. We know which FSM states are removed by the optimizations and accumulate their frequency during program execution to determine the optimization potential of $\mu$JSub. Based on our estimates, see Chapter 7, $\mu$JSub requires 2.65 times fewer clock cycles to execute JavaScript programs than a software interpreter on a general purpose processor.

2. We can vary the clock frequency of our hardware interpreter to increase performance or to reduce power consumption. When we drive our hardware interpreter with the same clock frequency as a general purpose processor, it executes JavaScript programs 2.65 times faster than a software interpreter. We can reduce the clock frequency of our hardware interpreter to reduce its power consumption, because the clock frequency contributes cubically to the power consumption of a digital circuit. If we drive $\mu$JSub with a clock frequency 2.65 times slower than the clock frequency of a general purpose processor, we still execute JavaScript programs as fast a software interpreter, but we estimate a reduction of the power consumption by one order of magnitude.
3. The software interpreters for JavaScript, V8 and SpiderMonkey disregard the ECMAScript Language Specification [7] in parts. We have specified the semantics of a JavaScript subset, JSub, formally (Chapter 3). We use the semantic framework $\mathbb{K}$ to obtain $\sigma_{\text{JSub}}$, a software interpreter for JSub programs. We execute programs on $\sigma_{\text{JSub}}$, on V8, and on SpiderMonkey and compare the results. We have found discrepancies, which indicates that V8 and SpiderMonkey disregard the ECMAScript Language Specification [7] in parts. Our hardware interpreter $\mu_{\text{JSub}}$ implements the ECMAScript Language Specification [7] faithfully, because we infer its datapath and control unit from the formal semantics of JSub.

4. We describe a design methodology to implement a hardware interpreter for JSub. First, we formalize the semantics of JSub and specify them in the semantic framework $\mathbb{K}$ to obtain $\sigma_{\text{JSub}}$ (see Chapter 3). We infer the datapath and the control unit of the hardware interpreter from the formal description of JSub (Chapter 5). We describe the datapath and the control unit in $\mathbb{K}$ to obtain $\epsilon_{\text{JSub}}$, a software emulator of our hardware interpreter. $\epsilon_{\text{JSub}}$ and $\sigma_{\text{JSub}}$ operate on different abstraction levels. $\epsilon_{\text{JSub}}$ executes JSub programs, represented as tagged graph, whereas $\sigma_{\text{JSub}}$ executes JSub source programs directly. We transliterate $\epsilon_{\text{JSub}}$ into a Verilog implementation, the hardware interpreter $\mu_{\text{JSub}}$. The transliteration into Verilog code is straightforward. We use JavaScript as example to demonstrate our design methodology and we expect that it is also applicable for other dynamic languages, for example Python or Ruby.

5. We use a unified representation, tagged graphs (Chapter 4), for JSub programs and JSub objects. Because such a unified representation exists, we can use the same memory operations for program traversal, object accesses, and variable lookup. Furthermore, garbage collection is automatically applied to JSub program graphs.

Based on our experiments, there are open problems we suggest as future work:

1. We have implemented a preliminary prototype of a hardware interpreter for JavaScript, $\mu_{\text{JSub}}$. We choose the functionality of $\mu_{\text{JSub}}$ in a way, that it is realizable in reasonable time. To create a robust system from our prototype, we suggest to implement the discussed optimizations, to perform power measurement, and to implement a compiler that translates JavaScript programs into JSub programs automatically.

2. We expect that dynamic optimization of the program graph improves the performance of $\mu_{\text{JSub}}$. Because of the unified representation of JSub programs and
JSub objects, we can manipulate JSub programs during program execution to perform optimizations. In Section 7.6, we discuss one dynamic optimization, a move-to-front heuristic to reduce the latency of property lookups. We suggest to investigate further dynamic optimizations in future projects.

3. Technically, hardware interpreters can utilize fine-grained parallelism, which is not possible in software interpreters, because fine-grained parallelism requires changes in the hardware. We expect to improve the performance of μJSub when we add multiple arithmetic units to its datapath and implement parallel execution of program graphs.

4. We suggest to investigate how to automate the design methodology, we used to implement μJSub. We have performed all steps in the design of μJSub manually, but we have not found any reason to assume that it is impossible to automate the design process. We identify three essential steps. It is necessary to find a set of type tags to represent the language constructs and the object model. Furthermore, we have to infer the datapath and the control unit from the formal semantics. As last step, we have to generate Verilog code for the control unit and the datapath.
Appendix  A

Formal Semantics of JavaScript

A.1 Semantic Domains

\[ Store = \text{Loc} \rightarrow (\text{LexicalEnvironment} + \text{Object}) \times \text{Loc} \]

\[ \text{LexicalEnvironment} = (\text{DeclarativeEnvironmentRecord} + \text{ObjectEnvironmentRecord}) \times (\bot + \text{Loc}) \]

\[ \text{ObjectEnvironmentRecord} = \text{Loc} \]

\[ \text{DeclarativeEnvironmentRecord} = \text{Identifier} \rightarrow \text{Binding} \]

\[ \text{Binding} = \text{Value} \times \text{Boolean} \]

\[ \text{Value} = \text{loc of Loc} + \text{PrimitiveValue} \]
**PrimitiveValue** = undefined + null + Boolean + String + Number

**Object** = (String → Property) × (loc(Loc) + null) × String × Boolean × (PrimitiveValue + Function)

**Property** = Value × Boolean × Boolean × Boolean

// (value, writable, enumerable, configurable)

**Function** = Boolean × Body × IL × Loc

**CompletionRecord** = CompletionType × (Value + empty) × (Identifier + empty)

**CompletionType** = normal + break + continue + return + throw

**ExprValue** = val(Value) + ref(ReferenceType) + exc(Value)

**ReferenceType** = Base × String

**BaseValue** = undefined + loc(Loc) + Boolean + String + Number

**AuxiliaryFunction** = GetValue + ToString + ToStringHelper + ToNumber + ToNumberHelper +ToObject
A.2 Judgements

\[<\text{Program}> \Downarrow_0 <\text{Store, CompletionType}>\]

\[<\text{Body, Loc, Value, Store}> \Downarrow_1 <\text{Store, CompletionType}>\]

\[<\text{FD + VD, Loc, Store}> \Downarrow_2 <\text{Store, ExprValue}>\]

\[<\text{S, Loc, Value, Store}> \Downarrow_3 <\text{Store, CompletionRecord}>\]

\[<\text{E, Loc, Value, Store, ExprValue}> \Downarrow_4 <\text{Store, ExprValue}>\]

\[<\text{IL, AL, Loc, Loc, Integer, Store}> \Downarrow_5 <\text{Store, ExprValue}>\]

\[<\text{AuxiliaryFunction, Store, ExprValue}> \Downarrow_6 <\text{Store, ExprValue}>\]

We define the update of a function as \(m[X/v]\) as:

\[
m[X/v](Y) = \begin{cases} 
v & \text{if } X = Y \\
m(Y) & \text{if } X \neq Y \end{cases}
\]
A.2.1 Program

\[
\text{Program} \\
\langle B, GE, \text{val}(GO), \sigma_{\text{init}} \rangle \Downarrow_1 \langle \sigma_1, \text{comp} \rangle \\
\Downarrow_0 \langle \sigma_1, \text{comp} \rangle
\]

\(\sigma_{\text{init}}\) is the initial store which has bindings for the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>Global Object</td>
</tr>
<tr>
<td>GE</td>
<td>Global Environment</td>
</tr>
<tr>
<td>OPO</td>
<td>Object prototype object</td>
</tr>
<tr>
<td>FPO</td>
<td>Function prototype object</td>
</tr>
<tr>
<td>BPO</td>
<td>Boolean prototype object</td>
</tr>
<tr>
<td>SPO</td>
<td>String prototype object</td>
</tr>
<tr>
<td>NPO</td>
<td>Number prototype object</td>
</tr>
<tr>
<td>EPO</td>
<td>Error prototype object</td>
</tr>
<tr>
<td>TEPO</td>
<td>TypeError prototype object</td>
</tr>
<tr>
<td>REPO</td>
<td>ReferenceError prototype object</td>
</tr>
</tbody>
</table>

A.2.2 Body

\[
\text{Body1} \\
\langle FD, env, \sigma \rangle \Downarrow_2 \langle \sigma_1, \text{val}(val_1) \rangle \\
\langle VD, env, \sigma_1 \rangle \Downarrow_2 \langle \sigma_2, \text{val}_2 \rangle \\
\langle S, env, this, \sigma_2 \rangle \Downarrow_3 \langle \sigma_3, \text{comp} \rangle \\
\Downarrow_1 \langle \sigma_3, \text{comp} \rangle
\]

\[
\text{Body2} \\
\langle FD, env, \sigma \rangle \Downarrow_2 \langle \sigma_1, \text{exc}(val_1) \rangle \\
\Downarrow_1 \langle \sigma_3, (\text{throw}, \text{val}_1, \text{empty}) \rangle
\]

A.2.3 Declarations

\[
\text{EmptyDecl} \\
\langle \text{empty}, env, \sigma \rangle \Downarrow_2 \langle \sigma, \text{val}(\text{null}) \rangle
\]

\[
\text{VarDecl1} \\
\sigma = (\text{map}, \text{top}) \\
\text{map}(\text{env}) = (d, \text{outer}) \\
i \notin \text{domain}(d) \\
\text{map}[\text{env}/(d[i/(\text{undefined}, \text{false})], \text{outer})], \text{top}) = \sigma_2 \\
\Downarrow_2 \langle \sigma_3, \text{comp} \rangle
\]

\[
\text{Decl} \\
\langle \text{var } i; VD, env, \sigma \rangle \Downarrow_2 \langle \sigma_3, v \rangle
\]
Formal Semantics of JavaScript

VARDECL2
\[\begin{align*}
\sigma &= (\text{map}, \text{top}) \\
\text{map}(\text{env}) &= (o, \text{outer}) \\
o &\in \text{Loc} \\
\text{HasProperty}(i, o, \sigma) \\
\text{map}(o) &= (m, \text{pr}, \text{cl}, \text{ex}, \text{prim}) \\
m[\text{Str}(i)/\text{(undefined, true, true, false) }] &= m_2 \\
\text{map}[o/(m_2, \text{pr}, \text{cl}, \text{ex}, \text{prim})], \text{top} &= \sigma_2 \\
&< V D, \text{env}, \sigma > \downarrow 2 < \sigma_3, v >
\end{align*}\]

HasProperty(s, o, (map, top))
\[= (s \in \text{domain}(m)) \vee (pr \in \text{Loc} \land \text{HasProperty}(s, pr, (map, top))),\]

where \(m, \text{pr}\) defined by
\[\text{map}[o] = (m, \text{pr}, \text{cl}, \text{ex}, \text{prim})\]

VARDECL3
\[\begin{align*}
\sigma &= (\text{map}, \text{top}) \\
(d, \text{outer}) &= \text{map}(\text{env}) \\
d &\in \text{DeclarativeEnvironmentRecord} \\
i &\in \text{domain}(d) \\
&< V D, \text{env}, \sigma > \downarrow 2 < \sigma_2, v >
\end{align*}\]

VARDECL4
\[\begin{align*}
\text{map}, \text{top} &= \sigma \\
(o, \text{outer}) &= \text{map}(\text{env}) \\
o &\in \text{Loc} \\
\text{HasProperty}(\text{Str}(i), o, \sigma) \\
&< V D, \text{env}, \sigma > \downarrow 2 < \sigma_2, v >
\end{align*}\]

FUNDECL1
\[\begin{align*}
\text{map}, \text{top} &= \sigma \\
(d, \text{outer}) &= \text{map}(\text{env}) \\
d &\in \text{DeclarativeEnvironmentRecord} \\
\text{proto} &= (\{(\text{"constructor"}, (\text{top} + 1, \text{true, true, false}))\}, \text{OPO}, \text{"Object"}, \text{true, null}) \\
m &= \{(\text{"length"}, (\text{Len}(i)), \text{false, false, false}), (\text{"prototype"}, (\text{top}, \text{true, false, false}))\} \\
f &= (m, \text{FPO}, \text{"Function"}, \text{true, (body, il, env, true)}) \\
d_2 &= d[i/(\text{loc}(\text{top} + 1), \text{false})] \\
\sigma_2 &= (\text{map}[\text{top}/\text{proto}, (\text{top} + 1)/f, \text{env}/(d_2, \text{outer})], \text{top} + 2) \\
&< \text{FD, env}, \sigma_2 > \downarrow 2 < \sigma_3, v >
\end{align*}\]

< function i ( il ) {body} \text{FD, env} > \downarrow 2 < \sigma_3, v >

FUNDECL2
\[\begin{align*}
\text{map}, \text{top} &= \sigma \\
(o, \text{outer}) &= \text{map}(\text{env}) \\
o &\in \text{Loc} \\
\text{map}(o) &= (m, \text{pr}, \text{cl}, \text{ex}, \text{prim}) \\
\text{Str}(i) &\notin \text{domain}(m) \\
\text{proto} &= (\{(\text{"constructor"}, (\text{top} + 1, \text{true, true, false}))\}, \text{OPO}, \text{"Object"}, \text{true, null}) \\
m_2 &= \{(\text{"length"}, (\text{Len}(i)), \text{false, false, false}), (\text{"prototype"}, (\text{top}, \text{true, false, false}))\} \\
f &= (m_2, \text{FPO}, \text{"Function"}, \text{true, (body, il, env, true)}) \\
m_3 &= m[\text{Str}(i)/\text{loc}(\text{top} + 1), \text{true, true, false}] \\
\sigma_2 &= (\text{map}[\text{top}/\text{proto}, (\text{top} + 1)/f, o/(m_3, \text{pr}, \text{cl}, \text{ex}, \text{prim})], \text{top} + 2) \\
&< \text{FD, env}, \sigma_2 > \downarrow 2 < \sigma_3, v >
\end{align*}\]

< function i ( il ) {body} \text{FD, env} > \downarrow 2 < \sigma_3, v >
FunDecl3

\[(map, top) = \sigma \quad (o, outer) = map(env)\]

\[o \in \text{Loc} \quad (m, pr, cl, ex, prim) = map(o) \quad (v, true, b, true) = m[Str(i)]\]

\[proto = \{(\"constructor\", (top + 1, true, false, true))\}, OPO, \"Object\", true, null\]

\[m_2 = \{(\"length\", (Len(il), false, false, false)), (\"prototype\", (top, true, false, false))\}\]

\[f = (m_2, FPO, \"Function\", true, (body, il, env, true))\]

\[m_3 = m[Str(i)/(loc(top + 1), true, true, false)]\]

\[\sigma_2 = (map[top/proto, top/(top + 1)/f, o/\sigma_3/m_3, pr, cl, ex, prim], top + 2)\]

\[<FD, env, \sigma_2 > \Downarrow_2 <\sigma_3, v_2>\]

FunDecl4

\[(map, top) = \sigma \quad (o, outer) = map(env)\]

\[o \in \text{Loc} \quad (m, pr, cl, ex, prim) = map(o) \quad (v, false, false, false) = m[Str(i)]\]

\[error = (\emptyset, TEPO, \"Error\", true, null)\]

\[\sigma_2 = (map[top/error, top/(top + 1), true, true, false])\]

\[<FD, env, \sigma_2 > \Downarrow_2 <\sigma_3, exc(top)>\]

A.2.4 Statements

StatementList1

\[<s_1, env, this, \sigma > \Downarrow_3 <\sigma_1, (type, value, target) > \quad type! = normal\]

\[<s_1 \quad s_2, env, this, \sigma > \Downarrow_3 <\sigma_1, (type, value, target) >\]

StatementList2

\[<s_1, env, this, \sigma > \Downarrow_3 <\sigma_1, (normal, value_1, target_1) >\]

\[<s_2, env, this, \sigma_1 > \Downarrow_3 <\sigma_2, (type_2, empty, target_2) >\]

\[<s_1 \quad s_2, env, this, \sigma > \Downarrow_3 <\sigma_1, (type_2, value_1, target_2) >\]

StatementList3

\[<s_1, env, this, \sigma > \Downarrow_3 <\sigma_1, (normal, value_1, target_1) >\]

\[<s_2, env, this, \sigma_1 > \Downarrow_3 <\sigma_2, (type_2, value_2, target_2) > \quad value_2 \in Value\]

\[<s_1 \quad s_2, env, this, \sigma > \Downarrow_3 <\sigma_2, (type_2, value_2, target_2) >\]

EmptyStatement

\[<empty, env, this, \sigma > \Downarrow_3 <\sigma_1, (normal, empty, empty) >\]

ExpressionStatementNoException

\[<e, env, this, \sigma, val(null) > \Downarrow_4 <\sigma_1, exprVal >\]

\[<\text{GetValue}, exprVal, \sigma_1 > \Downarrow_6 <\sigma_2, val(v) >\]

\[<e; , env, this, \sigma > \Downarrow_3 <\sigma_2, (normal, v, empty) >\]
ExpressionStatementException

\[ \langle e, env, this, \sigma, val(null) \rangle \Downarrow_4 \langle \sigma_1, exprVal \rangle \]
\[ \Downarrow \quad \langle \text{GetValue, exprVal, } \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, exc(v) \rangle \]
\[ \langle e ; , env, this, \sigma \rangle \Downarrow_3 \langle \sigma_2, (\text{throw, } v, \text{empty}) \rangle \]

IfStatement1

\[ \langle e, env, this, \sigma, val(null) \rangle \Downarrow_4 \langle \sigma_1, exprVal \rangle \]
\[ \Downarrow \quad \langle \text{GetValue, exprVal, } \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, exc(v) \rangle \]
\[ \langle \text{if( } e \text{ ) } s_1 \text{ else } s_2, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_2, (\text{throw, } v, \text{empty}) \rangle \]

IfStatement2

\[ \langle e, env, this, \sigma, val(null) \rangle \Downarrow_4 \langle \sigma_1, exprVal \rangle \]
\[ \Downarrow \quad \langle \text{GetValue, exprVal, } \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(v) \rangle \]
\[ \text{Bool}(v) \quad \langle s_1, env, this, \sigma_2 \rangle \Downarrow_3 \langle \sigma_3, \text{comp} \rangle \]
\[ \langle \text{if( } e \text{ ) } s_1 \text{ else } s_2, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_3, \text{comp} \rangle \]

IfStatement3

\[ \langle e, env, this, \sigma, val(null) \rangle \Downarrow_4 \langle \sigma_1, exprVal \rangle \]
\[ \Downarrow \quad \langle \text{GetValue, exprVal, } \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(v) \rangle \]
\[ \text{!Bool}(v) \quad \langle s_2, env, this, \sigma_2 \rangle \Downarrow_3 \langle \sigma_3, \text{comp} \rangle \]
\[ \langle \text{if( } e \text{ ) } s_1 \text{ else } s_2, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_3, \text{comp} \rangle \]

LabelledStatement1

\[ \langle s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{break, } v, \text{id}) \rangle \]
\[ \langle id : s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{normal, } v, \text{empty}) \rangle \]

LabelledStatement2

\[ \langle s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{continue, } v, \text{id}) \rangle \]
\[ \langle id : s, env, thisVal, \sigma_1 \rangle \Downarrow_3 \langle \sigma_2, (\text{type, empty, target}) \rangle \]
\[ \langle id : s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_2, (\text{type, } v, \text{target}) \rangle \]

LabelledStatement3

\[ \langle s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{continue, } v_1, \text{id}) \rangle \]
\[ \langle id : s, env, this, \sigma_1 \rangle \Downarrow_3 \langle \sigma_2, (\text{type, } v_2, \text{target}) \rangle \quad v_2 \in \text{Value} \]
\[ \langle \text{identifier} : s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_2, (\text{type, } v_2, \text{target}) \rangle \]

LabelledStatement4

\[ \langle s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{type, } v, \text{label}) \rangle \quad id \neq \text{label} \]
\[ \langle id : s, env, this, \sigma \rangle \Downarrow_3 \langle \sigma_1, (\text{type, } v, \text{label}) \rangle \]

While1
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow_6 < \sigma_2, \text{exc}(v) >\]
\[< \text{while}( e ) s, env, this, \sigma > \downarrow_3 < \sigma_2, (\text{throw}, v, \text{empty}) >\]

While2
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \text{!Bool}(v)\]
\[< \text{while}( e ) s, env, this, \sigma > \downarrow_3 < \sigma_2, (\text{normal, empty, empty}) >\]

While3
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \text{Bool}(v)\]
\[< s, env, this, \sigma_2 > \downarrow_3 < \sigma_3, (\text{type, } v_2, \text{label}) > \quad \text{type}! = \text{normal}\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_3 < \sigma_3, (\text{type, } v_2, \text{label}) >\]

While4
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \text{Bool}(v)\]
\[< s, env, this, \sigma_2 > \downarrow_3 < \sigma_3, (\text{normal, } v_2, \text{empty}) >\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_3 < \sigma_3, (\text{normal, empty, empty}) >\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_4 < \sigma_4, (\text{normal, } v_2, \text{empty}) >\]

While5
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \text{Bool}(v)\]
\[< s, env, this, \sigma_2 > \downarrow_3 < \sigma_3, (\text{normal, } v_2, \text{empty}) >\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_3 < \sigma_3, (\text{normal, } v_3, \text{empty}) > \quad v_3 \in \text{Value}\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_4 < \sigma_4, (\text{normal, } v_3, \text{empty}) >\]

While6
\[< e, env, this, \sigma, val(null) > \downarrow_4 < \sigma_1, exprVal >\]
\[< \text{GetValue}, exprVal, \sigma_1 > \downarrow < \sigma_2, \text{val}(v) > \text{Bool}(v)\]
\[< s, env, this, \sigma_2 > \downarrow_3 < \sigma_3, (\text{normal, } v_2, \text{empty}) >\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_3 < \sigma_3, (\text{type, } v_3, \text{target}) > \quad \text{type}! = \text{normal}\]
\[< \text{while}( e, s, env, this, \sigma ) > \downarrow_4 < \sigma_4, (\text{type, } V_3, \text{target}) >\]

ContinueStatement
\[< \text{continue } id ; , env, this, \sigma > \downarrow_3 < \sigma, (\text{continue, empty, } id) >\]
BreakStatement
< break id ; , env, this, σ > \downarrow_3 < σ, (break, empty, id) >

ReturnStatement1
< e, env, this, σ, val(null) > \downarrow_4 < σ_1, exprVal >
   < GetValue, exprVal, σ_1 > \downarrow_6 < σ_2, exc(v) >
   < return e ; , env, this, σ > \downarrow_3 < σ_2, (throw, v, empty) >

ReturnStatement2
< e, env, this, σ, val(null) > \downarrow_4 < σ_1, exprVal >
   < GetValue, exprVal, σ_1 > \downarrow_6 < σ_2, val(v) >
   < return e ; , env, this, σ > \downarrow_3 < σ_2, (return, v, empty) >

ThrowStatement1
< e, env, this, σ, val(null) > \downarrow_4 < σ_1, exprVal >
   < GetValue, exprVal, σ_1 > \downarrow_6 < σ_2, val(v) >
   < throw e ; , env, this, σ > \downarrow_3 < σ_2, (throw, v, empty) >

ThrowStatement2
< e, env, this, σ, val(null) > \downarrow_4 < σ_1, exprVal >
   < GetValue, exprVal, σ_1 > \downarrow_6 < σ_2, exc(v) >
   < throw e ; , env, this, σ > \downarrow_3 < σ_2, (throw, v, empty) >

TryStatement1
< s_1, env, this, σ > \downarrow_3 < σ_1, (t_1, v_1, tar_1) >
   < s_3, env, this, σ_1 > \downarrow_3 < σ_2, (normal, v_2, tar_2) >
   t_1 \neq \text{throw}
   < \text{try}\{s_1\} \ \text{catch}(id)\{s_2\} \ \text{finally}\{s_3\}, env, this, σ > \downarrow_3 < σ_2, (t_1, v_1, tar_1) >

TryStatement2
< s_1, env, this, σ > \downarrow_3 < σ_1, (t_1, v_1, tar_1) >
   < s_3, env, this, σ_1 > \downarrow_3 < σ_2, (t_2, v_2, tar_2) >
   t_1 \neq \text{throw} \quad t_2 \neq \text{normal}
   < \text{try}\{s_1\} \ \text{catch}(id)\{s_2\} \ \text{finally}\{s_3\}, env, this, σ > \downarrow_3 < σ_2, (t_2, v_2, tar_2) >

TryStatement3
< s_1, env, this, σ > \downarrow_3 < σ_1, (throw, v_1, empty) >
   (map, top) = σ_1 \quad env_2 = (\{(id, (v_1, false))\}, env)
   σ_2 = (map[top/env_2], top + 1)
   < s_2, top, this, σ_2 > \downarrow_3 < σ_3, \text{comp} >
   < s_3, env, this, σ_3 > \downarrow_3 < σ_4, (normal, v_2, tar_2) >
   < \text{try}\{s_1\} \ \text{catch}(id)\{s_2\} \ \text{finally}\{s_3\}, env, this, σ > \downarrow_3 < σ_4, \text{comp} >
TryStatement4
\[
< s_1, env, this, \sigma > \Downarrow < \sigma_3, (\text{throw}, v_1, \text{empty}) > \\
(map, top) = \sigma_1 \\
env_2 = \{ (id, (v_1, \text{false})) \}, env \\
\sigma_2 = (map[\text{top/env}_2], \text{top} + 1) \\
< s_2, \text{top}, this, \sigma_2 > \Downarrow < \sigma_3, \text{comp} > \\
< s_3, env, this, \sigma_3 > \Downarrow < \sigma_4, (t_2, v_2, \text{tar}_2) > \\
t_2 \neq \text{normal} \\
< \text{try}\{ s_1 \} \text{catch}(id)\{ s_2 \} \text{finally}\{ s_3 \}, env, this, \sigma > \Downarrow < \sigma_4, (t_2, v_2, \text{tar}_2) >
\]

A.2.5 Expressions

ThisExprException
\[
< \text{this}, env, this, \sigma, \text{exc}(v) > \Downarrow < \sigma, \text{exc}(v) >
\]

ThisExprNoException
\[
\lnot \text{IsException}(\text{exprVal}) \\
< \text{this}, env, this, \sigma, \text{exprVal} > \Downarrow < \sigma, \text{val}(\text{this}) >
\]

\text{IsException}(x) = (x = \text{exc}(y))

IdentifierExpr1
\[
< \text{id}, env, this, \sigma, \text{exc}(v) > \Downarrow < \sigma, \text{exc}(v) >
\]

IdentifierExpr2
\[
\lnot \text{IsException}(\text{exprVal}) \\
\text{id} \in I \\
(map, top) = \sigma \\
(o, \text{outer}) = map(env) \\
o \in \text{Loc} \\
\text{HasProperty}(\text{Str(id)}, o, \sigma) \\
< \text{id}, env, this, \sigma, \text{exprVal} > \Downarrow < \sigma, \text{ref}(\text{loc(env)}, \text{Str(id)}) >
\]

IdentifierExpr3
\[
\lnot \text{IsException}(\text{exprVal}) \\
\text{id} \in I \\
(map, top) = \sigma \\
(o, \bot) = map(env) \\
o \in \text{Loc} \\
\lnot \text{HasProperty}(\text{Str(id)}, o, \sigma) \\
< \text{id}, env, this, \sigma, \text{exprVal} > \Downarrow < \sigma, \text{ref}(\text{undefined}, \text{Str(id)}) >
\]
IdentifierExpr4

\[ IsException(exprVal) \quad id \in I \quad (map, top) = \sigma \]
\[ (d, outer) = map(env) \quad d \in \text{DeclarativeEnvironmentRecord} \quad id \in \text{domain}(d) \]
\[ <id, env, this, \sigma, exprVal > \downarrow 4 < \sigma, \text{ref}(\text{loc}(env), \text{Str}(id)) > \]

IdentifierExpr5

\[ IsException(exprVal) \quad id \in I \quad (map, top) = \sigma \quad (d, outer) = map(env) \quad d \in \text{DeclarativeEnvironmentRecord} \quad id \notin \text{domain}(d) \]
\[ <id, outer, this, \sigma, val(v) > \downarrow 4 < \sigma, exprVal1 > \]
\[ <id, env, this, \sigma, exprVal > \downarrow 4 < \sigma, exprVal1 > \]

LiteralExprNoException

\[ IsException(exprVal) \quad literal \in L \]
\[ <literal, env, this, \sigma, exprVal > \downarrow 4 < \sigma, \text{val}(literal) > \]

MemberExpr1

\[ <e_1, env, this, \sigma, exprVal > \downarrow 4 < \sigma_1, exprVal1 > \]
\[ <\text{GetValue}, exprVal1, \sigma_1 > \downarrow 6 < \sigma_2, \text{val}(v_1) > \]
\[ <e_2, env, this, \sigma_2, \text{val}(v_1) > \downarrow 4 < \sigma_3, exprVal2 > \]
\[ <\text{GetValue}, exprVal2, \sigma_3 > \downarrow 6 < \sigma_4, \text{val}(v_2) > \]
\[ v_1 \in \{\text{null, undefined}\} \quad \text{error} = (\emptyset, \text{TEPO}, \text{"Error"}, \text{true, null}) \quad (map, top) = \sigma_4 \]
\[ <e_1 \left[ e_2 \right], env, this, \sigma, exprVal > \downarrow 4 < \text{map}[\text{top/error}], \text{top} + 1), \text{exc}(\text{top}) > \]

MemberExpr2

\[ <e_1, env, this, \sigma, exprVal > \downarrow 4 < \sigma_1, exprVal1 > \]
\[ <\text{GetValue}, exprVal1, \sigma_1 > \downarrow 6 < \sigma_2, \text{val}(v) > \]
\[ <e_2, env, this, \sigma_2, \text{val}(v) > \downarrow 4 < \sigma_3, exprVal3 > \]
\[ <\text{GetValue}, exprVal3, \sigma_3 > \downarrow 6 < \sigma_4, \text{val}(v_2) > \]
\[ <\text{ToString}, \text{val}(v_2), \sigma_4 > \downarrow 6 < \sigma_5, \text{val}(s) > \quad v \notin \{\text{null, undefined}\} \]
\[ <e_1 \left[ e_2 \right], env, this, \sigma, exprVal > \downarrow 4 < \sigma_5, \text{ref}(v, s) > \]
MEMBEREXPR3
\[ < e_1, \text{env}, this, \sigma, exprVal_1 > \downarrow_4 < \sigma_1, exprVal_1 > \]
\[ < \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, exprVal_2 > \]
\[ < e_2, \text{env}, this, \sigma_2, exprVal_2 > \downarrow_4 < \sigma_3, exprVal_3 > \]
\[ < \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, exprVal_4 > \]
\[ < \text{ToString}, exprVal_4, \sigma_4 > \downarrow_6 < \sigma_5, \text{exc}(v) > \]
\[ < e_1[\ e_2\ ], \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_5, \text{exc}(v) > \]

ASSIGNEXPR1
\[ < e_1, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_1, \text{ref(undefined, name)} > \]
\[ < e_2, \text{env}, this, \sigma_1, \text{val(null)} > \downarrow_4 < \sigma_2, exprVal_2 > \]
\[ < \text{GetValue}, exprVal_2, \sigma_2 > \downarrow_6 < \sigma_3, \text{val}(v) > \]
\[ \text{error} = \{\emptyset, \text{REPO}, \text{"Error"}, \text{true}, \text{null}\} \]
\[ (map, top) = \sigma_3 \quad \sigma_4 = (map[\text{top/error}], top + 1) \]
\[ < e_1 = e_2, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_4, \text{exc}(\text{top}) > \]

ASSIGNEXPR2
\[ < e_1, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_1, \text{ref(base, name)} > \]
\[ < e_2, \text{env}, this, \sigma_1, \text{val(null)} > \downarrow_4 < \sigma_2, exprVal_2 > \]
\[ < \text{GetValue}, exprVal_2, \sigma_2 > \downarrow_6 < \sigma_3, \text{val}(v) > \]
\[ \text{base} \in \text{Boolean} \cup \text{Number} \cup \text{String} \quad \text{error} = \{\emptyset, \text{TEPO}, \text{"Error"}, \text{true}, \text{null}\} \]
\[ (map, top) = \sigma_3 \quad \sigma_4 = (map[\text{top/error}], top + 1) \]
\[ < e_1 = e_2, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_4, \text{exc}(\text{top}) > \]

ASSIGNEXPR3
\[ < e_1, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_1, \text{ref(loc(l), name)} > \]
\[ < e_2, \text{env}, this, \sigma_1, \text{val(null)} > \downarrow_4 < \sigma_2, exprVal_2 > \]
\[ < \text{GetValue}, exprVal_2, \sigma_2 > \downarrow_6 < \sigma_3, \text{val}(l) > \]
\[ \text{(map, top)} = \sigma_3 \quad (d, \text{outer}) = \text{map}(l) \]
\[ d \in \text{DeclarativeEnvironmentRecord} \quad (v, \text{true}) = d(\text{Id(name)}) \]
\[ \text{error} = \{\emptyset, \text{TEPO}, \text{"Error"}, \text{true}, \text{null}\} \quad \sigma_4 = (map[\text{top/error}], top + 1) \]
\[ < e_1 = e_2, \text{env}, this, \sigma, exprVal > \downarrow_4 < \sigma_4, \text{exc}(\text{top}) > \]
ASSIGNExpr4

\[
\begin{aligned}
&< e_1, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_1, ref(loc(l), name) > \\
&< e_2, env, this, \sigma_1, val(null) >^\downarrow_4 < \sigma_2, exprVal_2 > \\
&< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\end{aligned}
\]

\[(map, top) = \sigma_3 \quad (a, outer) = map(l) \quad d \in DeclarativeEnvironmentRecord \\
(v_2, true) \neq d(Id(name)) \quad \sigma_4 = (map[l/(d[Id(name)]/(v, false))], top) \]

\[
< e_1 = e_2, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_4, val(v) >
\]

ASSIGNExpr5

\[
\begin{aligned}
&< e_1, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_1, ref(loc(l), name) > \\
&< e_2, env, this, \sigma_1, val(null) >^\downarrow_4 < \sigma_2, exprVal_2 > \\
&< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\end{aligned}
\]

\[
< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\quad (map, top) = \sigma_3 \quad (o, ⊥) = map(l) \\
\quad o ∈ Loc \quad CanPut(name, o, \sigma_3) \quad (m, pr, cl, ex, prim) = map(o) \\
\quad name ∈ domain(m) \quad (v_dd, wr, en, conf) = m(name) \\
\quad m_2 = m[name/(v, wr, en, conf)] \quad \sigma_4 = (map[o/(m_2, pr, cl, ex, prim)], top) \\
\]

\[
< e_1 = e_2, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_4, val(v) >
\]

ASSIGNExpr6

\[
\begin{aligned}
&< e_1, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_1, ref(loc(l), name) > \\
&< e_2, env, this, \sigma_1, val(null) >^\downarrow_4 < \sigma_2, exprVal_2 > \\
&< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\end{aligned}
\]

\[
< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\quad (map, top) = \sigma_3 \quad (o, ⊥) = map(l) \\
\quad o ∈ Loc \quad CanPut(name, o, \sigma_3) \quad (m, pr, cl, ex, prim) = map(o) \\
\quad m_2 = m[name/(v, true, true, true)] \quad \sigma_4 = (map[o/(m_2, pr, cl, ex, prim)], top) \\
\]

\[
< e_1 = e_2, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_4, val(v) >
\]

ASSIGNExpr7

\[
\begin{aligned}
&< e_1, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_1, ref(loc(l), name) > \\
&< e_2, env, this, \sigma_1, val(null) >^\downarrow_4 < \sigma_2, exprVal_2 > \\
&< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\end{aligned}
\]

\[
< GetValue, exprVal_2, \sigma_2 >^\downarrow_6 < \sigma_3, val(v) > \\
\quad (map, top) = \sigma_3 \quad (o, ⊥) = map(l) \\
\quad o ∈ Loc \quad CanPut(name, o, \sigma_3) \\
\quad error = \{∅, TEPO, 'Error', true, null\} \quad \sigma_4 = (map[top/error], top + 1) \\
\]

\[
< e_1 = e_2, env, this, \sigma, exprVal >^\downarrow_4 < \sigma_4, exc(top) >
\]
Formal Semantics of JavaScript

\[ \text{CanPut}(s, o, (\text{map}, \text{top})) = \begin{cases} wr & \text{if } m[s] = (v, wr, en, conf) \\ ex \land \text{Wr}(s, o, (\text{map}, \text{top})) & \text{if } s \notin \text{domain}(m) \land pr \in \text{Loc} \\ ex & \text{if } s \notin \text{domain}(m) \land pr \notin \text{Loc} \end{cases} \]

\[ \text{Wr}(s, o, (\text{map}, \text{top})) = \begin{cases} wr & \text{if } m[s] = (v, wr, en, conf) \\ \text{Wr}(s, pr, (\text{map}, \text{top})) & \text{if } s \notin \text{domain}(m) \land pr \in \text{Loc} \\ \text{true} & \text{if } s \notin \text{domain}(m) \land pr \notin \text{Loc} \end{cases} \]

with

\[ \text{map}[o] = (m, pr, cl, ex, prim) \]

**ASSIGNEXPR8**

\[ \langle e_1, \text{env}, \text{this}, \sigma, \text{exprVal} \rangle \downarrow_4 \sigma_1, \text{ref(loc(l), name)} \rangle \]

\[ \langle e_2, \text{env}, \text{this}, \sigma_1, \text{val(null)} \rangle \downarrow_4 \sigma_2, \text{exprVal}_2 \rangle \]

\[ \langle \text{GetValue}, \text{exprVal}_2, \sigma_2 \rangle \downarrow_6 \sigma_3, \text{val}(v) \rangle \]

\[ (\text{map}, \text{top}) = \sigma_3 \quad (m, pr, cl, ex, prim) = \text{map}(l) \]

\[ \text{CanPut}(\text{name}, l, \sigma_3) \quad \text{name} \in \text{domain}(m) \quad (v, \text{old}, wr, en, conf) = m(\text{name}) \]

\[ m_2 = m[\text{name}/(v, wr, en, conf)] \quad \sigma_4 = (\text{map}[l/(m_2, pr, cl, ex, prim)], \text{top}) \]

\[ \langle e_1 = e_2, \text{env}, \text{this}, \sigma, \text{exprVal} \rangle \downarrow_4 \sigma_4, \text{val}(v) \rangle \]

**ASSIGNEXPR9**

\[ \langle e_1, \text{env}, \text{this}, \sigma, \text{exprVal} \rangle \downarrow_4 \sigma_1, \text{ref(loc(l), name)} \rangle \]

\[ \langle e_2, \text{env}, \text{this}, \sigma_1, \text{val(null)} \rangle \downarrow_4 \sigma_2, \text{exprVal}_2 \rangle \]

\[ \langle \text{GetValue}, \text{exprVal}_2, \sigma_2 \rangle \downarrow_6 \sigma_3, \text{val}(v) \rangle \quad (\text{map}, \text{top}) = \sigma_3 \]

\[ (m, pr, cl, ex, prim) = \text{map}(l) \quad \text{CanPut}(\text{name}, l, \sigma_3) \quad \text{name} \notin \text{domain}(m) \]

\[ m_2 = m[\text{name}/(v, true, true, true)] \quad \sigma_4 = (\text{map}[l/(m_2, pr, cl, ex, prim)], \text{top}) \]

\[ \langle e_1 = e_2, \text{env}, \text{this}, \sigma, \text{exprVal} \rangle \downarrow_4 \sigma_4, \text{val}(v) \rangle \]
AssignExpr10
\[
\langle e_1, env, this, \sigma, exprVal > \downarrow 4 < \sigma_1, ref(loc(l), name) >
\]
\[
\langle e_2, env, this, \sigma_1, val(null) > \downarrow 4 < \sigma_2, exprVal_2 >
\]
\[
\langle \text{GetVal}, exprVal_2, \sigma_2 > \downarrow 6 < \sigma_3, val(v) >
\]
\[
(map, top) = \sigma_3 \quad !\text{CanPut}(name, l, \sigma_3)
\]
error = \{\emptyset, TEPO, 'Error', true, null\} \quad \sigma_4 = (map[\text{top}/error], top + 1)
\[
\langle e_1 = e_2, env, this, \sigma, exprVal > \downarrow 4 < \sigma_4, exc(top) >
\]

AssignExpr11
\[
\langle e_1, env, this, \sigma, exprVal > \downarrow 4 < \sigma_1, exprVal_1 >
\]
\[
\langle e_2, env, this, \sigma_1, exprVal_1 > \downarrow 4 < \sigma_2, exprVal_2 >
\]
\[
\langle \text{GetVal}, exprVal_2, \sigma_2 > \downarrow 6 < \sigma_3, exc(e) >
\]
\[
\langle e_1 = e_2, env, this, \sigma, exprVal > \downarrow 4 < \sigma_4, exc(e) >
\]

UnaryOpExpr1
\[
\langle e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 >
\]
\[
\langle \text{GetVal}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, eVal_2 >
\]
\[
\langle \text{ToNumber}, eVal_2, \sigma_2 > \downarrow 6 < \sigma_3, val(n) >
\]
\[
\langle + e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_2, val(+n) >
\]

UnaryOpExpr2
\[
\langle e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 >
\]
\[
\langle \text{GetVal}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, eVal_2 >
\]
\[
\langle \text{ToNumber}, eVal_2, \sigma_2 > \downarrow 6 < \sigma_3, val(n) >
\]
\[
\langle - e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_2, val(-n) >
\]

UnaryOpExpr3
\[
\langle e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 >
\]
\[
\langle \text{GetVal}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, eVal_2 >
\]
\[
\langle \text{ToNumber}, eVal_2, \sigma_2 > \downarrow 6 < \sigma_3, val(n) >
\]
\[
\langle - e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_2, val(-n) >
\]

UnaryOpExpr4
\[
\langle e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 >
\]
\[
\langle \text{GetVal}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, eVal_2 >
\]
\[
\langle \text{ToNumber}, eVal_2, \sigma_2 > \downarrow 6 < \sigma_3, val(n) >
\]
\[
\langle ! e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_2, exc(e) >
\]
UNARYOPExpr6
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle GetValue, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(v) \rangle \\
\langle ! e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_2, val(Bool(v)) \rangle
\]

UNARYOPExpr7
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle ! e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_2, exc(e) \rangle
\]

UNARYOPExpr8
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle GetValue, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(v) \rangle
\]

UNARYOPExpr9
\[
eVal_1 \neq \text{ref(undefined, name)} \quad \langle GetValue, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(v) \rangle \\
\langle typeof e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_2, val(\text{Type}(v)) \rangle
\]

DELETEExpr1
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, exc(e) \rangle \\
\langle ! e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, exc(e) \rangle
\]

DELETEExpr2
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, val(v) \rangle \\
\langle ! e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, val(true) \rangle
\]

DELETEExpr3
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, ref(base, name) \rangle \\
\langle ToObject, base, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(loc(l)) \rangle \\
\sigma_2 = \langle map, top \rangle \\
map(l) = \langle m, pr, cl, ex, prim \rangle \\
names \not\in \text{domain}(m)
\]

DELETEExpr4
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, ref(base, name) \rangle \\
\langle ToObject, base, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(loc(l)) \rangle \\
\sigma_2 = \langle map, top \rangle \\
map(l) = \langle m, pr, cl, ex, prim \rangle \\
m(name) = (v, wr, en, true) \\
m_2 = \{ (s, p) | (s, p) \in m \land s \neq \text{name} \} \\
\sigma_3 = \langle map[l/(m_2, pr, cl, ex, prim)], top \rangle
\]

DELETEExpr5
\[
\langle e, env, thisVal, \sigma, eVal \rangle \Downarrow_4 \langle \sigma_1, ref(base, name) \rangle \\
\langle ToObject, base, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, val(loc(l)) \rangle \\
\sigma_2 = \langle map, top \rangle \\
map(l) = \langle m, pr, cl, ex, prim \rangle \\
m(name) = (v, wr, en, false) \\
error = (\emptyset, TEPO, 'Error', true, null) \\
(map[top/error], top + 1) = \sigma_3
\]

< delete e, env, thisVal, \sigma, eVal > \Downarrow_4 \langle \sigma_3, exc(top) \rangle
CALLExpr1
\[
\langle e, env, this, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle \text{GetValue}, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, \text{exc}(e) \rangle \\
\langle e(\text{al}), env, this, eVal \rangle \Downarrow_4 \langle \sigma_2, \text{exc}(e) \rangle
\]

CALLExpr2
\[
\langle e, env, thisVal, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle \text{GetValue}, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\text{IsCallable}(v, \sigma_2) = \{(^*\text{length}^*, \text{Len(al)}, \text{false}, \text{false}, \text{false})\}
\]
\[
\text{args} = (m, \text{OPO}, ^*\text{Arguments}^*, \text{true}, \text{null}) \\
\text{env}_2 = \{(\text{arguments}, (\text{top}, \text{true}))\}, \text{GE}
\]
\[
\sigma_2 = (\text{map}, \text{top}) \\
\text{map}_2 = \text{map}[\text{top/args}, (\text{top} + 1)/\text{env}_2]
\]
\[
\text{error} = (\emptyset, \text{TEPO}, ^*\text{Error}^*, \text{true}, \text{null}) \\
\sigma_4 = (\text{map}_3[\text{top/error}], \text{top}_3 + 1)
\]
\[
\langle e(\text{al}), env, thisVal, eVal \rangle \Downarrow_4 \langle \sigma_4, \text{exc}(\text{top}_3 + 1) \rangle
\]

\text{IsCallable}(v, (\text{map}, \text{top})) = f \in \text{Function},

where f defined by
\[
\text{map}(v) = (m, pr, d, ex, f)
\]

CALLExpr3
\[
\langle e, env, thisVal, eVal \rangle \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
\langle \text{GetValue}, eVal_1, \sigma_1 \rangle \Downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\text{IsCallable}(v, \sigma_2) = \{(^*\text{length}^*, \text{Len(al)}, \text{false}, \text{false}, \text{false})\}
\]
\[
\text{args} = (m, \text{OPO}, ^*\text{Arguments}^*, \text{true}, \text{null}) \\
\text{env}_2 = \{(\text{arguments}, (\text{top}, \text{true}))\}, \text{GE}
\]
\[
\sigma_2 = (\text{map}, \text{top}) \\
\text{map}_2 = \text{map}[\text{top/args}, (\text{top} + 1)/\text{env}_2]
\]
\[
\langle e(\text{al}), env, thisVal, eVal \rangle \Downarrow_4 \langle \sigma_3, \text{exc}(e) \rangle
\]
Formal Semantics of JavaScript

\[ \text{CallExpr4} \]

\[ e, env, thisVal, \sigma, eVal \triangledown 4 < \sigma_1, eVal_1 > \]
\[ \text{GetValue, eVal}_1, \sigma_1 \triangledown 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]
\[ m_a = \{("length", (Len(al), false, false, false))\} \]
\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]
\[ env_2 = \{(\text{arguments}, (top, true)), env_0\} \quad \text{map}_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2] \]
\[ < \text{il}, al, env, top + 1, 0, (\text{map}_2, top + 2) \triangledown 5 < \sigma_3, \text{exc}(e) > \]
\[ e (\text{al}) , env, thisVal, \sigma, eVal \triangledown 4 < \sigma_3, \text{exc}(e) > \]

\[ \text{CallExpr5} \]

\[ e, env, thisVal, \sigma, eVal \triangledown 4 < \sigma_1, eVal_1 > \]
\[ \text{GetValue, eVal}_1, \sigma_1 \triangledown 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]
\[ m_a = \{("length", (Len(al), false, false, false))\} \]
\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]
\[ env_2 = \{(\text{arguments}, (top, true)), env_0\} \quad \text{map}_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2] \]
\[ < \text{il}, al, env, top + 1, 0, (\text{map}_2, top + 2) \triangledown 5 < \sigma_3, \text{val}(v) > \]
\[ < \text{body, top} + 1, ThisVal(eVal_1), \sigma_3 \triangledown 3 < \sigma_4, (\text{return}, v_4, \text{empty}) > \]
\[ e (\text{al}) , env, thisVal, \sigma, eVal \triangledown 4 < \sigma_4, \text{val}(v_4) > \]

\[ \text{CallExpr6} \]

\[ e, env, thisVal, \sigma, eVal \triangledown 4 < \sigma_1, eVal_1 > \]
\[ \text{GetValue, eVal}_1, \sigma_1 \triangledown 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]
\[ m_a = \{("length", (Len(al), false, false, false))\} \]
\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]
\[ env_2 = \{(\text{arguments}, (top, true)), env_0\} \quad \text{map}_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2] \]
\[ < \text{il}, al, env, top + 1, 0, (\text{map}_2, top + 2) \triangledown 5 < \sigma_3, \text{val}(v) > \]
\[ < \text{body, top} + 1, ThisVal(eVal_1), \sigma_3 \triangledown 3 < \sigma_4, (\text{throw}, v_4, \text{empty}) > \]
\[ e (\text{al}) , env, thisVal, \sigma, eVal \triangledown 4 < \sigma_4, \text{exc}(v_4) > \]
CALLExpr7
\[
< e, env, thisVal, σ, eVal > \downarrow_4 < σ_1, eVal_1 > \\
< \text{GetValue}, eVal_1, σ_1 > \downarrow_6 < σ_2, \text{val}(\text{loc}(l)) > \\
σ_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \\
m_a = \{("\text{length}"), (\text{Len}(al), false, false, false))\}
\]
args = (m_a, OPO, "Arguments", true, null)

env_2 = (\{(arguments, (top, true)), env_0\}) \quad \text{map}_2 = \text{map}[\text{top}/\text{args}, (\text{top} + 1)/env_2]
\[
< il, al, env, top + 1, 0, (\text{map}_2, \text{top} + 2) > \downarrow_5 < σ_3, \text{val}(v) > \\
< \text{body}, top + 1, \text{ThisVal}(eVal_1), σ_3 > \downarrow_3 < σ_4, (\text{normal}, v_4, \text{empty}) > \\
< e (al), env, thisVal, σ, eVal > \downarrow_4 < σ_4, \text{val}(\text{undefined}) >
\]

NEWExpr1
\[
< e, env, this, σ, eVal > \downarrow_4 < σ_1, eVal_1 > \\
< \text{GetValue}, eVal_1, σ_1 > \downarrow_6 < σ_2, \text{exc}(e) > \\
< e (al), env, this, σ, eVal > \downarrow_4 < σ_2, \text{exc}(e) >
\]

NEWExpr2
\[
< e, env, thisVal, σ, eVal > \downarrow_4 < σ_1, eVal_1 > \\
< \text{GetValue}, eVal_1, σ_1 > \downarrow_6 < σ_2, \text{val}(v) > \\
\text{IsConstructor}(v, σ_2) \quad m = \{("\text{length}"), (\text{Len}(al), false, false, false))\}
\]
args = (m, OPO, "Arguments", true, null) \quad \text{env}_2 = (\{(arguments, (top, true)), GE\})
\[
σ_2 = (\text{map}, \text{top}) \quad \text{map}_2 = \text{map}[\text{top}/\text{args}, (\text{top} + 1)/\text{env}_2]
\]
\[
< \text{empty}, al, env, top + 1, 0, (\text{map}_2, \text{top} + 2) > \downarrow_5 < σ_3, \text{val}(v_2) > \quad σ_3 = (\text{map}_3, \text{top}_3)
\]
error = (∅, TEPO, "Error", true, null) \quad σ_4 = (\text{map}_3[\text{top}/\text{error}], \text{top}_3 + 1)
\[
< e (al), env, thisVal, σ, eVal > \downarrow_3 < σ_4, \text{exc}(\text{top}_3 + 1) >
\]

\text{IsConstructor}(f, (\text{map}, \text{top})) = (\text{map}[f] = (m, pr, cl, ex, (true, b, il, l)))
NewExpr3
\[
\begin{align*}
\langle e, \text{env}, thisVal, \sigma, eVal \rangle & \Downarrow_4 \langle \sigma_1, eVal_1 \rangle \\
& \Downarrow \text{GetValue, } eVal_1, \sigma_1 \Downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
& \Downarrow IsConstructor(v, \sigma_2) \\
& \quad m = \{("length", (\text{Len}(al), \text{false}, \text{false}, \text{false}))\} \\
& \quad \text{args} = (m, OPO, "Arguments", \text{true}, \text{null}) \\
& \quad env_2 = \{(\text{arguments, (top, true)}\}, GE) \\
& \quad \sigma_2 = (map, top) \\
& \quad \text{map}_2 = \text{map}\{\text{top/args}, (\text{top + 1})/\text{env}_2\} \\
& \quad < \emptyset, al, env, top + 1, 0, (\text{map}_2, \text{top + 2}) \Downarrow_5 < \sigma_3, \text{exc}(e) > \\
& \Downarrow < e \ (\text{al} \ ), \text{env}, thisVal, \sigma, eVal \Downarrow_4 < \sigma_3, \text{exc}(e) > 
\end{align*}
\]

NewExpr4
\[
\begin{align*}
\langle e, \text{env}, thisVal, \sigma, eVal \rangle & \Downarrow_4 < \sigma_1, eVal_1 > \\
& \Downarrow \text{GetValue, } eVal_1, \sigma_1 \Downarrow_6 < \sigma_2, \text{val}(l) > \\
& \quad \sigma_2 = (map, top) \\
& \quad \text{map}(l) = (m, pr, cl, ext, (\text{true}, b, il, env_0)) \\
& \quad m_a = \{("length", (\text{Len}(al), \text{false}, \text{false}, \text{false}))\} \\
& \quad \text{args} = (m_a, OPO, "Arguments", \text{true}, \text{null}) \\
& \quad env_2 = \{(\text{arguments, (top, true)}\}, env_0) \\
& \quad \text{map}_2 = \text{map}\{\text{top/args}, (\text{top + 1})/\text{env}_2\} \\
& \quad < il, al, env, top + 1, 0, (\text{map}_2, \text{top + 2}) \Downarrow_5 < \sigma_3, \text{exc}(e) > \\
& \Downarrow < e \ (\text{al} \ ), \text{env}, thisVal, \sigma, eVal \Downarrow_4 < \sigma_3, \text{exc}(e) > 
\end{align*}
\]

NewExpr5
\[
\begin{align*}
\langle e, \text{env}, thisVal, \sigma, eVal \rangle & \Downarrow_4 < \sigma_1, eVal_1 > \\
& \Downarrow \text{GetValue, } eVal_1, \sigma_1 \Downarrow_6 < \sigma_2, \text{val}(l) > \\
& \quad \sigma_2 = (map, top) \\
& \quad \text{map}(l) = (m, pr, cl, ext, (\text{true}, b, il, env_0)) \\
& \quad m_a = \{("length", (\text{Len}(al), \text{false}, \text{false}, \text{false}))\} \\
& \quad \text{args} = (m_a, OPO, "Arguments", \text{true}, \text{null}) \\
& \quad env_2 = \{(\text{arguments, (top, true)}\}, env_0) \\
& \quad m("prototype") = (\text{loc}(l_2), wr, en, co) \\
& \quad \text{this}_{obj} = (\emptyset, l_2, "Object", \text{true}, \text{null}) \\
& \quad \text{map}_2 = \text{map}\{\text{top/args}, (\text{top + 1})/\text{env}_2, (\text{top + 2})/\text{this}_{obj}\} \\
& \quad < il, al, env, top + 1, 0, (\text{map}_2, \text{top + 2}) \Downarrow_5 < \sigma_3, \text{val}(v) > \\
& \Downarrow < \text{body}, top + 1, (\text{top + 2}), \sigma_3 \Downarrow_3 < \sigma_4, (\text{normal}, v_4, \text{empty}) > \\
& \Downarrow < e \ (\text{al} \ ), \text{env}, thisVal, \sigma, eVal \Downarrow_4 < \sigma_4, \text{val}(\text{loc}(\text{top + 2})) > 
\end{align*}
\]
**NewExpr6**

\[ < e, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, eVal_1 > \]

\[ \text{GetValue}, eVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(\text{loc}(l)) > \]

\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]

\[ m_a = \{("length", (Len(al), false, false, false))\} \]

\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]

\[ env_2 = (\{(\text{arguments}, (\text{top}, true)), env_0\}) \quad m("prototype") = (v, wr, en, co) \]

\[ v \neq \text{loc}(l_2) \quad \text{this}_\text{obj} = (\emptyset, OPO, "Object", true, null) \]

\[ \text{map}_2 = \text{map}([\text{top}/\text{args}, (\text{top} + 1)/\text{env}_2, (\text{top} + 2)/\text{this}_\text{obj}] \]

\[ < \text{il}, \text{al}, \text{env}, \text{top} + 1, 0, (\text{map}_2, \text{top} + 3) > \downarrow_5 < \sigma_3, \text{val}(v) > \]

\[ < \text{body}, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow_3 < \sigma_4, (\text{normal}, v_4, \text{empty}) > \]

\[ < e (\text{al} ), \text{env}, \text{thisVal}, \sigma, eVal > \downarrow_4 < \sigma_1, \text{val}(\text{loc}(\text{top} + 2)) > \]

**NewExpr7**

\[ < e, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, eVal_1 > \]

\[ \text{GetValue}, eVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(\text{loc}(l)) > \]

\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]

\[ m_a = \{("length", (Len(al), false, false, false))\} \]

\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]

\[ env_2 = (\{(\text{arguments}, (\text{top}, true)), env_0\}) \quad \text{this}_\text{obj} = (\emptyset, l_2, "Object", true, null) \]

\[ m("prototype") = (\text{loc}(l_2), wr, en, co) \]

\[ \text{map}_2 = \text{map}([\text{top}/\text{args}, (\text{top} + 1)/\text{env}_2, (\text{top} + 2)/\text{this}_\text{obj}] \]

\[ < \text{il}, \text{al}, \text{env}, \text{top} + 1, 0, (\text{map}_2, \text{top} + 3) > \downarrow_5 < \sigma_3, \text{val}(v) > \]

\[ < \text{body}, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow_3 < \sigma_4, (\text{throw}, v_4, \text{empty}) > \]

\[ < e (\text{al} ), \text{env}, \text{thisVal}, \sigma, eVal > \downarrow_4 < \sigma_1, \text{exc}(v_4) > \]

**NewExpr8**

\[ < e, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, eVal_1 > \]

\[ \text{GetValue}, eVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(\text{loc}(l)) > \]

\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, pr, cl, ext, (c, b, il, env_0)) \]

\[ m_a = \{("length", (Len(al), false, false, false))\} \]

\[ \text{args} = (m_a, OPO, "Arguments", true, null) \]

\[ env_2 = (\{(\text{arguments}, (\text{top}, true)), env_0\}) \quad m("prototype") = (v, wr, en, co) \]

\[ v \in \text{PrimitiveValue} \quad \text{this}_\text{obj} = (\emptyset, OPO, "Object", true, null) \]

\[ \text{map}_2 = \text{map}([\text{top}/\text{args}, (\text{top} + 1)/\text{env}_2, (\text{top} + 2)/\text{this}_\text{obj}] \]

\[ < \text{il}, \text{al}, \text{env}, \text{top} + 1, 0, (\text{map}_2, \text{top} + 3) > \downarrow_5 < \sigma_3, \text{val}(v) > \]

\[ < \text{body}, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow_3 < \sigma_4, (\text{throw}, v_4, \text{empty}) > \]

\[ < e (\text{al} ), \text{env}, \text{thisVal}, \sigma, eVal > \downarrow_4 < \sigma_1, \text{exc}(v_4) > \]
Formal Semantics of JavaScript

NewExpr9
\[ < e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 > \]
\[ < \text{GetValue}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, \text{pr}, cl, \text{ext}, (c, b, il, env_0)) \]
\[ m_a = \{ \text{"length"}, (\text{Len}(al), \text{false}, \text{false}, \text{false}) \} \]
\[ \text{args} = (m_a, \text{OPO}, \text{"Arguments"}, \text{true}, \text{null}) \]
\[ env_2 = (\{ \text{(arguments, (top, true))}, env_0 \}) \]
\[ m(\text{"prototype"}) = (\text{loc}(l_2), wr, en, co) \quad this_{obj} = (\emptyset, l_2, \text{"Object"}, \text{true}, \text{null}) \]
\[ map_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2, (\text{top} + 2)/this_{obj}] \]
\[ < il, al, env, \text{top} + 1, 0, (map_2, \text{top} + 3) > \downarrow 5 < \sigma_3, \text{val}(v) > \]
\[ < body, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow 3 < \sigma_4, (\text{return}, \text{loc}(l_4), \text{empty}) > \]
\[ < e ( al ), env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_4, \text{val(loc}(l_4)) > \]

NewExpr10
\[ < e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 > \]
\[ < \text{GetValue}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, \text{pr}, cl, \text{ext}, (c, b, il, env_0)) \]
\[ m_a = \{ \text{"length"}, (\text{Len}(al), \text{false}, \text{false}, \text{false}) \} \]
\[ \text{args} = (m_a, \text{OPO}, \text{"Arguments"}, \text{true}, \text{null}) \]
\[ env_2 = (\{ \text{(arguments, (top, true))}, env_0 \}) \quad m(\text{"prototype"}) = (v, wr, en, co) \]
\[ v \in \text{PrimitiveValue} \quad this_{obj} = (\emptyset, \text{OPO}, \text{"Object"}, \text{true}, \text{null}) \]
\[ map_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2, (\text{top} + 2)/this_{obj}] \]
\[ < il, al, env, \text{top} + 1, 0, (map_2, \text{top} + 3) > \downarrow 5 < \sigma_3, \text{val}(v) > \]
\[ < body, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow 3 < \sigma_4, (\text{return}, \text{loc}(l_4), \text{empty}) > \]
\[ < e ( al ), env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_4, \text{val(loc}(l_4)) > \]

NewExpr11
\[ < e, env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_1, eVal_1 > \]
\[ < \text{GetValue}, eVal_1, \sigma_1 > \downarrow 6 < \sigma_2, \text{val(loc(l))} > \]
\[ \sigma_2 = (\text{map}, \text{top}) \quad \text{map}(l) = (m, \text{pr}, cl, \text{ext}, (c, b, il, env_0)) \]
\[ m_a = \{ \text{"length"}, (\text{Len}(al), \text{false}, \text{false}, \text{false}) \} \]
\[ \text{args} = (m_a, \text{OPO}, \text{"Arguments"}, \text{true}, \text{null}) \]
\[ env_2 = (\{ \text{(arguments, (top, true))}, env_0 \}) \quad m(\text{"prototype"}) = (\text{loc}(l_2), wr, en, co) \quad this_{obj} = (\emptyset, l_2, \text{"Object"}, \text{true}, \text{null}) \]
\[ map_2 = \text{map}[\text{top/args}, (\text{top} + 1)/env_2, (\text{top} + 2)/this_{obj}] \]
\[ < il, al, env, \text{top} + 1, 0, (map_2, \text{top} + 3) > \downarrow 5 < \sigma_3, \text{val}(v) > \]
\[ < body, \text{top} + 1, (\text{top} + 2), \sigma_3 > \downarrow 3 < \sigma_4, (\text{return}, v_4, \text{empty}) > \quad v_4 \in \text{PrimitiveValue} \]
\[ < e ( al ), env, thisVal, \sigma, eVal > \downarrow 4 < \sigma_4, \text{val(loc}(top + 2)) > \]
Formal Semantics of JavaScript

Function

\[ \text{FunctionExpr} \]

\[ (m, \text{top}) = \sigma \]

\[ \text{proto} = \{(\text{"constructor"}, (\text{top} + 1, \text{true}, \text{false}, \text{true}))\}, \text{OPO}, \text{"Object"}, \text{true}, \text{null} \]

\[ \text{newEnv} = \{(i, (\text{loc(top} + 1), \text{true}))\}, \text{env} \]

\[ m_2 = \{(\text{"length"}, (\text{Len}(\text{il}), \text{false}, \text{false}, \text{false}))\}, \{(\text{"prototype"}, (\text{top}, \text{true}, \text{false}, \text{false}))\} \]

\[ f = (m_2, \text{FPO}, \text{"Function"}, \text{true}, (\text{body}, \text{il}, \text{newEnv}, \text{true})) \]

\[ \sigma_2 = (\text{map[top/proto}, (\text{top} + 1)/f, (\text{top} + 2)/\text{newEnv}], \text{top} + 3) \]

\[ < \text{function } i ( \text{ il } ) \text{ decl stmt, env, thisVal, } \sigma, eVal >_{\downarrow 4} < \sigma_4, \text{loc(top} + 1) > \]

BinOpExpr1

\[ < e_1, \text{env, thisVal, } \sigma, eVal >_{\downarrow 4} < \sigma_1, \text{exprVal}_1 > \]

\[ < \text{GetValue, exprVal}_1, \sigma_1 >_{\downarrow 6} < \sigma_2, \text{exprVal}_2 > \]

\[ < e_2, \text{env, thisVal, } \sigma_2, \text{exprVal}_2 >_{\downarrow 4} < \sigma_3, \text{exprVal}_3 > \]

\[ < \text{GetValue, exprVal}_3, \sigma_3 >_{\downarrow 6} < \sigma_4, \text{exc(e)} > \quad \text{binOp} \notin \{\&\&\}, || \}

\[ < e_1 \text{ binOp e}_2, \text{env, thisVal, } \sigma, eVal >_{\downarrow 4} < \sigma_3, \text{exc(e)} > \]

BinOpExpr2

\[ < e_1, \text{env, thisVal, } \sigma, eVal >_{\downarrow 4} < \sigma_1, \text{exprVal}_1 > \]

\[ < \text{GetValue, exprVal}_1, \sigma_1 >_{\downarrow 6} < \sigma_2, \text{val(v}_1) > \]

\[ < e_2, \text{env, thisVal, } \sigma_2, \text{val(v}_1) >_{\downarrow 4} < \sigma_3, \text{exprVal}_3 > \]

\[ < \text{GetValue, exprVal}_3, \sigma_3 >_{\downarrow 6} < \sigma_4, \text{val(v}_2) > \]

\[ < \text{ToNumber, val(v}_1), \sigma_4 >_{\downarrow 6} < \sigma_5, \text{exc(e)} > \quad \text{binOp} \in \{\ast, /, -, <, >, \&\&\}, || \}

\[ < e_1 \text{ binOp e}_2, \text{env, thisVal, } \sigma, eVal >_{\downarrow 4} < \sigma_5, \text{exc(e)} > \]
BinOpExpr3

\[
\begin{align*}
&< e_1, env, thisVal, \sigma, eVal > ↓ 4 < σ_1, exprVal_1 > \\
&< \text{GetValue}, exprVal_1, σ_1 > ↓ 6 < σ_2, \text{val}(v_1) > \\
&< e_2, env, thisVal, σ_2, \text{val}(v_1) > ↓ 4 < σ_3, exprVal_3 > \\
&< \text{GetValue}, exprVal_3, σ_3 > ↓ 6 < σ_4, \text{val}(v_2) > \\
&< \text{ToNumber}, \text{val}(v_1), σ_4 > ↓ 6 < σ_5, \text{val}(v_3) > \\
&< \text{ToNumber}, \text{val}(v_2), σ_5 > ↓ 6 < σ_6, \text{exc}(e) > \\
&\text{binOp} \in \{+, /, -, <<, >>, \&\} \\
&< e_1 \text{ binOp } e_2, env, thisVal, σ, eVal > ↓ 4 < σ_6, \text{exc}(e) >
\end{align*}
\]

BinOpExpr4

\[
\begin{align*}
&< e_1, env, this, σ, eVal > ↓ 4 < σ_1, exprVal_1 > \\
&< \text{GetValue}, exprVal_1, σ_1 > ↓ 6 < σ_2, exprVal_2 > \\
&< e_2, env, this, σ_2, exprVal_2 > ↓ 4 < σ_3, exprVal_3 > \\
&< \text{GetValue}, exprVal_3, σ_3 > ↓ 6 < σ_4, exprVal_4 > \\
&< \text{ToNumber}, exprVal_2, σ_3 > ↓ 6 < σ_4, \text{val}(n_1) > \\
&< \text{ToNumber}, exprVal_4, σ_3 > ↓ 6 < σ_4, \text{val}(n_2) > \\
&\text{binOp} \in \{*, /, -, <<, >>, \&\} \\
&< e_1 \text{ binOp } e_2, env, this, σ, eVal > ↓ 4 < σ_6, \text{val}(\text{NumOp(binOp, n_1, n_2)}) >
\end{align*}
\]

BinOpExpr5

\[
\begin{align*}
&< e_1, env, thisVal, σ, eVal > ↓ 4 < σ_1, exprVal_1 > \\
&< \text{GetValue}, exprVal_1, σ_1 > ↓ 6 < σ_2, \text{val}(v_2) > \\
&< e_2, env, thisVal, σ_2, \text{val}(v_2) > ↓ 4 < σ_3, exprVal_3 > \\
&< \text{GetValue}, exprVal_3, σ_3 > ↓ 6 < σ_4, \text{val}(v_4) > \\
&< \text{ToPrimitive}, \text{val}(v_2), σ_4 > ↓ 6 < σ_5, \text{exc}(e) > \\
&\text{binOp} \in \{+, <\} \\
&< e_1 \text{ binOp } e_2, env, thisVal, σ, eVal > ↓ 4 < σ_5, \text{exc}(e) >
\end{align*}
\]

BinOpExpr6

\[
\begin{align*}
&< e_1, env, thisVal, σ, eVal > ↓ 4 < σ_1, exprVal_1 > \\
&< \text{GetValue}, exprVal_1, σ_1 > ↓ 6 < σ_2, exprVal_2 > \\
&< e_2, env, thisVal, σ_2, exprVal_2 > ↓ 4 < σ_3, exprVal_3 > \\
&< \text{GetValue}, exprVal_3, σ_3 > ↓ 6 < σ_4, \text{val}(v_4) > \\
&< \text{ToPrimitive}, \text{val}(v_2), σ_4 > ↓ 6 < σ_5, \text{val}(v_3) > \\
&< \text{ToPrimitive}, \text{val}(v_4), σ_5 > ↓ 6 < σ_6, \text{exc}(e) > \\
&\text{binOp} \in \{+, <\} \\
&< e_1 \text{ binOp } e_2, env, thisVal, σ, eVal > ↓ 4 < σ_6, \text{exc}(e) >
\end{align*}
\]
BinOpExpr7
\[ < e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \]
\[ < \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, exprVal_2 > \]
\[ < e_2, env, thisVal, \sigma_2, exprVal_2 > \downarrow_4 < \sigma_3, exprVal_3 > \]
\[ < \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_4) > \]
\[ < \text{ToPrimitive}, val(v_2), \sigma_4 > \downarrow_6 < \sigma_5, val(p_1) > \]
\[ < \text{ToPrimitive}, val(v_1), \sigma_5 > \downarrow_6 < \sigma_6, val(p_2) > \quad (p_1 \in \text{String} \lor p_2 \in \text{String}) \]
\[ < e_1 + e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_6, val(Str(p_1) + Str(p_2)) > \]

BinOpExpr8
\[ < e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \]
\[ < \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, exprVal_2 > \]
\[ < e_2, env, thisVal, \sigma_2, exprVal_2 > \downarrow_4 < \sigma_3, exprVal_3 > \]
\[ < \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_4) > \]
\[ < \text{ToPrimitive}, val(v_2), \sigma_4 > \downarrow_6 < \sigma_5, val(p_1) > \]
\[ < \text{ToPrimitive}, val(v_1), \sigma_5 > \downarrow_6 < \sigma_6, val(p_2) > \quad p_1 \notin \text{String} \quad p_2 \notin \text{String} \]
\[ < e_1 + e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_6, val(\text{Num}(p_1) + \text{Num}(p_2)) > \]

BinOpExpr9
\[ < e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \]
\[ < \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, val(v_1) > \]
\[ < e_2, env, thisVal, \sigma_2, val(v_1) > \downarrow_4 < \sigma_3, exprVal_3 > \]
\[ < \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_2) > \]
\[ v_2 \in \text{PrimitiveValue} \quad \sigma_4 = (\text{map}, \text{top}) \]
\[ \text{error} = (\emptyset, TEPO, \text{"Error"}, \text{true}, \text{null}) \quad \sigma_5 = (\text{map[top/error], top + 1}) \]
\[ < e_1 \text{ in } e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{exc}(\text{top}) > \]

BinOpExpr10
\[ < e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \]
\[ < \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, val(v_1) > \]
\[ < e_2, env, thisVal, \sigma_2, val(v_1) > \downarrow_4 < \sigma_3, exprVal_3 > \]
\[ < \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(loc(l)) > \]
\[ < \text{ToString}, val(v_1), \sigma_4 > \downarrow_6 < \sigma_5, \text{exc}(e) > \]
\[ < e_1 \text{ in } e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{exc}(e) > \]
Formal Semantics of JavaScript

BinOpExpr11
< $e_1, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_1, exprVal_1 >$
< GetValue, exprVal_1, \sigma_1 >_{\downarrow} < \sigma_2, val(v_1) >
< $e_2, \text{env}, thisVal, \sigma_2, val(v_1) >_{\downarrow} < \sigma_3, exprVal_3 >$
< GetValue, exprVal_3, \sigma_3 >_{\downarrow} < \sigma_4, val(loc(l)) >
< ToString, val(v_1), \sigma_4 >_{\downarrow} < \sigma_5, val(s) >

< $e_1$ in $e_2, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_5, val(HasProperty(s, l, \sigma_5)) >$

BinOpExpr12
< $e_1, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_1, exprVal_1 >$
< GetValue, exprVal_1, \sigma_1 >_{\downarrow} < \sigma_2, val(v_1) >
< $e_2, \text{env}, thisVal, \sigma_2, val(v_1) >_{\downarrow} < \sigma_3, exprVal_3 >$
< GetValue, exprVal_3, \sigma_3 >_{\downarrow} < \sigma_4, val(v_2) >
v_2 \in PrimitiveValue \quad \sigma_4 = (map, top)

error = (\emptyset, TEPO, 'Error', true, null) \quad \sigma_5 = (map[top/error], top + 1)

< $e_1$ instanceof $e_2, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_5, exc(top) >$

BinOpExpr13
< $e_1, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_1, exprVal_1 >$
< GetValue, exprVal_1, \sigma_1 >_{\downarrow} < \sigma_2, val(v_1) >
< $e_2, \text{env}, thisVal, \sigma_2, val(v_1) >_{\downarrow} < \sigma_3, exprVal_3 >$
< GetValue, exprVal_3, \sigma_3 >_{\downarrow} < \sigma_4, val(loc(l)) >

\sigma_4 = (map, top) \quad map(l) = (m, pr, cl, ex, f) \quad f \notin Function

error = (\emptyset, TEPO, 'Error', true, null) \quad \sigma_5 = (map[top/error], top + 1)

< $e_1$ instanceof $e_2, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_5, exc(top) >$

BinOpExpr14
< $e_1, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_1, exprVal_1 >$
< GetValue, exprVal_1, \sigma_1 >_{\downarrow} < \sigma_2, val(v_1) >
< $e_2, \text{env}, thisVal, \sigma_2, val(v_1) >_{\downarrow} < \sigma_3, exprVal_3 >$
< GetValue, exprVal_3, \sigma_3 >_{\downarrow} < \sigma_4, val(loc(l)) > \quad \sigma_4 = (map, top)

map(l) = (m, pr, cl, ex, f) \quad f \notin Function \quad v_1 \in PrimitiveValue

< $e_1$ instanceof $e_2, \text{env}, thisVal, \sigma, eVal >_{\downarrow} < \sigma_5, val(false) >
\begin{align*}
\text{BinOpExpr15} & \quad < e_1, env, thisVal, \sigma, eVal > \Downarrow_4 < \sigma_1, exprVal_1 > \\
& < \text{GetValue}, exprVal_1, \sigma_1 > \Downarrow_6 < \sigma_2, \text{val}(loc(l_1)) > \\
& < e_2, env, thisVal, \sigma_2, \text{val}(v_1) > \Downarrow_4 < \sigma_3, exprVal_3 > \\
& < \text{GetValue}, exprVal_3, \sigma_3 > \Downarrow_6 < \sigma_4, \text{val}(loc(l_2)) > \\
& \quad \sigma_4 = (\text{map}, \text{top}) \quad \text{map}(l_2) = (m, \text{pr}, \text{cl}, \text{ex}, f) \\
& f \in \text{Function} \quad < \text{GetValue}, \text{ref}(loc(l_2), \text{'prototype'}), \sigma_4 > \Downarrow < \sigma_4, \text{val}(\text{proto}) > \\
& \quad \text{error} = (\emptyset, \text{TEPO}, \text{'Error'}, \text{true}, \text{null}) \quad \sigma_5 = (\text{map}[\text{top/error}], \text{top} + 1) \\
& < e_1 \text{ instanceof } e_2, env, thisVal, \sigma, eVal > \Downarrow_4 < \sigma_5, \text{exc}(\text{top}) > \\
\end{align*}

\begin{align*}
\text{BinOpExpr16} & \quad < e_1, env, this, \sigma, eVal > \Downarrow_4 < \sigma_1, exprVal_1 > \\
& < \text{GetValue}, exprVal_1, \sigma_1 > \Downarrow_6 < \sigma_2, \text{val}(loc(l_1)) > \\
& < e_2, env, this, \sigma_2, \text{val}(v_1) > \Downarrow_4 < \sigma_3, exprVal_3 > \\
& < \text{GetValue}, exprVal_3, \sigma_3 > \Downarrow_6 < \sigma_4, \text{val}(loc(l_2)) > \\
& \quad \sigma_4 = (\text{map}, \text{top}) \quad \text{map}(l_2) = (m, \text{pr}, \text{cl}, \text{ex}, f) \\
& f \in \text{Function} \quad < \text{GetValue}, \text{ref}(loc(l_2), \text{'prototype'}), \sigma_4 > \Downarrow < \sigma_4, \text{val}(\text{loc}(\text{proto})) > \\
& < e_1 \text{ instanceof } e_2, env, this, \sigma, eVal > \Downarrow_4 < \sigma_4, \text{val}(\text{HasInstance}(l_1, l_2, \sigma_4)) > \\
\end{align*}

\text{HasInstance}(l_1 : \text{Loc}, l_2 : \text{Loc}, \sigma : \text{Store}) \\
\quad = (l_1 = l_2) \lor (\text{pr} \in \text{Loc} \land \text{HasInstance}(\text{pr}, l_2, \sigma)),

where \text{pr} \text{ defined in } \sigma(l_1) = (m, \text{pr}, \text{cl}, \text{ext}, \text{prim})

\begin{align*}
\text{BinOpExpr17} & \quad < e_1, env, thisVal, \sigma, eVal > \Downarrow_4 < \sigma_1, exprVal_1 > \\
& < \text{GetValue}, exprVal_1, \sigma_1 > \Downarrow_6 < \sigma_2, exprVal_2 > \\
& < e_2, env, thisVal, \sigma_2, exprVal_2 > \Downarrow_4 < \sigma_3, exprVal_3 > \\
& < \text{GetValue}, exprVal_3, \sigma_3 > \Downarrow_6 < \sigma_4, \text{val}(v_4) > \\
& < \text{ToPrimitive}, exprVal_2, \sigma_4 > \Downarrow_6 < \sigma_5, \text{val}(p_1) > \\
& < \text{ToPrimitive}, \text{val}(v_4), \sigma_5 > \Downarrow_6 < \sigma_6, \text{val}(p_2) > \\
& \quad p_1 \in \text{String} \quad p_2 \in \text{String} \\
& < e_1 < e_2, env, thisVal, \sigma, eVal > \Downarrow_4 < \sigma_6, \text{val}(p_1 < \text{str} \ p_2) >
\end{align*}
Formal Semantics of JavaScript

BinOpExpr18
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< GetValue, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, exprVal_2 > \\
< e_2, env, thisVal, \sigma_2, exprVal_2 > \downarrow_4 < \sigma_3, exprVal_3 > \\
< GetValue, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_4) > \\
< ToPrimitive, exprVal_2, \sigma_4 > \downarrow_6 < \sigma_5, val(p_1) > \\
< ToPrimitive, val(v_4), \sigma_5 > \downarrow_6 < \sigma_6, val(p_2) > \quad (p_1 \notin \text{String} \lor p_2 \notin \text{String}) \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_6, val(\text{Num}(p_1)) < \text{Num} \text{ Num}(p_2) >
\]

BinOpExpr19
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< GetValue, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, val(loc(l)) > \\
< e_2, env, thisVal, \sigma_2, val(v_2) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< GetValue, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_4) > \\
v_4 \in \text{Number} \cup \text{String} \quad < \text{ToPrimitive}, val(loc(l)), \sigma_4 > \downarrow_6 < \sigma_5, \text{exc}(e) > \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{exc}(e) >
\]

BinOpExpr20
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< GetValue, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, val(loc(l)) > \\
< e_2, env, thisVal, \sigma_2, val(v_2) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< GetValue, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(v_4) > \\
v_4 \in \text{Number} \cup \text{String} \quad < \text{ToPrimitive}, val(loc(l)), \sigma_4 > \downarrow_6 < \sigma_5, val(v_5) > \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, val(v_4 = \text{NonStrict} v_5) >
\]

BinOpExpr21
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< GetValue, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, val(v_2) > \\
< e_2, env, thisVal, \sigma_2, val(v_2) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< GetValue, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, val(loc(l)) > \\
v_2 \in \text{Number} \cup \text{String} \quad < \text{ToPrimitive}, val(loc(l)), \sigma_4 > \downarrow_6 < \sigma_5, \text{exc}(e) > \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{exc}(e) >
\]
Formal Semantics of JavaScript

\[ \text{BinOpExpr22} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v_2) > \\
< e_2, env, thisVal, \sigma_2, \text{val}(v_2) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, \text{val}(\text{loc}(l)) > \\
v_2 \in \text{Number} \cup \text{String} \quad < \text{ToPrimitive}, \text{val}(\text{loc}(l)), \sigma_4 > \downarrow_6 < \sigma_5, \text{val}(v_5) > \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{val}(v_4 = \text{NonStrict} \ v_5) >
\]

\[ \text{BinOpExpr23} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v_2) > \\
< e_2, env, thisVal, \sigma_2, \text{val}(v_2) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, \text{val}(v_4) > \\
(v_2 \neq \text{loc}(l_1) \lor v_4 \notin \text{Number} \cup \text{String}) \quad (v_2 \notin \text{Number} \cup \text{String} \lor v_4 \neq \text{loc}(l_2)) \\
< e_1 < e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_5, \text{val}(v_4 = \text{NonStrict} \ v_4) >
\]

\[ \&\&\text{Expression1} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{exc}(e) > \\
< e_1 \&\& e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_2, \text{exc}(e) >
\]

\[ \&\&\text{Expression2} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \\
< e_1 \&\& e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_2, \text{val}(v) > \\
\text{Bool}(v)
\]

\[ \&\&\text{Expression3} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \\
\text{Bool}(v) \quad < e_2, env, thisVal, \sigma_2, \text{val}(v) > \downarrow_4 < \sigma_3, exprVal_3 > \\
< \text{GetValue}, exprVal_3, \sigma_3 > \downarrow_6 < \sigma_4, exprVal_4 > \\
< e_1 \&\& e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_2, exprVal_4 >
\]

\[ ||\text{Expression1} \]
\[
< e_1, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_1, exprVal_1 > \\
< \text{GetValue}, exprVal_1, \sigma_1 > \downarrow_6 < \sigma_2, \text{exc}(e) > \\
< e_1 \mid e_2, env, thisVal, \sigma, eVal > \downarrow_4 < \sigma_2, \text{exc}(e) >
\]
Expression2
\[
\begin{array}{l}
\langle e_1, env, thisVal, \sigma, eVal \rangle \downarrow_4 \langle \sigma_1, exprVal_1 \rangle \\
\langle \text{GetValue}, exprVal_1, \sigma_1 \rangle \downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\text{Bool}(v)
\end{array}
\]
\[
\begin{array}{l}
\langle e_1, env, thisVal, \sigma, eVal \rangle \downarrow_4 \langle \sigma_1, exprVal_1 \rangle \\
\langle \text{GetValue}, exprVal_1, \sigma_1 \rangle \downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\end{array}
\]

Expression3
\[
\begin{array}{l}
\langle e_1, env, thisVal, \sigma, eVal \rangle \downarrow_4 \langle \sigma_1, exprVal_1 \rangle \\
\langle \text{GetValue}, exprVal_1, \sigma_1 \rangle \downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\text{Bool}(v)
\end{array}
\]
\[
\begin{array}{l}
\langle e_1, env, thisVal, \sigma, eVal \rangle \downarrow_4 \langle \sigma_1, exprVal_1 \rangle \\
\langle \text{GetValue}, exprVal_1, \sigma_1 \rangle \downarrow_6 \langle \sigma_2, \text{val}(v) \rangle \\
\end{array}
\]

A.2.6 Argument Lists

ArgumentsList1
\[
\langle \text{empty}, \text{empty}, env, thisVal, \text{argNr}, \sigma, \text{val} \rangle \downarrow_5 \langle \sigma, \text{val} \rangle
\]

ArgumentsList2
\[
\begin{array}{l}
(map, top) = \sigma \\
(d, outer) = map(env) \\
d \in \text{DeclarativeEnvironment}
\end{array}
\]
\[
\begin{array}{l}
d_2 = d[i/(\text{undefined}, \text{false})] \\
\sigma_2 = (map[env]/(d_2, outer)), top
\end{array}
\]
\[
\langle i, il, \text{empty}, env, thisVal, \text{argNr}, \sigma, \text{val} \rangle \downarrow_5 \langle \sigma_3, \text{val} \rangle
\]

ArgumentsList3
\[
\begin{array}{l}
(map, top) = \sigma \\
(d, outer) = map(env)
\end{array}
\]
\[
\begin{array}{l}
d \in \text{DeclarativeEnvironment} \\
\langle e, outer, thisVal, \sigma, \text{val} \rangle \downarrow_4 \langle \sigma_1, \text{val}_2 \rangle
\end{array}
\]
\[
\begin{array}{l}
\langle \text{GetValue}, \text{val}_2, \sigma_1 \rangle \downarrow_6 \langle \sigma_2, \text{v} \rangle \\
\text{IsException}(v)
\end{array}
\]
\[
\langle \text{empty}, e, al, env, thisVal, \text{argNr}, \sigma, \text{val} \rangle \downarrow_5 \langle \sigma_2, \text{v} \rangle
\]
ArgumentsList4

\[(\text{map}, \text{top}) = \sigma \quad (d, \text{outer}) = \text{map}(\text{env})\]

\[d \in \text{DeclarativeEnvironment} \quad < e, \text{outer}, \text{thisVal}, \sigma, \text{val} > \downarrow_4 < \sigma_1, \text{val}_2 > \]

\[< \text{GetValue}, \text{val}_2, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \quad (\text{map}_2, \text{top}_2) = \sigma_2\]

\[(\text{loc}(l), \text{true}) = d(\text{"arguments"}) \quad \text{obj}(m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}_2(l)\]

\[o_2 = \text{obj}(m[\text{String(argNr)}/(v, \text{true}, \text{true}, \text{true})], \text{pr}, \text{cl}, \text{ext}, \text{prim})\]

\[\sigma_3 = (\text{map}_2[l/o_2], \text{top}_2)\]

\[< \text{empty}, \text{al}, \text{env}, \text{thisVal}, \text{argNr} + 1, \sigma_3, \text{val}(\text{null}) > \downarrow_5 < \sigma_4, \text{val}_3 >\]

\[< \text{empty}, e \text{ al}, \text{env}, \text{thisVal}, \text{argNr}, \sigma, \text{val} > \downarrow_5 < \sigma_4, \text{val}_3 >\]

ArgumentsList5

\[(\text{map}, \text{top}) = \sigma \quad (d, \text{outer}) = \text{map}(\text{env})\]

\[d \in \text{DeclarativeEnvironment} \quad < e, \text{outer}, \text{thisVal}, \sigma, \text{val} > \downarrow_4 < \sigma_1, \text{val}_2 >\]

\[< \text{GetValue}, \text{val}_2, \sigma_1 > \downarrow_6 < \sigma_2, v > \quad \text{IsException}(v)\]

\[< \text{il}, e \text{ al}, \text{env}, \text{thisVal}, \text{argNr}, \sigma, \text{val} > \downarrow_5 < \sigma_2, v >\]

ArgumentsList6

\[(\text{map}, \text{top}) = \sigma \quad (d, \text{outer}) = \text{map}(\text{env})\]

\[d \in \text{DeclarativeEnvironment} \quad < e, \text{outer}, \text{thisVal}, \sigma, \text{val} > \downarrow_4 < \sigma_1, \text{val}_2 >\]

\[< \text{GetValue}, \text{val}_2, \sigma_1 > \downarrow_6 < \sigma_2, \text{val}(v) > \quad (\text{map}_2, \text{top}_2) = \sigma_2\]

\[(\text{loc}(l), \text{true}) = d(\text{"arguments"}) \quad \text{obj}(m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}_2(l)\]

\[o_2 = \text{obj}(m[\text{String(argNr)}/(v, \text{true}, \text{true}, \text{true})], \text{pr}, \text{cl}, \text{ext}, \text{prim})\]

\[d_2 = d[i/v] \quad \sigma_3 = (\text{map}_2[\text{env}/(d_2, \text{outer}), l/o_2], \text{top}_2)\]

\[< \text{il}, \text{al}, \text{env}, \text{thisVal}, \text{argNr} + 1, \sigma_3, \text{val}(\text{null}) > \downarrow_5 < \sigma_4, \text{val}_3 >\]

\[< \text{il}, e \text{ al}, \text{env}, \text{thisVal}, \text{argNr}, \sigma, \text{val} > \downarrow_5 < \sigma_4, \text{val}_3 >\]

A.2.7 Auxiliary Functions

\[\text{GetValue0}\]

\[
\text{ref}(\text{base, name}) \neq \text{val}
\]

\[< \text{GetValue}, \text{val}, \sigma > \downarrow_6 < \sigma, \text{val} >\]
GetValue1
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(d, \text{outer}) = \text{map}(\text{base}) \quad d \in \text{DeclarativeEnvironment} \quad (\text{val}_2, \text{b}) = \text{d}(\text{name}) \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma, \text{val}_2 >
\]

GetValue2
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(o, \text{null}) = \text{map}(\text{base}) \quad (m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}(o) \\
\text{HasProperty}(o, \text{name}, \sigma) \quad <\text{GetValue, ref}(o, \text{name}), \sigma \triangleright_6 <\sigma, \text{val}_2 > \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma, \text{val}_2 >
\]

GetValue3
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(o, \text{null}) = \text{map}(\text{base}) \quad (m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}(o) \\
\text{refErr} = (\emptyset, \text{REPO}, \text{"Error"}, \text{true}, \text{undefined}) \quad \sigma_2 = (\text{map}[\text{top}/\text{refErr}], \text{top} + 1) \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma_2, \text{exc}(\text{refErr}) >
\]

GetValue4
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}(\text{base}) \\
\text{name} \in \text{domain}(m) \quad (\text{val}_2, \text{b}_1, \text{b}_2, \text{b}_3) = m(\text{name}) \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma, \text{val}_2 >
\]

GetValue5
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}(\text{base}) \\
\text{name} \notin \text{domain}(m) \quad \text{pr} \neq \text{null} \quad <\text{GetValue, ref}(\text{pr}, \text{name}), \sigma \triangleright_6 <\sigma, \text{val}_2 > \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma, \text{val}_2 >
\]

GetValue6
\[
\text{ref}(\text{base}, \text{name}) = \text{val} \quad (\text{map}, \text{top}) = \sigma \\
(m, \text{pr}, \text{cl}, \text{ext}, \text{prim}) = \text{map}(\text{base}) \\
\text{name} \notin \text{domain}(m) \\
\text{VALUE} <\text{GetValue, val, } \sigma \triangleright_6 <\sigma, \text{val(undefined)} >
\]
Formal Semantics of JavaScript

GetValuer7

\[
\text{ref(undefined, name) = val}
\]

\[
\text{refErr} = (\emptyset, \text{REPO}, \text{'Error'}, \text{true}, \text{undefined})
\]

\[
\sigma_2 = (\text{map[top/refErr]}, \text{top} + 1)
\]

\[
< \text{GetValue, val, } \sigma >\downarrow_6 < \sigma_2, \text{exc(refErr)} >
\]

GetValues8

\[
\text{ref(base, name) = val}
\]

\[
\text{base} \neq \text{loc(l)} \quad \text{base} \neq \text{undefined} 
\]

\[
< \text{ToObject, base, } \sigma >\downarrow_6 < \sigma_2, \text{obj} >
\]

\[
< \text{GetValue, ref(obj, name), } \sigma_2 >\downarrow_6 < \sigma_2, \text{val}_2 >
\]

\[
< \text{GetValue, val, } \sigma >\downarrow_6 < \sigma_2, \text{val}_2 >
\]

ToStringHelper1

\[
\text{IsException(val)}
\]

\[
< \text{ToStringHelper, val, } \sigma >\downarrow_6 < \sigma, \text{val} >
\]

ToStringHelper2

\[
\text{val} \in \text{PrimitiveValue}
\]

\[
< \text{ToStringHelper, val(pv), } \sigma >\downarrow_6 < \sigma, \text{val(Str(pv))} >
\]

ToStringHelper3

\[
\text{loc(o) = val} 
\]

\[
< \text{GetValue, ref(loc(o), 'toString'), } \sigma >\downarrow_6 < \sigma, \text{loc(val}_2) >
\]

\[
\text{(map, top)} = \sigma \quad (m, pr, ex, cl, (ic, body, il, env)) = \text{map(val}_2)
\]

\[
\text{args} = (\{\text{'length'}, (0, \text{false}, \text{false}, \text{false})\}, \text{OPO, 'Arguments', true, null})
\]

\[
\text{d} = (\{\text{'arguments'}, (\text{loc(top)}, \text{true})\})
\]

\[
\sigma_2 = (\text{map[top/args, (top} + 1)/(d, env)], \text{top} + 2)
\]

\[
< \text{il, empty, (top} + 1), \text{val(null)}, 0, \sigma_2, \text{null} >\downarrow_5 < \sigma_3, \text{val}_3 >
\]

\[
< \text{stmt, (top} + 1), \text{loc(o), } \sigma_4 >\downarrow_1 < \sigma_4, \text{(type, val}_4, \text{target}) >
\]

\[
\text{type} \in \{\text{return, throw}\}
\]

\[
< \text{ToStringHelper, val, } \sigma >\downarrow_6 < \sigma_4, \text{val}_4 >
\]
Formal Semantics of JavaScript

81

ToStringHelper

loc(o) = val < GetValue, ref(loc(o), "toString"), σ >ψ6< σ, loc(val2) >

(map, top) = σ

args = ({{"length", (0, false, false, false)}}, OPO, "Arguments", true, null)

d = ({{"arguments", (loc(top), true)}})

σ2 = (map[top/args, (top + 1)/(d, env)], top + 2)

< il, empty, (top + 1), null, 0, σ2, val(null) >ψ3< σ3, val3 >

< body, (top + 1), loc(o), σ3 >ψ1< σ4, (normal, val4, target) >

< ToStringHelper, val, σ >ψ6< σ4, val(undefined) >

ToStringHelper5

loc(o) = val

< GetValue, ref(loc(o), "toString"), σ >ψ6< σ, val2 >

IsNullCallable(val2, σ)

< ToStringHelper, val, σ >ψ6< σ, val(loc(0)) >

ToString1

< ToStringHelper, val, σ >ψ6< σ2, val2 >

val(loc(o)) ≠ val2 < ToStringHelper, val2, σ2 >ψ6< σ2, val3 >

< ToString, val, σ >ψ6< σ2, val3 >

ToString2

< ToStringHelper, val, σ >ψ6< σ2, val(loc(o)) >

< ToNumberHelper, val, σ2 >ψ6< σ3, val2 >

val(loc(o2)) ≠ val2 < ToStringHelper, val2, σ2 >ψ6< σ2, val3 >

< ToString, val, σ >ψ6< σ2, val3 >

ToString3

< ToStringHelper, val, σ >ψ6< σ2, val(loc(o)) >

ToNumberHelper, val, σ2 >ψ6< σ3, val(loc(o2)) >

(map, top) = σ3

typeErr = (∅, TEPO, "Error", true, undefined) σ4 = (map[top/refErr], top + 1)

< ToString, val, σ >ψ6< σ4, exc(top) >

ToNumberHelper1

IsException(val)

< ToNumberHelper, val, σ >ψ6< σ, val >
\begin{align*}
\text{ToNumberHelper2} \\
pv \in \text{PrimitiveValue} \\
< \text{ToNumberHelper}, \text{val}(\pv), \sigma > \downarrow_6 < \sigma, \text{val}(\text{Str}(\pv)) >
\end{align*}

\begin{align*}
\text{ToNumberHelper3} \\
\text{loc}(o) = \text{val} & \quad < \text{GetValue, ref(loc(o), "valueOf"), } \sigma > \downarrow_6 < \sigma, \text{loc}(\text{val}_2) > \quad (\text{map, top}) = \sigma \quad (m, pr, ex, cl, (ic, body, il, env)) = \text{map}(\text{val}_2) \\
\text{args} = (\{(\text{"length"}, (0, \text{false}, \text{false}, \text{false}))\}, \text{OPO, "Arguments"}, \text{true, null}) \\
& \quad \quad d = (\{(\text{"arguments"}, (\text{loc}(\text{top}), \text{true}))\}) \\
& \quad \quad \sigma_2 = (\text{map}[\text{top}/\text{args}, (\text{top} + 1)/(d, env)], \text{top} + 2) \\
& \quad \quad < \text{il, empty, (top} + 1), \text{val(null), } 0, \sigma_2, \text{null} > \downarrow_5 < \sigma_3, \text{val}_3 > \\
& \quad \quad < \text{stmt, (top} + 1), \text{loc(o), } \sigma_4 > \downarrow_4 < \sigma_4, (\text{type, val}_4, \text{target}) > \\
& \quad \quad \quad \quad \text{type} \in \{\text{return}, \text{throw}\} \\
& \quad \quad \quad \quad \quad \text{---} \\
& \quad \quad < \text{ToNumberHelper}, \sigma > \downarrow_6 < \sigma_4, \text{val}_4 >
\end{align*}

\begin{align*}
\text{ToNumberHelper4} \\
\text{loc}(o) = \text{val} & \quad < \text{GetValue, ref(loc(o), "valueOf"), } \sigma > \downarrow_6 < \sigma, \text{loc}(\text{val}_2) > \quad (\text{map, top}) = \sigma \quad (m, pr, ex, cl, (ic, body, il, env)) = \text{map}(\text{val}_2) \\
\text{args} = (\{(\text{"length"}, (0, \text{false}, \text{false}, \text{false}))\}, \text{OPO, "Arguments"}, \text{true, null}) \\
& \quad \quad d = (\{(\text{"arguments"}, (\text{loc}(\text{top}), \text{true}))\}) \\
& \quad \quad \sigma_2 = (\text{map}[\text{top}/\text{args}, (\text{top} + 1)/(d, env)], \text{top} + 2) \\
& \quad \quad < \text{il, empty, (top} + 1), \text{val(null), } 0, \sigma_2, \text{null} > \downarrow_5 < \sigma_3, \text{val}_3 > \\
& \quad \quad < \text{body, (top} + 1), \text{loc(o), } \sigma_3 > \downarrow_4 < \sigma_4, (\text{normal, val}_4, \text{target}) > \\
& \quad \quad \text{---} \\
& \quad \quad < \text{ToStringHelper}, \sigma > \downarrow_6 < \sigma_4, \text{val}(\text{undefined}) >
\end{align*}

\begin{align*}
\text{ToNumberHelper5} \\
\text{loc}(o) = \text{val} \\
& \quad < \text{GetValue, ref(loc(o), "valueOf"), } \sigma > \downarrow_6 < \sigma, \text{val}_2 > \quad !\text{Callable}(\text{val}_2, \sigma) \\
& \quad < \text{ToNumberHelper}, \sigma > \downarrow_6 < \sigma, \text{val}(\text{loc}(0)) >
\end{align*}

\begin{align*}
\text{ToNumber1} \\
& < \text{ToNumberHelper, } \sigma > \downarrow_6 < \sigma_2, \text{val}_2 > \quad \text{val(loc(o))} \neq \text{val}_2 \\
& < \text{ToNumberHelper, } \text{val}_2, \sigma_2 > \downarrow_6 < \sigma_2, \text{val}_3 > \\
& < \text{ToNumber, } \sigma > \downarrow_6 < \sigma_2, \text{val}_3 >
\end{align*}
\[
\text{ToNumber2} \\
\quad \langle \text{ToNumberHelper}, \text{val}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{val(loc(o))} \rangle \\
\quad \langle \text{ToStringHelper}, \text{val}, \sigma_2 \rangle \downarrow_6 \langle \sigma_3, \text{val}_2 \rangle \\
\quad \text{val}(\text{loc}(o_2)) \neq \text{val}_2 \quad \langle \text{ToNumberHelper}, \text{val}_2, \sigma_2 \rangle \downarrow_6 \langle \sigma_2, \text{val}_3 \rangle \\
\quad \langle \text{ToNumber}, \text{val}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{val}_3 \rangle
\]

\[
\text{ToNumber3} \\
\quad \langle \text{ToNumberHelper}, \text{val}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{val(loc(o))} \rangle \\
\quad \langle \text{ToStringHelper}, \text{val}, \sigma_2 \rangle \downarrow_6 \langle \sigma_3, \text{val(loc(o_2))} \rangle \\
\quad (\text{map}, \text{top}) = \sigma_3 \quad \text{type} = \text{obj}(\emptyset, \text{TEPO}, \text{"Error"}, \text{true}, \text{undefined}) \\
\quad \sigma_4 = (\text{map}[\text{top}/\text{ref} \text{Err}], \text{top} + 1) \\
\quad \langle \text{ToNumber}, \text{val}, \sigma \rangle \downarrow_6 \langle \sigma_4, \text{exc}(\text{top}) \rangle
\]

\[
\text{ToObject1} \\
\quad \langle \text{ToObject}, \text{exc}(e), \sigma \rangle \downarrow_6 \langle \sigma_4, \text{exc}(e) \rangle
\]

\[
\text{ToObject2} \\
\quad \langle \text{ToObject}, \text{val(loc(l))}, \sigma \rangle \downarrow_6 \langle \sigma_4, \text{val(loc(l))} \rangle
\]

\[
\text{ToObject3} \\
\quad v \in \{\text{null, undefined}\} \\
\quad (\text{map}, \text{top}) = \sigma \quad \text{error} = (\emptyset, \text{TEPO}, \text{"Error"}, \text{true}, \text{null}) \\
\quad \sigma_2 = (\text{map}[\text{top}/\text{error}], \text{top} + 1) \\
\quad \langle \text{ToObject}, \text{val(v)}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{exc(loc(top))} \rangle
\]

\[
\text{ToObject3} \\
\quad v \in \text{Boolean} \\
\quad \text{obj} = (\emptyset, \text{BPO}, \text{"Boolean"}, \text{true}, v) \\
\quad \sigma_2 = (\text{map}[\text{top}/\text{obj}], \text{top} + 1) \\
\quad \langle \text{ToObject}, \text{val(v)}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{val(loc(top))} \rangle
\]

\[
\text{ToObject4} \\
\quad v \in \text{Number} \\
\quad \text{obj} = (\emptyset, \text{NPO}, \text{"Number"}, \text{true}, v) \\
\quad \sigma_2 = (\text{map}[\text{top}/\text{obj}], \text{top} + 1) \\
\quad \langle \text{ToObject}, \text{val(v)}, \sigma \rangle \downarrow_6 \langle \sigma_2, \text{val(loc(top))} \rangle
\]
ToObject5

\[
<v \in \text{String} \\
(map, top) = \sigma \\
obj = (\emptyset, SPO, \text{"String"}, \text{true}, v) \\
\sigma_2 = (map[\text{top/obj}], \text{top} + 1)
\]

\[
< \text{ToObject}, val(v), \sigma > \Downarrow_6 < \sigma_2, val(\text{loc(top)}) >
\]
Appendix B

Tagged Pointers

B.1 Member Expression

Call Expression

B.2 Call Expression
B.3 New Expression

\[
\begin{align*}
\text{Call Exp} & \quad \text{No of Args} \\
\text{Value} & \quad \text{Value}
\end{align*}
\]

B.4 Function Expression

\[
\begin{align*}
\text{Function Name} & \quad \text{No of Args} \\
\text{Function Body} & \quad \text{Function Body}
\end{align*}
\]

B.5 Function Declaration

\[
\begin{align*}
\text{Function Name} & \quad \text{No of Args} \\
\text{Function Body} & \quad \text{Function Body}
\end{align*}
\]
B.6 ALU Operation

\[ \text{Left Operand} \xrightarrow{\text{alu}} \text{Right Operand} \]

B.7 Assign Expression

\[ \text{Left Operand} \xrightarrow{\text{as}} \text{Right Operand} \]

B.8 Add Expression

\[ \text{Left Operand} \xrightarrow{\text{add}} \text{Right Operand} \]

B.9 Less than Expression

\[ \text{Left Operand} \xrightarrow{\text{lt}} \text{Right Operand} \]
B.10 Instanceof Expression

\[ \text{Left Operand} \quad \text{inst} \rightarrow \text{Right Operand} \]

B.11 In Expression

\[ \text{Left Operand} \quad \text{in} \rightarrow \text{Right Operand} \]

B.12 Equals Expression

\[ \text{Left Operand} \quad \text{eq} \rightarrow \text{Right Operand} \]

B.13 Logical Expression

\[ \text{Left Operand} \quad \text{log} \rightarrow \text{Right Operand} \]
B.14 Typeof Expression

B.15 Delete Expression

B.16 If Statement

B.17 Statement List
B.18 Return Statement

\[
\text{ret} \quad \bot \quad \text{any exp} \\
\text{RetVal}
\]

B.19 Throw Statement

\[
\text{th} \quad \bot \quad \text{any exp} \\
\text{RetVal}
\]

B.20 Labelled Statement

\[
\text{la} \quad \text{any stmt} \\
\text{Sym} \\
\text{Label}
\]

B.21 Try Statement

\[
\text{try} \quad \text{any stmt} \quad \text{p} \quad \text{finally Stmt} \\
\text{try Stmt} \quad \text{any stmt} \\
\text{sym} \\
\text{Identifier}
\]

\[
\text{catch Stmt} \quad \text{any stmt} \quad \text{p}
\]
B.22 Environment

\[
\text{env} \quad \perp \quad o \quad \text{global Object}
\]

\[
\text{env} \quad p \quad \text{env} \quad \text{outer Env} \quad \text{binding 1} \quad \text{val 1} \quad \text{binding n} \quad \text{val n}
\]

B.23 Object

\[
o \quad \text{prop} \quad \text{op} \quad \text{primVal}
\]

\[
\text{prop} \quad \text{any val} \quad \text{Property Value}
\]

\[
\text{pr} \quad \text{s} \quad \text{Property Name}
\]

B.24 Function

\[
o \quad \text{op} \quad f \quad \text{p} \quad \text{sl} \quad \text{Body}
\]

\[
\text{prop} \quad \text{proto} \quad \text{env} \quad \text{param list}
\]
B.25 Reference Pointer
Appendix C

FSM of the Control Unit

S0
   dispatch tag(EXP)

//nil
   VAL = THIS
   next-state = pop(CStack)
   CStack.SP = 0
   this
   CStack.SP \neq 0

HALT

nil
   VAL = EXP
   next-state = pop(CStack)
   \text{nil, u, n, b, s, o}

next-state = pop(CStack)
**S0**
- dispatch tag(\(\text{EXP}\))

try

- push(\text{LabelStack.SP}, \text{XStack})
- push(\text{CStack.SP}, \text{XStack})
- push(\text{CallStack.SP}, \text{XStack})
- push(\text{DStack.SP}, \text{XStack})
- push(\text{THIS}, \text{XStack})
- push(\text{ENV}, \text{XStack})
- push(\text{cdr(\(\text{EXP}\))}, \text{XStack})
- \(\text{EXP} = \text{car(\(\text{EXP}\))}\)
- push(\text{noException, CStack})
- next-state = S0

**noException**
- \(\text{EXP} = \text{cdr(pop(\text{XStack}))}\)
- \(\text{XStack.SP} -= 6\)
- next-state = 0
FSM of the Control Unit

**S0**
- dispatch tag(EXP)
  - sl
  - push(cdr(EXP), DStack)
  - EXP = car(EXP)
  - push(S1, CStack)
  - next-state = S0

**S2: If Statement**
- boolVal
  - EXP = car(pop(DStack))
  - next-state = S0

**S1: Statement List**
- EXP = pop(DStack)
  - next-state = S0

**S0**
- dispatch tag(EXP)
  - if
  - push(cdr(EXP), DStack)
  - EXP = car(EXP)
  - push(S2, CStack)
  - next-state = S0

**S1: Statement List**
- EXP = pop(DStack)
  - next-state = S0

\[ \text{boolVal is 1 if } \text{toBool}(VAL) \text{ yields true, otherwise 0.} \]
FSM of the Control Unit

**S0**
- dispatch tag(\(\text{EXP}\))

\[ ret \]
- \(\text{EXP} = \text{car}(\text{EXP})\)
- push(S3, CStack)
- next-state = S0

**S3: Return Statement**

\(\vdash \text{XSPeq}\)
- // unnecessary (same val)
  - XStack.SP = pop(CallStack)
  - Stack.SP = pop(CallStack)
  - CStack.SP = pop(CallStack)
  - LabelStack.SP = pop(CallStack)
  - THIS = pop(CallStack)
  - ENV = pop(CallStack)
  - next-state = pop(CStack)

\(\vdash !\text{XSPeq}\)
- \(\text{tag}(\text{EXP}) \neq p\)
  - \(\text{EXP} = \text{cdr}(\text{EXP})\)
  - next-state = S0

**S4: Return-finally-done**
- VAL = pop(DStack)
- next-state = S3

\(\vdash \text{ret} \)

\(\vdash !\text{XSPeq}\)
- \(\text{EXP} = \text{pop}(\text{XStack})\)
- ENV = pop(XStack)
- XStack.SP -= 5
- push(\(\text{VAL}\), DStack)
- push(S4, CStack)

\(\vdash \text{tag}(\text{EXP}) = p\)
- EXP = pop(XStack)
- ENV = pop(XStack)
- XStack.SP -= 5
- push(\(\text{VAL}\), DStack)
- push(S4, CStack)

\(\vdash \text{tag}(\text{EXP}) = p\)

\(\vdash \text{EXP} = \text{cdr}(\text{EXP})\)

\(\vdash \text{next-state} = \text{S0}\)

\(\vdash \text{XSPeq}\) yields 1 if the value at the top of Stack equals to the stack pointer of XStack.
**FSM of the Control Unit**

**S0**

dispatch tag(EXP)

\[
\text{EXP} = \text{car(EXP)} \\
\text{push}(S6, \text{CStack}) \\
\text{next-state} = S0
\]

**S6: Throw Exception**

\[
\text{XStack.SP} \neq 0 \\
\text{EXP} = \text{pop(XStack)} \\
\text{ENV} = \text{pop(XStack)} \\
\text{THIS} = \text{pop(XStack)} \\
\text{Stack.SP} = \text{pop(XStack)} \\
\text{CallStack.SP} = \text{pop(XStack)} \\
\text{CStack.SP} = \text{pop(XStack)} \\
\text{LabelStack.SP} = \text{pop(XStack)} \\
\text{tag(EXP)} = p \\
\text{push(LabelStack.SP, XStack)} \\
\text{push(CStack.SP, XStack)} \\
\text{push(CallStack.SP, XStack)} \\
\text{push(DStack.SP, XStack)} \\
\text{push(THIS, XStack)} \\
\text{push(ENV, XStack)} \\
\text{push(cdr(EXP), XStack)} \\
\text{EXP} = \text{car(EXP)} \\
\text{PROC} = \text{car(EXP)} \\
\text{EXP} = \text{cdr(EXP)} \\
\text{UNEV} = \text{cons(PROC, VAL, bi, 0)} \\
\text{UNEV} = \text{cons(UNEV, nil, p)} \\
\text{ENV} = \text{cons(ENV, UNEV, env)} \\
\text{push}(S8, \text{CStack}) \\
\text{next-state} = S0
\]

**Uncaught Exception**

\[
//\text{Exception in Catch} \\
\text{push}(\text{VAL}, \text{DStack}) \\
\text{push}(S7, \text{CStack}) \\
\text{next-state} = S0
\]

**S7: Ex-finally-done**

\[
\text{VAL} = \text{pop(DStack)} \\
\text{next-state} = S6
\]

**S8: Ex-catch-done**

\[
\text{EXP} = \text{pop(XStack)} \\
\text{ENV} = \text{pop(XStack)} \\
\text{XStack.SP} -= 5 \\
\text{next-state} = S0
\]

\[\text{cons}(C1, C2, T)\text{ creates a new pair with tag } T. \text{ cons}(C1, C2, T, F)\text{ creates a new pair with tag } T \text{ and the flags } F.\]
FSM of the Control Unit

\( S_0 \)

\[
\begin{align*}
\text{dispatch tag(} & \text{EXP}\text{)} \\
vd & \\
\text{UNEV} = (s, \text{EXP}) \\
\text{EXP} = & \text{cdr(ENV)} \\
push(S_9, \text{CStack}) \\
tag(\text{EXP}) = o & \\
\text{//object env rec} \\
\text{VAL} = & \text{EXP} \\
\text{next-state} = & \text{getProperty} \\
tag(\text{EXP}) \neq o & \\
\text{//lexical env rec} \\
\text{next-state} = & \text{hasBinding}
\end{align*}
\]

\( S_9: \text{Variable Declaration} \)

\[
\begin{align*}
\text{!boolVal} & \\
\text{EXP} = & \text{cdr(ENV)} \\
tag(\text{EXP}) \neq o & \\
\text{UNEV} = & \text{cons(UNEV, undefined, bi, 0)} \\
\text{UNEV} = & \text{cons(UNEV, EXP, p)} \\
cdr(\text{ENV}) = & \text{UNEV} \\
\text{next-state} = & \text{pop(CStack)} \\
tag(\text{EXP}) = o & \\
\text{ARGS} = & \text{car(EXP)} \\
\text{UNEV} = & \text{cons(UNEV, ARGS, p)} \\
\text{car(EXP)} = & \text{UNEV} \\
\text{next-state} = & \text{pop(CStack)}
\end{align*}
\]

\((T, P)\) changes the pointer type of \(P\) to \(T\).
**S0**

dispatch tag(EXP)

\[ sym \]

\[ UNEV = (s, EXP) \]

\[ ARG5 = ENV \]

\[ next-state = S10 \]

---

**S10: Variable Lookup**

\[ EXP = cdr(ARGS) \]

\[ tag(EXP) = o \]

//object env rec

\[ push(S11-1, CStack) \]

\[ VAL = EXP \]

\[ next-state = getProperty \]

\[ tag(EXP) \neq o \]

//lexical env rec

\[ push(S11, CStack) \]

\[ VAL = cons(UNEV, ARGS, ref) \]

\[ next-state = pop(CStack) \]

---

**S11: Variable Lookup**

\[ARGS = car(ARGS)\]

\[ VAL = cons(UNEV, ARGS, ref) \]

\[ next-state = S10 \]

\[ VAL = cons(UNEV, undefined, ref) \]

\[ next-state = pop(CStack) \]

\[ VAL = cons(UNEV, (env, -1), ref) \]

\[ next-state = pop(CStack) \]

---

**S11-1: Variable Lookup**

\[ tag(VAL) = u \]

\[ tag(VAL) \neq u \]

\[ VAL = cons(UNEV, undefined, ref) \]

\[ next-state = pop(CStack) \]

\[ VAL = cons(UNEV, (env, -1), ref) \]

\[ next-state = pop(CStack) \]
S0

dispatch tag(EXP)

push(S13, CStack)
push(cdr(EXP), DStack)
EXP = car(EXP)
push(EvalOpd2, CStack)
next-state = S0

S13: Assign

UNEV = pop(DStack)
EXP = cdr(UNEV)
UNEV = car(UNEV)
dispatch tag(EXP)

next-state = put
next-state = throwRefError
next-state = put
next-state = throwTypeError
next-state = setBinding
S0
dispatch tag(EXP)

mem

push(S15, CStack)
push(cdr(EXP), DStack)
EXP = car(EXP)
push(EvalOpd2, CStack)
next-state = S0

S15: Member Expression
EXP = pop(DStack)
tag(EXP) = nl, u
next-state = throwTypeError
tag(EXP) ≠ nl, u
push(EXP, DStack)
push(S16, CStack)
next-state = toString

S16: Member Expression
VAL = cons(VAL, pop(DStack), ref)
next-state = pop(CStack)
S0
dispatch tag(\text{EXP})

call
\begin{align*}
\text{UNEV} &= \text{cdr}(\text{EXP}) \\
\text{EXP} &= \text{car}(\text{EXP}) \\
push(\text{cdr}(\text{UNEV}), \text{DStack}) \\
push(\text{car}(\text{UNEV}), \text{DStack}) \\
push(S22, \text{CStack}) \\
push(S17, \text{CStack}) \\
\text{next-state} &= s0
\end{align*}

S17: Call Expression
\begin{align*}
\text{UNEV} &= \text{pop}(\text{DStack}) \\
\text{ARGS} &= \text{pop}(\text{DStack}) \\
push(\text{VAL}, \text{DStack}) \\
push(\text{ARGS}, \text{DStack}) \\
push(\text{UNEV}, \text{DStack}) \\
push(S18, \text{CStack}) \\
\text{next-state} &= \text{getValue}
\end{align*}

S18: Call Expression
\begin{align*}
\text{UNEV} &= \text{cons}((a,-3), \text{nil}, \text{op}, 01011) \\
\text{ARGS} &= \text{cons}((s, \text{"length"}), \text{pop}(\text{DStack}), \text{pr}, 101) \\
\text{ARGS} &= \text{cons}(\text{ARGS}, \text{nil}, p) \\
\text{ARGS} &= \text{cons}(\text{ARGS}, \text{UNEV}, o) \\
\text{UNEV} &= \text{pop}(\text{DStack})
\end{align*}

\[\text{tag}(\text{UNEV}) = \perp\]
\[\text{tag}(\text{UNEV}) \neq \perp\]

// no args
\begin{align*}
\text{PROC} &= \text{VAL} \\
\text{VAL} &= \text{pop}(\text{DStack}) \\
\text{next-state} &= \text{pop}(\text{CStack})
\end{align*}

\[\text{push}(\text{VAL}, \text{DStack}) \]
\[\text{push}(\text{ARGS}, \text{DStack}) \]
\[\text{ARGS} = \text{car}(\text{ARGS}) \]
\[\text{ARGs} = \text{car}(\text{ARGS}) \]
\[\text{next-state} = S19 \]
**S19: Call Expression**

- push(ARGS, DStack)
- VAL = cdr(UNEV)
- UNEV = car(UNEV)
- EXP = cdr(UNEV)

![Transition Diagram]

- tag(VAL) ≠ ⊥ ➔ push(VAL, DStack) push(S20, CStack)
- tag(VAL) = ⊥ ➔ //last operand push(S21, CStack)

- push(car(UNEV), DStack)
- next-state = S0

**S20: Call Expression**

- ARGS = cons(pop(DStack), VAL, pr, 111)
- ARGS = cons(ARGS, nil, p)
- UNEV = pop(DStack)
- cdr(pop(DStack)) = ARGS
- next-state = S19

**S21: Call Expression**

- ARGS = cons(pop(DStack), VAL, pr, 111)
- ARGS = cons(ARGS, nil, p)
- cdr(pop(DStack)) = ARGS
- ARGS = pop(DStack)
- PROC = pop(DStack)
- VAL = pop(DStack)
- next-state = pop(CStack)
S22: Call Expression

\[ \text{tag}(\text{PROC}) = o \]
\[ \text{PROC} = \text{cdr}(\text{PROC}) \]
\[ \text{PROC} = \text{cdr}(\text{PROC}, 5) \]
\[ \text{next-state} = \text{throwTypeError} \]

\[ \text{tag}(\text{PROC}) \neq o \]

\[ \text{push}(\text{ENV}, \text{CallStack}) \]
\[ \text{push}(\text{THIS}, \text{CallStack}) \]
\[ \text{push}(\text{LabelStack.SP}, \text{CallStack}) \]
\[ \text{push}(\text{CStack.SP}, \text{CallStack}) \]
\[ \text{push}(\text{DStack.SP}, \text{CallStack}) \]
\[ \text{push}(\text{XStack.SP}, \text{CallStack}) \]
\[ \text{ENV} = \text{car}(\text{PROC}) \]
\[ \text{ENV} = \text{cons}(\text{ENV}, \text{nil}, \text{env}) \]
\[ \text{push}(S24, \text{CStack}) \]
\[ \text{next-state} = S23 \]

S23: Call Expression

\[ \text{UNEV} = \text{cons}((s, \text{"arguments"}, \text{ARGS}, bi, 1)) \]
\[ \text{UNEV} = \text{cons}(\text{UNEV}, \text{nil}, p) \]
\[ \text{cdr}(\text{ENV}) = \text{UNEV} \]
\[ \text{UNEV} = \text{cdr}(\text{PROC}) \]
\[ \text{EXP} = \text{cdr}(\text{UNEV}) \]
\[ \text{UNEV} = \text{car}(\text{UNEV}) \]
\[ \text{ARGS} = \text{car}(\text{ARGS}) \]
\[ \text{ARGS} = \text{cdr}(\text{ARGS}) \]
\[ \text{next-state} = S23A \]
**S23A: Call Expression**

```plaintext
tag(UNEV) = ⊥  
tag(UNEV) ≠ ⊥  
next-state = pop(CStack)  
//bind arguments
```

PROC = car(UNEV)
PROC = cons((s, PROC), undefined, bi, 0)
THIS = cdr(ENV)
PROC = cons(PROC, THIS, p)
cdr(ENV) = PROC
UNEV = cdr(UNEV)
next-state = S23A

**S24: Call Expression**

```plaintext
tag(VAL) = ref  
tag(VAL) ≠ ref
```

THIS = cdr(VAL)
push(S24ret, CStack)
next-state = S0

THIS = undefined
push(S24ret, CStack)
next-state = S0

**S24ret**

VAL = undefined
CallStack.SP -= 4
THIS = pop(CallStack)
ENV = pop(CallStack)
next-state = pop(CStack)
The value of nameOK is set to 1 if the string in UNEV in not equal to 'NaN', 'Infinity' and 'undefined', otherwise 0.
FSM of the Control Unit

FD
ARGS = cdr(EXP)
EXP = car(ARGS)
PROC = cdr(ARGS)
VAL = cdr(EXP)
VAL = cons(VAL, PROC, p)
VAL = cons(ENV, VAL, f, 1)
VAL = cons((o, -4), VAL, op, 00011)
PROC = car(EXP)
PROC = cons((s, 'length'), PROC, pr, 000)
PROC = cons(PROC, nil, p)
EXP = cons((s, *constructor*), nil, pr, 101)
EXP = cons(EXP, nil, p)
ARGS = cons((o, -3), nil, op, 00001)
ARGS = cons(EXP, ARGs, o)
ARGS = cons((s, *prototype*), ARGs, pr, 100)
ARGS = cons(ARGs, PROC, p)
VAL = cons(ARGs, VAL, o)
EXP = car(EXP)
cdr(EXP) = VAL
next-state = pop(CStack)

S26
EXP = cdr(EVN)
ARGs = cons(UNEV, VAL, pr, 110)
UNEV = car(EXP)
ARGs = cons(ARGs, UNEV, p)
car(EXP) = ARGs
next-state = pop(CStack)

S26A
EXP = cdr(EVN)
next-state = setProperty

S27A
EXP = cdr(EVN)
next-state = setBinding

27
ARGs = cons(UNEV, VAL, bi, 0)
UNEV = cdr(ENV)
ARGs = cons(ARGs, UNEV, p)
cdr(ENV) = ARGs
next-state = pop(CStack)
FSM of the Control Unit

S0
dispatch tag(EXP)

push(EXP, DStack)
push(S29, CStack)
push(S28, CStack)
push(cdr(EXP), DStack)
EXP = car(EXP)
push(EvalOpd2, CStack)
next-state = S0

S28:ALU
UNEV = pop(DStack)
push(VAL, DStack)
VAL = UNEV
next-state = toNumber

S29:ALU
A = pop(DStack)
PROC = pop(DStack)
VAL = alu(VAL)
next-state = pop(CStack)

EvalOpd2
EXP = pop(DStack)
push(VAL, DStack)
next-state = S0

EXP & 15 ≠ 10
push(S28, CStack)
push(S28, CStack)

EXP & 15 = 10
push(cdr(EXP), DStack)

alu

EXP & 15
= 10

EXP & 15
̸= 10
FSM of the Control Unit

**S0**
- dispatch tag(EXP)
  
  push(EXP, DStack)
  push(cdr(EXP, DStack)
  EXP = car(EXP)
  push(S30, CStack)
  next-state = S0

**S30: LOGIC Expression**

A = VAL

VAL = pop(DStack)
PROC = pop(DStack)
EXP = alu(VAL)
next-state = S0

**S31: Typeof**

VAL = typeComb

next-state = pop(CState)

VAL = (s, "object")
next-state = pop(CStack)

VAL = (s, "function")
next-state = pop(CStack)

VAL = (s, "undefined")
next-state = pop(CStack)

log
type

tag(VAL) = u, nl, b, n, s

EXP = cdr(VAL)

EXP = cdr(EXP)

VAL = (s, "object")
next-state = pop(CStack)

VAL = (s, "function")
next-state = pop(CStack)

VAL = (s, "undefined")
next-state = pop(CStack)
FSM of the Control Unit

**S0**

- dispatch tag(EXP)

  - push(S33, CStack)
  - push(cdr(EXP), DStack)
  - EXP = car(EXP)
  - push(EvalOpd2, CStack)
  - next-state = S0

**S33:In**

- tag(VAL) = o
  - UNEV = VAL
  - VAL = pop(DStack)
  - push(UNEV, DStack)
  - push(S34, CStack)
  - next-state = toString

- tag(VAL) ≠ o
  - next-state = throwTypeError

**S34:In**

- UNEV = VAL
  - VAL = pop(DStack)
  - push(S35, CStack)
  - next-state = getProperty

- tag(VAL) = pr
  - VAL = (b, 1)
  - next-state = pop(CStack)

- tag(VAL) ≠ pr
  - VAL = (b, 0)
  - next-state = pop(CStack)
FSM of the Control Unit

**S0**

dispatch tag(\(\text{EXP}\))

\[\begin{array}{c}
\text{add} \\
push((\text{alu}, 2), \text{DStack}) \\
push(\text{S39}, \text{CStack}) \\
push(\text{S37}, \text{CStack}) \\
push(\text{cdr}(\text{EXP}), \text{DStack}) \\
\text{EXP} = \text{car}(\text{EXP}) \\
push(\text{EvalOpd2}, \text{CStack}) \\
\text{next-state} = \text{S0}
\end{array}\]

\[\begin{array}{c}
\text{S37:Add} \\
\text{UNEV} = \text{pop(\text{DStack})} \\
\text{push}(\text{VAL}, \text{DStack}) \\
\text{push}(\text{UNEV}, \text{DStack}) \\
\text{\text{next-state} = \text{toPrimitive}}
\end{array}\]

\[\begin{array}{c}
\text{S39:Add} \\
\text{tag(\text{VAL})} = s \text{ or} \\
\text{tag(\text{UNEV})} = s \\
\text{PROC} = \text{pop(\text{DStack})} \\
push(\text{S40}, \text{CStack}) \\
\text{next-state} = \text{toString}
\end{array}\]

\[\begin{array}{c}
\text{S40:Add} \\
\text{ARGS} = \text{VAL} \\
\text{VAL} = \text{UNEV} \\
push(\text{S41}, \text{CStack}) \\
\text{next-state} = \text{toString}
\end{array}\]

\[\begin{array}{c}
\text{S41:Add} \\
\text{IR.V1} = \text{VAL} \\
\text{IR.V2} = \text{ARGS} \\
\text{IR.Code} = '\text{strConcat}' \\
\text{//Interrupt pending} \\
\text{VAL} = \text{IR.Result} \\
\text{\\text{next-state} = pop(\text{CStack})}
\end{array}\]

\(\text{toNumberComb}\) performs a combinatorial type conversion to number for pointer types \(u, nl, b\) and \(n\).
FSM of the Control Unit

**S0**

```
dispatch tag(EXPR)
```

```
lr
push((alu, 6), DStack)
push(S42, CStack)
push(S37, CStack)
push(S37, CStack)
push(cdr(EXPR), DStack)
EXPR = car(EXPR)
push(EvalOpd2, CStack)
next-state = S0
```

**S42:LT**

```
UNEV = pop(DStack)
tag(VAL) = s and tag(UNEV) = s
```

```
PROC = pop(DStack)
IR.V1 = UNEV
IR.V2 = VAL
IR.Code = "strLt"
//Interrupt pending
VAL = IR.Result
next-state = pop(CStack)
```
FSM of the Control Unit

**S0**

dispatch tag(EXP)

\[ eq \]

push(S45, CStack)
push(cdr(EXP), DStack)
EXP = car(EXP)
push(EvalOpd2, CStack)
next-state = S0

**S45:Equals**

A = pop(DStack)
next-state = S46

**PROC** = \((alu, 10)\)
VAL = alu(VAL)
next-state = pop(CStack)

**PROC** = \((alu, 14)\)
VAL = alu(VAL)
next-state = pop(CStack)

\[ \text{tag}(VAL) = \text{tag}(A) \]
\[ \text{tag}(VAL) \neq \text{tag}(A) \]

**S46:Equals**

\[ (s,n) \text{ or } (b, s \text{ or } n) \]

\[ (n,s) \text{ or } (s \text{ or } n, b) \]

\[ (o,n \text{ or } s) \]

\[ (n \text{ or } s, o) \]

\[ \text{otherwise} \]

UNEV = A
A = VAL
VAL = UNEV

UNEV = A
A = VAL
VAL = UNEV

push(S46, CStack)
next-state = toNumber

push(S45, CStack)
push(A, DStack)
UNEV = \( (s, \text{valueOf}) \)
next-state = toPrimitive

\[ \text{tag}(VAL) = \text{tag}(A) \]
FSM of the Control Unit

S0
dispatch tag(EXP)

\[
\text{delimiter} \\
\text{EXP} = \text{car(EXP)} \\
\text{push}(S49, \text{CStack}) \\
\text{next-state} = S0
\]

Delete
ARGS = car(VAL)
next-state = Delete2

\[
\text{S49:Delete} \\
\text{tag(VAL)} = \text{ref} \\
\text{UNEV} = \text{car(VAL)} \\
\text{VAL} = \text{cdr(VAL)} \\
\text{push}(\text{Delete}, \text{CStack}) \\
\text{next-state} = \text{toObject} \\
\]

\[
\text{tag(VAL)} \neq \text{ref} \\
\text{VAL} = (b, 1) \\
\text{next-state} = \text{pop(CStack)}
\]

Delete2

\[
\text{tag}(\text{ARGS}) \neq \bot \\
\text{tag}(\text{ARGS}) = \bot \\
\text{PROC} = \text{car(ARGS)} \\
\text{PROC} = \text{car(PROC)} \\
\text{UNEV} \neq \text{PROC} \\
\]

\[
\text{VAL} = \text{ARGS} \\
\text{ARGS} = \text{cdr(ARGS)} \\
\text{next-state} = \text{Delete2} \\
\]

\[
\text{UNEV} = \text{car(ARGS)} \\
\text{UNEV} = \text{car(ARGS)} \\
\text{UNEV} = \text{PROC} \\
\text{UNEV} \& \ 1 \\
\text{!(UNEV} \& \ 1) \\
\text{tag}(\text{VAL}) \neq \text{a} \\
\text{tag}(\text{VAL}) = \text{o} \\
\]

\[
\text{cdr(VAL)} = \text{UNEV} \\
\text{VAL} = (b, 1) \\
\text{next-state} = \text{pop(CStack)} \\
\]

\[
\text{car(VAL)} = \text{UNEV} \\
\text{VAL} = (b, 1) \\
\text{next-state} = \text{pop(CStack)}
\]
**S0**

- dispatch tag(EXP)

\[\text{next-state} = S0\]

\[\text{inst}\]

- push(S51, CStack)
- push(cdr(EXP))
- EXP = car(EXP)
- push(EvalOpd2, CStack)

**S51: InstanceOf**

- tag(VAL) = o
- UNEV = cdr(VAL)
- UNEV = cdr(UNEV)
- tag(UNEV) = f
- VAL = pop(DStack)
- tag(UNEV) = o
- tag(VAL) ≠ o

**S53: InstanceOf**

- PROC = cdr(VAL)
- tag(PROC) = o \& tag(VAL) = pr
- UNEV = pop(DStack)
- next-state = hasInstance

- tag(PROC) ≠ o \| tag(VAL) ≠ pr

- next-state = throwTypeError

- VAL = (b, 0)

- next-state = pop(CStack)

- tag(VAL) ≠ o
FSM of the Control Unit

PROC ≠ UNEV

hasInstance

PROC = UNEV

tag(UNEV) = ⊥

UNEV = cdr(UNEV)

UNEV = car(UNEV, 5)

next-state = hasInstance

VAL = (b, 0)

next-state = pop(CStack)

VAL = (b, 1)

next-state = pop(CStack)
FSM of the Control Unit

**S0**
- dispatch tag(Exp)

new

- UNEV = cdr(Exp)
- EXP = car(Exp)
- push((nil, 0), DStack)
- push(cdr(UNEV), DStack)
- push(car(UNEV), DStack)
- push(S56, CStack)
- push(S22new, CStack)
- push(S18, CStack)
- push(getValue, CStack)

next-state = s0

**S22new: New Expression**

- tag(PROC) = o
- tag(PROC) ≠ o

- push(PROC, DStack)
- PROC = cdr(PROC)
- PROC = cdr(PROC)
- UNEV = PROC

next-state = throwTypeError

- tag(PROC) ≠ f ∨ !(UNEV & 1)

- push(ENV, CallStack)
- push(THIS, CallStack)
- push(LabelStack.SP, CallStack)
- push(CStack.SP, CallStack)
- push(DStack.SP, CallStack)
- push(XStack.SP, CallStack)
- ENV = car(PROC)
- ENV = cons(ENV, nil, env)
- push(S54, CStack)

next-state = S23
S54: New Expression
UNEV = pop(DStack)
push(EXP, DStack)
EXP = car(UNEV)
UNEV = (s, "prototype")
push(S55, CStack)
next-state = getOwnProperty

S55: New Expression
VAL = cdr(VAL)
tag(V) ≠ o
VAL = (o, -3)
tag(V) = o
VAL = cons(V, nil, op, 00001)
THIS = cons(nil, VAL, o)
EXP = pop(DStack)
push(THIS, DStack)
push(S24ret, CStack)
next-state = S0

S56: New Expression
tag(V) = o
pop(DStack) // discard
next-state = pop(CStack)
tag(V) ≠ o
VAL = pop(DStack)
next-state = pop(CStack)
**S0**

- dispatch tag(\text{EXP})

\[ fe \]

- UNEV = car(\text{EXP})
- push(FE, CStack)
- next-state = FD

**FE**

- ARG\text{S} = \text{cdr}(\text{VAL})
- ARG\text{S} = \text{cdr}(\text{ARG\text{S}}, 5)
- UNE\text{V} = \text{cons}((s, UNE\text{V}), \text{VAL, bi, 1})
- UNE\text{V} = \text{cons}(\text{UNE\text{V}}, \text{nil, p})
- UNE\text{V} = \text{cons}(\text{ENV, UNE\text{V}}, \text{env})
- car(\text{ARG\text{S}}) = UNE\text{V}
- next-state = pop(CStack)
S0
dispatch tag(EXP)

la
push(DStack.SP, LabelStack)
push(CStack.SP, LabelStack)
push(cdr(EXP), LabelStack)
push(XStack.SP, LabelStack)
push(car(EXP), LabelStack)
EXP = cdr(EXP)
push(S57, CStack)
next-state = S0

S57:Labelled Stmt
LabelStack.SP -= 5
next-state = pop(CStack)
FSM of the Control Unit

**S0**

dispatch tag(EXP)

\[ \text{PROC} = \text{pop(LabelStack)} \]
\[ \text{UNEV} = (\text{sym}, \text{EXP}) \]

\[ \text{PROC} = \text{UNEV} \]

\[ \text{PROC} = \text{UNEV} \]

\[ \text{UNEV} = \text{pop(LabelStack)} \]
\[ \text{VAL} = \text{PROC} \]
\[ \text{PROC} = \text{XStack.SP} \]

\[ \text{push(EXP, DStack)} \]
\[ \text{push(S58, CStack)} \]
\[ \text{push(UNEV, LabelStack)} \]
\[ \text{push(VAL, LabelStack)} \]
\[ \text{EXP} = \text{pop(XStack)} \]
\[ \text{ENV} = \text{pop(XStack)} \]
\[ \text{XStack.SP} -= 5 \]

\[ \text{tag(EXP)} \neq p \]

\[ \text{next-state} = \text{S0} \]

**S58**

\[ \text{EXP} = \text{pop(DStack)} \]

\[ \text{next-state} = \text{S0} \]
S0
dispatch tag(EXP)

PROC = pop(LabelStack)
UNEV = (sym, EXP)

PROC = UNEV

UNEV = pop(LabelStack)
VAL = PROC
PROC = XStack.SP

push(EXP, DStack)
push(S58, CStack)
push(UNEV, LabelStack)
push(VAL, LabelStack)
EXP = pop(XStack)
ENV = pop(XStack)
XStack.SP -= 5

if tag(EXP) ≠ p
    next-state = S0
else
    LabelStack.SP -= 4
    next-state = S0

co

PROC ≠ UNEV

PROC = UNEV

push(EXP, DStack)
push(S58, CStack)
push(UNEV, LabelStack)
push(VAL, LabelStack)
EXP = pop(XStack)
ENV = pop(XStack)
XStack.SP -= 5

if tag(EXP) ≠ p
    next-state = S0
else
    EXP = cdr(EXP)
    next-state = S0

next-state = EXP

S0
dispatch tag(EXP)

internal
FSM of the Control Unit

\[
\text{VAL} = \text{cons}((o, -6), \text{VAL}, op, 00101) \\
\text{VAL} = \text{cons}(\text{nil}, \text{VAL}, o) \\
\text{next-state} = \text{pop}(\text{CStack})
\]

\[
\text{VAL} = \text{cons}((o, -7), \text{VAL}, op, 00111) \\
\text{VAL} = \text{cons}(\text{nil}, \text{VAL}, o) \\
\text{next-state} = \text{pop}(\text{CStack})
\]

\[
\text{IR.V1} = \text{VAL} \\
\text{IR.Code} = '\text{strLen}' \\
\text{//Interrupt pending} \\
\text{ARGS} = \text{cons}((s, \text{'length'}, \text{IR.Result}, pr, 000) \\
\text{ARGS} = \text{cons}(\text{ARGS}, \text{nil}, p) \\
\text{VAL} = \text{cons}((o, -8), \text{VAL}, op, 01001) \\
\text{VAL} = \text{cons}(\text{ARGS}, \text{VAL}, o) \\
\text{next-state} = \text{pop}(\text{CStack})
\]

\[
\text{dispatch tag}(\text{VAL}) \\
\text{next-state} = \text{throwTypeError}
\]

\[
\text{IR.V1} = \text{VAL} \\
\text{IR.Code} = '\text{strLen}' \\
\text{//Interrupt pending} \\
\text{ARGS} = \text{cons}((s, \text{'length'}, \text{IR.Result}, pr, 000) \\
\text{ARGS} = \text{cons}(\text{ARGS}, \text{nil}, p) \\
\text{VAL} = \text{cons}((o, -8), \text{VAL}, op, 01001) \\
\text{VAL} = \text{cons}(\text{ARGS}, \text{VAL}, o) \\
\text{next-state} = \text{pop}(\text{CStack})
\]
VAL = toNumberComb
next-state = pop(CStack)

\[ u, nl, b, n \]

**toNumber**
dispatch tag(VAL)

\[ s \]

IR.V1 = VAL
IR.Code = 'strToNum'
// Interrupt pending
VAL = IR.Result
next-state = pop(CStack)

\[ o \]

UNEV = (s, 'valueOf')
push(toNumber, CStack)
next-state = toPrimitive

\[ s \]

toStringComb performs a combinatorial type conversion to string for pointer types \( u, nl, b \) and \( s \).

\[ o \]

UNEV = (s, 'toString')
push(toString, CStack)
next-state = toPrimitive

**toString**
dispatch tag(VAL)

\[ n \]
FSM of the Control Unit

**toPrimitive**

- tag(VAL) = o
- push(toPrimitive3, CStack)
- push(toPrimitive2, CStack)
- UNEV ≠ "valueOf"
- push((s, "valueOf"), DStack)
- push(VAL, DStack)
- next-state = getProperty
- UNEV = "valueOf"
- push((s, "toString"), DStack)
- push(VAL, DStack)
- next-state = getProperty

**toPrimitive2**

- tag(VAL) = pr
- VAL = cdr(VAL)
- tag(VAL) ≠ pr
- VAL = (o, 0)
- next-state = pop(CStack)
- tag(VAL) = o
- VAL = cdr(VAL)
- VAL = cdr(VAL, 5)
- tag(VAL) ≠ o
- VAL = (o, 0)
- next-state = pop(CStack)
- tag(VAL) = f
- push(nil, DStack)
- push(nil, DStack)
- push((n, 0), DStack)
- push(S22toPrim, CStack)
- next-state = S18
- tag(VAL) ≠ f
- VAL = (o, 0)
- next-state = pop(CStack)
toPrimitive3

\[
\text{tag}(\text{VAL}) = o \\
\text{VAL} = \text{pop} (\text{DStack}) \\
\text{UNEV} = \text{pop} (\text{DStack}) \\
\text{push} (\text{UNEV}, \text{DStack}) \\
\text{push} (\text{VAL}, \text{DStack}) \\
\text{push} (\text{toPrimitive4}, \text{CStack}) \\
\text{push} (\text{toPrimitive2}, \text{CStack}) \\
\text{next-state} = \text{getProperty}
\]

\[\text{tag}(\text{VAL}) \neq o \]

\[
\text{Stack.SP} -= 2 \\
\text{next-state} = \text{pop} (\text{CStack})
\]

toPrimitive4

\[
\text{tag}(\text{VAL}) = o \\
\text{tag}(\text{VAL}) \neq o
\]

\[
\text{next-state} = \text{throwTypeError}
\]

\[\text{Stack.SP} -= 2 \\
\text{next-state} = \text{pop} (\text{CStack})
\]

S22toPrim

\[
\text{push} (\text{ENV}, \text{CallStack}) \\
\text{push} (\text{THIS}, \text{CallStack}) \\
\text{push} (\text{LabelStack.SP}, \text{CallStack}) \\
\text{push} (\text{CStack.SP}, \text{CallStack}) \\
\text{push} (\text{DStack.SP}, \text{CallStack}) \\
\text{push} (\text{XStack.SP}, \text{CallStack}) \\
\text{ENV} = \text{car} (\text{PROC}) \\
\text{ENV} = \text{cons} (\text{ENV}, \text{nil}, \text{env}) \\
\text{THIS} = \text{pop} (\text{DStack}) \\
\text{push} (\text{THIS}, \text{DStack}) \\
\text{push} (\text{S24ret}, \text{CStack}) \\
\text{push} (\text{S0}, \text{CStack}) \\
\text{next-state} = \text{S23}
\]
FSM of the Control Unit

**setBinding**
ARGS = car(EXP)
PROC = car(ARG)

PROC = UNEV

//dummy

PROC ≠ UNEV

EXP = cdr(EXP)

ARGS & 1

|ARGS & 1|

next-state = throwTypeError

cdr(ARGS) = VAL
nenext-state = pop(CStack)

**setProperty**
ARGS = car(EXP)
PROC = car(ARG)

PROC = UNEV

PROC ≠ UNEV

cdr(ARGS) = VAL
car(EXP) = (ARGS & 0xFFFFFFFF8) | 6
next-state = pop(CStack)

EXP = cdr(EXP)
getValue

UNET = car(VAL)
VAL = cdr(VAL)
next-state = getValue2

getValue2

EXP = car(VAL)
next-state = get Property

getValue3

tag(VAL) ≠ ref

tag(VAL) = ref

UNEV = car(VAL)
VAL = cdr(VAL)
next-state = getValue2

getValue4

next-state = throwRefError

EXP = VAL
next-state = get Binding

getValue2

EXP = car(VAL)
next-state = get Property

getValue3

tag(VAL) ≠ u

tag(VAL) = u

next-state = pop(CStack)

getValue4

tag(VAL) ≠ u

tag(VAL) = u

next-state = throwRefError

next-state = pop(CStack)
**put**
push(VAL, DStack)
push(EXP, DStack)
EXP = car(EXP)
push(canPut, CStack)
next-state = getOwnProperty

canPut

```plaintext
tag(VAL) = u
VAL = pop(DStack)
push(VAL, DStack)
VAL = cdr(VAL)
VAL = car(VAL, 5)
push(canPut2, CStack)
next-state = getProperty

TAG = (VAL » 2) & 1
VAL = (b, TAG & 1)
next-state = throwTypeError
```

canPut2

```plaintext
PROC = pop(DStack)
ARGS = cdr(PROC)
tag(VAL) ≠ u
VAL = (b, ARGS & (VAL » 2) & 1)
// dummy

TAG = (VAL » 2) & 1
VAL = (b, TAG & 1)
next-state = throwTypeError
```
**getBinding**
ARGS = car(EXP)
PROC = car(ARGs)

PROC ≠ UNEV
EXP = cdr(EXP)

PROC = UNEV
VAL = cdr(ARGs)
next-state = pop(CStack)

**getOwnProperty**
ARGS = car(EXP)
PROC = car(ARGs)

PROC ≠ UNEV
EXP = cdr(EXP)
next-state = getOwnProperty
VAL = ARGS
next-state = pop(CStack)

**getProperty**
ARGS = cdr(VAL)
push(car(ARGs, 5), DStack)
push(getProperty2, CStack)
EXP = car(VAL)
next-state = getOwnProperty

**getProperty2**
TAG(VAL) ≠ u
Stack.SP -= 1
next-state = pop(CStack)

TAG(VAL) = u
VAL = pop(DStack)
next-state = getProperty

PROC ≠ UNEV
TAG(VAL) ≠ ⊥
TAG(EXP) ≠ ⊥
TAG(EXP) = ⊥
TAG(VAL) = ⊥
hasBinding

\[ \text{tag(}\text{EXP}\text{)} \neq \bot \]

\[ \text{PROC} = \text{car(}\text{EXP}\text{)} \]
\[ \text{PROC} = \text{car(}\text{PROC}\text{)} \]
\[ \text{EXP} = \text{cdr(}\text{EXP}\text{)} \]
\[ \text{next-state} = \text{hasBinding} \]

\[ \text{PROC} \neq \text{UNEV} \]

\[ \text{EXP} = \text{cdr(}\text{EXP}\text{)} \]
\[ \text{next-state} = \text{hasBinding} \]

\[ \text{PROC} = \text{UNEV} \]

\[ \text{VAL} = (b, 0) \]
\[ \text{next-state} = \text{pop(CStack)} \]

\[ \text{VAL} = (b, 0) \]
\[ \text{next-state} = \text{pop(CStack)} \]

\[ \text{throwTypeError} \]

\[ \text{VAL} = \text{cons((}\text{o, -10}\text{), nil, op, 01111)} \]
\[ \text{VAL} = \text{cons(nil, VAL; o)} \]
\[ \text{next-state} = \text{S6} \]

\[ \text{throwRefError} \]

\[ \text{VAL} = \text{cons((}\text{o, -11}\text{), nil, op, 01111)} \]
\[ \text{VAL} = \text{cons(nil, VAL; o)} \]
\[ \text{next-state} = \text{S6} \]
objectConst
UNEV = (s, "value")
EXP = cdr(ENV)
push(objectConst2, CStack)
next-state = getBinding

objectConst2

next-state = S3
push(S3, CStack)
next-state = toObject

toStringMethod

next-state = S3
push(toStringMethod2, CStack)
next-state = toObject

toStringMethod2

next-state = S3

printMethod

next-state = pop(CStack)

print

IR.V1 = VAL
IR.Code = "print"
//Interrupt pending
Bibliography


Curriculum Vitae

Personal Information

Name Alwin W. Zulehner
Address Weidfeldstraße 120, 4050 Traun
Telephone +43 650 2529000
Email alwin.zulehner@gmx.at
Date of Birth February 25 1990
Nationality Austrian

Professional Experience

10/2012–10/2014 Student Researcher, Institute for Computer Architecture, Linz
05/2009–09/2012 Tutor of mathematics, Schülerhilfe, Traun

Education

10/2009–09/2012 BSc in Computer Science, Johannes Kepler University, Linz, Austria.
09/2000–06/2008 Matura, Bundesrealgymnasium Traun, Austria.
Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Traun, am 8. Juni 2015

Alwin W. Zulehner