A sound type system for the meta language of theJavascript standard

(extended abstract of the MSc dissertation)

Pedro José Fernandes Nunes
Departamento de Matemática
Instituto Superior Técnico

Advisors: Professors Paulo Mateus and José Fragoso Santos

Abstract

JavaScript is the programming language most commonly used for client-side scripting in the world wide web and has been gaining popularity for other types of applications via Node.js. The complexity of the JavaScript semantics makes it a hard target for static analyses. Thus, a new intermediate untyped language named ECMA-SL was developed to assist with the analysis and specification of JavaScript programs. In this thesis, we introduce Typed ECMA-SL, a typed version of ECMA-SL, together with a flow-sensitive type system for the language. We further define a big-step and a small-step operational semantics for Typed ECMA-SL and prove the soundness of the proposed type system with respect to both semantics.

1 Introduction

JavaScript is the programming language most commonly used for client-side scripting in the world wide web [4] and has been gaining increasingly popularity for other types of applications via Node.js [6], a run-time environment for developing stand-alone JavaScript applications built on top of the V8 JavaScript engine [5]. In order to guarantee that JavaScript programs behave consistently throughout all existing browsers, the Ecma International association develops and maintains the JavaScript standard, a long complex document written in English, about one thousand pages long, describing both the syntax and semantics of the JavaScript language. The complexity of the JavaScript semantics makes it a hard target for static analyses, which, in order to be sound, have to reason about all the corner cases described in the official standard.

The standard way to deal with the complexity of real world programming languages when designing new program analyses is to first compile the given program to a simpler intermediate language and then apply the analysis at the intermediate language level. Following this approach, a research team at INESC-ID developed ECMA-SL, a new intermediate language for JavaScript analysis and specification. The ECMA-SL project [27] comes with a compiler from JavaScript to ECMA-SL, thereby allowing new static analyses for JavaScript to target ECMA-SL instead of JavaScript directly. ECMA-SL is a simple untyped imperative language with extensible objects and standard control flow constructs. In contrast to the semantics of JavaScript which is about 1000 pages long, the semantics of ECMA-SL can be formally described in one page, making it a suitable target for static analysis.

Currently the ECMA-SL project supports ECMAScript 5, the 5th version [2] of the JavaScript standard, which is now in its 12th version [1] (ECMAScript 12). The ECMA-SL project has at its core an ECMAScript 5 interpreter written in ECMA-SL called ECMARef5 [23], which consists of more than 10K lines of ECMA-SL code. In order to adapt the ECMA-SL project to the more recent versions of the ECMAScript standard, one has to adapt and extend the ECMARef5 interpreter. This is by no means an easy task as the size and complexity of the standard grew substantially since its 5th version. Furthermore, the fact that ECMA-SL is untyped makes any sort of refactoring of the existing code base extremely error prone and time consuming. For this reason, the aim of this project is to streamline the management and maintenance of the ECMARef interpreter by adding a type system to ECMA-SL.

A type system is a syntactic method for checking the absence of certain classes of errors in programs by classifying the given program's statements and expressions according to the kinds of values that it computes. Typically, a type system is defined as a set of rules, with each rule applying to a specific phrase of the language. These rules target type errors such as an operand or argument passed to a function being incompatible with the type expected by that operator or function. Ideally, type systems are supposed to be *sound*: if a sound type system accepts a program, then there are no inputs for which the execution of that program throws a runtime error. In order to guarantee soundness, type systems have to be conservative, meaning that they have to reject not only incorrect programs but also correct programs that cannot be proven so. We say that a type system is more precise than another if the former rejects fewer correct programs than the latter. In practice, there is a trade-off between precision and complexity of type annotations. The more precise a type system is, the more complex and unwieldy are its corresponding type annotations. Hence, in general, more precise type systems are more difficult to use. In order to avoid this trade-off, some type systems are purposely designed to be unsound with the goal of rejecting as few correct programs as possible whilst not having an overly complex notational burden.

Much like JavaScript, ECMA-SL is a highly dynamic programming language, including features such as extensible objects and dynamic binding of function calls, which make it a hard target for standard type systems. In particular, the combination of aliasing with object mutability is difficult to control if one wants to keep the type system both sound and precise. The goal of this thesis is to formalise a typed version of ECMA-SL, named Typed ECMA-SL, together with a type system with the two following characteristics:

- Soundness: well-typed programs cannot go wrong;
- Flow-Sensitivity: program variables and objects are allowed to change their types during execution.

Flow-sensitivity is key for precision, allowing us to keep our type system as little restrictive as possible. The key idea of our type system is to explicitly track object aliasing and constrain object mutation. In particular, objects are only allowed to be mutated if there is a single pointer to them.

With the gaining popularity of JavaScript, many industrial and academic research groups have developed type systems for different fragments of the language [8, 10, 29]. As ECMA-SL and JavaScript have many common features, one would expect that one of the proposed type systems for JavaScript could be applied to ECMA-SL. This is, however, not the case as none of these systems meets our requirements; some of them are explicitly unsound [8], some are flow-insensitive [29], and others are overly complex due to features of JavaScript that are not included in ECMA-SL [10] such as prototype inheritance.

We consider this thesis to have three contributions: first, the formalisation of the Typed ECMA-SL language; second, the development of a type system based on a novel idea - open/closed types for tracking aliasing; finally, two soundness proofs written with respect to two different operational semantics. Below, we briefly describe each of these contributions.

Typed ECMA-SL The first contribution of this work is the definition of Typed ECMA-SL, a typed version of ECMA-SL. In order to facilitate the transition for developers, Typed ECMA-SL was designed to be as similar to ECMA-SL as possible, thus minimizing the number of extra annotations required.

Type System The developed type system is the central contribution of this thesis as it not only provides a set of rules for supporting the development of correct programs, but also presents a novel idea which can be adapted to other object-oriented scripting languages: open/closed objects. This idea, which is at the core of our type system, is key for allowing it to be flow-sensitive, while keeping it sound.

Soundness Proofs We provide two soundness proofs for our type system. In order to do this we introduce two different semantics: a big-step semantics [22] and a small-step semantics [26]. By proving the soundness of our type system we ensure that well-typed programs cannot go wrong, which was one of our type system's requirements. We chose to provide two different proofs to explore different trade-offs between clarity and expressivity.

2 Typed ECMA-SL

ECMA-SL is a simple imperative language with extensible objects developped to assist with JavaScript analysis and specification. The ECMA-SL language was used to develop ECMARef, a new JavaScript refernece interpreter that follows the ECMAScript standard [2], the official JavaScript standard, faithfully. The ECMA-SL project has been developed by a research team at INESC-ID and has been throughly tested against Test262 [3], the official ECMAScript conformance suite.

So far, ECMA-SL is an untyped language and, therefore, it has two major disadvantages when compared to typed languages. Firstly, a typed language allows for the static detection of code errors. Secondly, programs written in untyped languages are harder to maintain than typed languages, which promote a design-by-contract approach to the software development process, with function signatures acting as a clear interface between the code of the corresponding functions and the programs that use them. This project contributes to the overall ECMA-SL project by designing a new typed version of ECMA-SL, which mitigates these two defects.

In this chapter, we introduce our typed version of ECMA-SL, called Typed ECMA-SL, together with its type system.

2.1 Syntax

A Typed ECMA-SL program $p \in Progs$ is a collection of Typed ECMA-SL functions. A Typed ECMA-SL function $func \in Funcs$ is of the form $function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$, where f is the identifier, $x_1, ..., x_n$ are the function formal parameters with types $\tau_1, ..., \tau_n$, and s is the body of the function. The syntax of Typed ECMA-SL is given in Figure 1, mostly coinciding with that of untyped ECMA-SL.

Typed vs Untyped ECMA-SL The main differences between Typed ECMA-SL and ECMA-SL are the following:

1. Function parameters must be annotated with their corresponding types.
2. The syntax is extended with a special statement `commit` to provide information to the type system.
3. We restrict field look-ups, deletions, and assignments to require the name of the field to appear statically. For instance, we do not support the syntax o[x], where x is a program variable that denotes the name of the field being inspected; instead, we only have the syntax o.p, where p is exactly the name of the field being inspected.
4. Functions calls are fully static; that is, the identifier of the function to be called must be known at static time. In contrast, in untyped ECMA-SL, that identifier can be computed dynamically.

Types Typed ECMA-SL includes three main categories of types: primitive types, function types, and object types. Primitive types comprise the string type, the number type, the

Expressions	Statements	Types
$e \in \mathcal{E} ::= x$	$s \in \mathcal{S} ::= x := e$	$\tau \in T ::= number$
$\mid v$	$\mid x.f := e$	$\mid str$
$\mid \oplus(e)$	$\mid x := g.f$	$\mid null$
$\mid \otimes(e_1, e_2)$	$\mid x := \{\}$	$\mid undefined$
	$\mid \mathsf{commit}(x)$	$\mid \{p_i \tau_i \mid_{i=1}^{n}\}^{\circ}$
$\otimes \in \{+, x, -, ...\}$	$\mid \mathsf{delete}(x.f)$	$\mid \{p_i \tau_i \mid_{i=1}^{n}\}^{\bullet}$
	$\mid \mathsf{skip}$	$\mid (\tau_1, ..., \tau_n) \rightarrow \tau$
$v \in \mathcal{V} ::= true$	$\mid s_1; s_2$	
$\mid false$	$\mid \mathsf{if}(e)\{s_1\}$ else $\{s_2\}$	
$\mid n \in \mathbb{N}$	$\mid \mathsf{while}(e)\{s\}$	
$\mid string$	$\mid x := f(e_1, ..., e_n)$	
	$\mid \mathsf{return}(e)$	

Figure 1. Typed ECMA-SL Syntax

boolean type, and the special undefined and null types. Function types have their standard interpretation:

The type $(\tau_1, ..., \tau_n) \rightarrow \tau$ is the type of the functions that take arguments of types $\tau_1, ..., \tau_n$ and produce a result of type τ. Object types are more complicated. The object type $\{p_i \tau_i \mid_{i=1}^{n}\}^{*}$ denotes objects that only contain the fields p_1 to p_n, mapping each field p_i to a value of type τ_i. Given an object type $\tau = \{p_i \tau_i \mid_{i=1}^{n}\}^{*}$, we write $dom(\tau)$ to mean the set of fields that it contains: $\{p_i \mid_{i=1}^{n}\}$ and $\lfloor \tau \rfloor$ to refer to its openness flag, $*$.

We have two classes of object types: open object types, $\{p_i \tau_i \mid_{i=1}^{n}\}^{\circ}$, and closed object types, $\{p_i \tau_i \mid_{i=1}^{n}\}^{\bullet}$. If an object has an open object type, it is referred to as an *open object*, and, if not, a *closed object*. Only open objects can be extended or shrank during execution, meaning that we can only add new fields or delete existing fields to/from open objects. The domain of a closed object is not allowed to change during execution and the types of its fields must remain the same. When an object is created, it is assumed to be open. After populating an object with all the fields that it should contain, the programmer must close it using the commit statement. Closing an object is essential if one intends to assign it to other variables, as our type system enforces that only closed objects can be referenced by more than one pointer.

2.2 Type System

Definition 1 (Store Typing Environment)**.** *A store typing environment is a function* $\Gamma : Var \mapsto \mathrm{T}$ *mapping variables in* Var *to types in* T.

Definition 2 (Global Typing Context)**.** *A global typing context is a partial function* $\Delta : F \rightharpoonup \mathrm{T}$ *mapping function identifiers in the set of all function identifiers* F *to function types in* T.

Typing Rules for Expressions Given a store typing environment Γ, an expression e and a type τ it is said that Γ types the expression e with type τ, written $\Gamma \vdash e : \tau$ as long as there is a derivation for it according to the rules defined in Figure 2.

$$\textsc{Variable}\ \frac{\tau = \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \textsc{Value}\ \frac{Type(v) = \tau}{\Gamma \vdash v : \tau} \qquad \textsc{Unary Operation}\ \frac{\Gamma \vdash e : \tau_e \qquad \oplus(\tau_e) = \tau}{\Gamma \vdash \oplus(e) : \tau}$$

$$\textsc{Binary Operation}\ \frac{\Gamma \vdash e_1 : \tau_{e_1} \qquad \Gamma \vdash e_2 : \tau_{e_2} \qquad \otimes(\tau_{e_1}, \tau_{e_2}) = \tau}{\Gamma \vdash \otimes(e_1, e_2) : \tau}$$

Figure 2. Typing Rules for Expressions: $\Gamma \vdash e : \tau$

Typing Rules for Statements Given a function identifier g, a global typing context Δ, two store typing environment Γ_1 and Γ_2, and a statement s, the typing judgement $g, \Delta \vdash \{\Gamma_1\}\ s\ \{\Gamma_2\}$ means that s occurs within the body of g and that under the global typing context Δ, the execution of s on a variable store satisfying the initial store typing environment Γ_1 results in a variable store satisfying the final variable typing environment Γ_2.

The key insight of our type system is that we have to control aliasing. In particular, our type system enforces a no aliasing policy for open objects, which guarantees that open objects can only be accessed through a single program variable at a time. To ensure this, we do not allow open objects to be assigned to program variables and/or object fields. Once an object is closed, such assignments are allowed. We call this policy *no aliasing for open objects (NAOO)* and will discuss it thoroughly in the subsequent chapters of this thesis. In the rules, we make use of a predicate $\mathrm{Closed}(\tau)$ to determine whether or not the given type τ is closed. Primitive types are treated as closed object types, meaning that $\mathrm{Closed}(\tau)$ also holds when τ is a primitive type.

In Figure 3 we provide some of our typing rules.

Typing functions A function $function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$ is said to be *typable* under the global typing context Δ, written $\Delta \vdash function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}$, if its body is typable with respect to the typing environment obtained by

FIELD ASSIGNMENT - OPEN EXIST

$$\frac{\Gamma \vdash e : \tau_e \qquad \Gamma(x) = \{f_i : \tau_i|_{i=1}^k\}^\circ \qquad \exists_{j\in\{1,...,k\}} f = f_j \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x.f := e\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1,i\neq j}^k, f : \tau_e\}^\circ]\}}$$

FIELD ASSIGNMENT - OPEN NON EXIST

$$\frac{\Gamma \vdash e : \tau_e \qquad \Gamma(x) = \{f_i : \tau_i|_{i=1}^k\}^\circ \qquad \forall_{i\in\{1,...,k\}} f \neq f_i \qquad \text{Closed}(\tau_e)}{g, \Delta \vdash \{\Gamma\}\ x.f := e\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^k, f : \tau_e\}^\circ]\}}$$

COMMIT

$$\frac{\Gamma(x) = \{f_i : \tau_i|_{i=1}^k\}^\circ}{g, \Delta \vdash \{\Gamma\}\ \text{commit}(x)\ \{\Gamma[x \mapsto \{f_i : \tau_i|_{i=1}^k\}^\bullet]\}}$$

FUNCTION CALL

$$\frac{\Gamma \vdash e_i : \tau_i|_{i=1}^n \qquad \Delta(f) = (\tau_1, ..., \tau_n) \mapsto \tau \qquad \text{Closed}(\tau_i)|_{i=1}^n}{g, \Delta \vdash \{\Gamma\}\ x := f(e|_{i=1}^n)\ \{\Gamma[x \mapsto \tau]\}}$$

RETURN

$$\frac{\Delta(g) = (\tau_1, ..., \tau_n) \mapsto \tau \qquad \Gamma \vdash e : \tau}{g, \Delta \vdash \{\Gamma\}\ \text{return}(e)\ \{\Gamma\}}$$

Figure 3. Typing Rules for Statements: $f, \Delta \vdash \{\Gamma_1\}\ s\ \{\Gamma_2\}$

mapping its formal parameters to their respective types. This concept is formally defined below.

$$\frac{\Gamma = [x_1 \mapsto \tau_1, ..., x_n \mapsto \tau_n] \qquad \Delta(f) = (\tau_1, ..., \tau_n) \to \tau \qquad f, \Delta \vdash \{\Gamma\} s \{\Gamma'\}}{\Delta \vdash function\ f(x_1 : \tau_1, ..., x_n : \tau_n)\{s\}}$$

A program p is said to be *typable* under a global typing context Δ, written $\Delta \vdash p$, if all functions in the range of the program are typable with respect to Δ. The notation $\Delta_r(f)$ is used to refer to the return type of f.

3 Big-Step Soundness

In this section we prove the soundness of our type system with respect to a big-step semantics of ECMA-SL. In order to simplify the exposition, the semantics of ECMA-SL that we first present does not model wrong executions and function calls. We later extend it to cater for these two aspects.

3.1 ECMA-SL State Properties

ECMA-SL States An ECMA-SL state is composed of a heap $h : Loc \times Str \rightharpoonup \mathcal{V}$, mapping pairs of locations and string to values, and a store $\rho : Var \rightharpoonup \mathcal{V}$, mapping program variables to values.

Following well-established approaches for modelling the semantics of JavaScript [17, 28], instead of modelling a heap as a function from locations to objects, objects are not explicitly represented in the formalism. At the semantic level, an object can be seen as a *region of the heap*. More concretely, the object pointed to by location l corresponds to the set of cells whose first element is l. In the following, we write $h(l)$ to mean $\{(l, f) \mid (l, f) \in dom(h)\}$ and $dom(h(l))$ to mean $\{f \mid (l, f) \in dom(h)\}$.

3.1.1 State Satisfiability

Here we define what it means for an ECMA-SL state to satisfy a given typing environment. To this end, we first extend the notion of typing environment to heaps, introducing the concept of *heap typing environment* and then give the formal definition of state satisfiability.

Heap Typing Environment In order to define state satisfiability we first introduce the concept of *heap typing environment* which maps each heap location to the type of the object it refers to.

Definition 3 (Heap Typing Environment)**.** *A heap typing environment is a partial function* $\Sigma : Loc \rightharpoonup \mathrm{T}$ *that maps locations from the set of heap locations* Loc *to the set of types* T.

In the following, we use $\Sigma(l, f)$ to refer to the type of the field f in the object pointed to by location l. Put formally: $\Sigma(l, f) = \tau_f \iff \exists\, \tau.\ \Sigma(l) = \tau \ \wedge\ \tau = \{..., f : \tau_f, ...\}^*$.

State Satisfiability We define the state satisfiability relation with the help of three auxiliary satisfiability relations:

- *Value Satisfiability:* describing what it means for a value v to satisfy a given type τ under a heap typing environment Σ – written $v \vDash_\Sigma \tau$;
- *Store Satisfiability:* describing what it means for a store ρ to satisfy a given store typing environment Γ under a heap typing environment Σ – written $\rho \vDash_\Sigma \Gamma$;
- *Heap Satisfiability:* describing what it means for a heap to satisfy a heap typing enviroment – written $h \vDash \Sigma$.

Definition 4 (Value Satisfiability)**.** *A value v is said to satisfy a type τ with respect to a heap typing environment Σ, written $v \vDash_\Sigma \tau$, if either:*

- *v is a number and τ is the number type;*
- *v is a string and τ is the string type;*
- *v is $true$ or $false$ and τ is the bool type;*
- *if v is a location l and $\Sigma(l) = \tau$*

Definition 5 (Store Satisfiability)**.** *Given a heap typing environment Σ, a store ρ is said to satisfy a store typing environment Γ, written $\rho \vDash_\Sigma \Gamma$), if and only if:*

$$dom(\rho) = dom(\Gamma) \ \wedge\ \forall_{x\in dom(\rho)} \rho(x) \vDash_\Sigma \Gamma(x)$$

Definition 6 (Heap Satisfiability)**.** *A heap h is said to satisfy a heap typing environment Σ, written $h \vDash \Sigma$, if and only if:*

- $dom(h) = dom(\Sigma)$
- $\forall_{l\in dom(h)} dom(h(l)) = dom(\Sigma(l))$
- $\forall_{l\in dom(h)} \forall_{f\in dom(h(l))} h(l, f) \vDash_\Sigma \Sigma(l, f)$

To avoid clutter, we use the notation $h, \rho \vDash \Sigma, \Gamma$ to mean that $h \vDash \Sigma$ *and* $\rho \vDash_\Sigma \Gamma$.

VARIABLE $[\![x]\!]_\rho \triangleq \rho(x)$ VALUE $[\![v]\!]_\rho \triangleq v$ UNARY OPERATION $[\![\oplus(e)]\!]_\rho \triangleq \underline{\oplus}([\![e]\!]_\rho)$

BINARY OPERATION $[\![\otimes(e_1, e_2)]\!]_\rho \triangleq \underline{\otimes}([\![e_1]\!]_\rho, [\![e_2]\!]_\rho)$

Figure 4. Big-step semantics for expressions $[\![e]\!]_\rho \triangleq v$

3.1.2 No-aliasing Invariant

As stated before, in order to deal with aliasing and mutation, our type system enforces a simple invariant: *only closed objects can be referenced by more than one pointer*. We formalise this invariant as the state property given in the definition below.

Definition 7 (No Aliasing for Open Objects)**.** *A heap h, a store ρ and a heap typing environment Σ are said to satisfy the no aliasing for open objects (NAOO) property, written $NAOO(h, \rho, \Sigma)$, if and only if:*

- $\forall_{l \in dom(\Sigma)} \ \lfloor \Sigma(l) \rfloor = \circ \Rightarrow \neg \exists_{(l', f)} : h(l', f) = l$ $NAOO_1$
- $\forall_{l \in dom(\Sigma)} \ \lfloor \Sigma(l) \rfloor = \circ \Rightarrow \neg \exists_{x_1, x_2} \ x_1 \neq x_2 \wedge \rho(x_1) = \rho(x_2) = l$ $NAOO_2$

Essentially the no aliasing for open objects (NAOO) property states that an open object can only be referenced by a single program variable; this means that: **(1)** it cannot be referenced by an object field ($NAOO_1$) and **(2)** it cannot be referenced by two distinct program variables x_1 and x_2 ($NAOO_2$).

Essentially, our type system enforces that objects can only be mutated if they are open, meaning that there is a single reference pointing to them. This guarantees that object mutation does not cause the type of a given reference (variable or object field) to become inconsistent with the type of its corresponding value.

3.2 Big-Step Semantics

Our semantics for statements makes use of a simple big-step semantics for ECMA-SL expressions given in the figure below, where we use the notation $[\![e]\!]_\rho$ to mean the evaluation of the expression e in the store ρ. Note that, given that ECMA-SL expressions do not interact with the object heap, the semantics of expressions only depends on the variable store.

We are now at the position to define our semantic judgement for statements. The semantic judgement have the form $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$, meaning that the evaluation of the statement s in the heap h and store ρ results in the heap h' and store ρ'. In order to reason about the types of the objects in the heap, we have to instrument the semantics to keep track of the types of the objects created at runtime. To this end, the semantic judgement for statements additionally include the initial and final heap typing environments, respectively Σ and Σ'. Some examples of the semantic rules are given in Figure 5.

FIELD ASSIGNMENT - OPEN

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \tau = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \circ \qquad \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]] \qquad \text{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e \rangle \Downarrow_i \langle \Sigma', h[(l, f) \mapsto v], \rho \rangle}$$

FIELD ASSIGNMENT - CLOSE

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \Sigma(l, f) = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \bullet \qquad (l, f) \in dom(h) \qquad \text{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e \rangle \Downarrow_i \langle \Sigma, h[(l, f) \mapsto v], \rho \rangle}$$

COMMIT

$$\frac{[\![x]\!]_\rho = l \qquad \lfloor \Sigma(l) \rfloor = \circ}{\langle \Sigma, h, \rho, \text{commit}(x) \rangle \Downarrow_i \langle \Sigma[l \mapsto \Sigma(l)^\bullet], h, \rho \rangle}$$

Figure 5. Big-step semantics for statements: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$

The proposed semantics enforces the *no aliasing for open objects* (NAOO) invariant. To this end, before every assignment, the semantics checks if the value being assigned is either a primitive value or a closed object; only in such cases is the assignment allowed to go through. To avoid clutter, we introduce the predicate $\text{Closed}_\Sigma(v)$ to mean that v is either a primitive value or a closed object.

Definition 8 (Closed Values)**.** *Let Σ be a heap typing environment, a value v is said to be of a closed with respect to Σ, written $\text{Closed}_\Sigma(v)$, if and only if it is of a primitive type or if $\lfloor \Sigma(v) \rfloor = \bullet$.*

3.3 Well-Typed Expressions - Safety

Our type system for statements relies on a simple type system for expressions. Unsurprisingly, in order to establish the safety of the type system for statements, we first have to establish the safety of our type system for expressions.

Lemma 1 (Well-typed Expressions - Safety)**.** *Let e be an expression, τ_e a type, ρ a store, Σ a heap typing environment and Γ store typing environment. Suppose that $\Gamma \vdash e : \tau_e$, $\rho \vDash_\Sigma \Gamma$ and $[\![e]\!]_\rho = v$. Then $v \vDash_\Sigma \tau_e$.*

3.4 Soundness - Type Safety

Theorem 1 states that the proposed type system satisfies the Type Safety property.

Theorem 1 (Soundness - Type Safety)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle$ and $h, \rho \vDash \Sigma, \Gamma$. Then $h', \rho' \vDash \Sigma', \Gamma'$.*

3.5 Soundness - Fault Avoidance

In this subsection, we extend our operational semantics to take into account erroneous executions and prove the standard fault avoidance property of sound type systems: *well-typed programs cannot go wrong*. To this end, we extend the operational semantics defined in Subsection 3.2 with explicit error

OPEN RHS - LOOKUP

$$\frac{[\![e]\!]_\rho = l \qquad h(l,f) = v \qquad \neg \mathsf{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x := e.f \rangle \Downarrow_i \lightning}$$

CLOSED OBJECT - ILLEGAL DELETION

$$\frac{[\![x]\!]_\rho = l \qquad \lfloor Sigma(l) \rfloor = \bullet}{\langle \Sigma, h, \rho, \mathsf{delete}(x.f) \rangle \Downarrow_i \lightning}$$

IF - ILLEGAL GUARD

$$\frac{[\![e]\!]_\rho = v \qquad Type_\Sigma(v) \neq bool}{\langle \Sigma, h, \rho, \mathsf{if}(e)\{s_1\} \ \mathsf{else} \ \{s_2\} \rangle \Downarrow_i \lightning}$$

Figure 6. Big-Step semantics for statements - erroneous executions: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \lightning$

derivations, writing: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \lightning$ to mean that evaluation of the statement s in the heap h, store ρ, and heap typing environment Σ leads to an execution error. One can leverage these error derivations to formally define fault avoidance. Essentially, if a statement s is typable with respect to a given function g, typing context Δ and store typing environments Γ and Γ', i.e. $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, and if one executes s in a state (h, ρ) such that $h, \rho \vDash \Sigma, \Gamma$, for a given heap typing environment Σ, then the execution of s will not result in a runtime error; put formally:

$$(g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\} \ \wedge \ h, \rho \vDash \Sigma, \Gamma) \implies \langle \Sigma, h, \rho, s \rangle \not\Downarrow_i \lightning \quad (1)$$

3.5.1 Error Executions

Figure 6 provides an overview of the extended operational semantics given in Subsection 3.2 with a set of explicit error derivations. These derivations model the runtime errors that can occur during the execution of an ECMA-SL statement. We consider the following five types of runtime errors: **(1)** branching on a value that is not of boolean type; **(2)** creating a second reference to an open object, thereby violating the NAOO invariant; **(3)** updating a field of a closed object to a value of a different type; **(4)** adding a new field to a closed object; and **(5)** deleting a field from a closed object.

3.5.2 Soundness - Fault Avoidance

Theorem 2 states that the proposed type system satisfies the Fault Avoidance property.

Theorem 2 (Soundness - Fault Avoidance)**.** *Let g be a function and Δ a Typing Context. Let h be a heap, ρ a store and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$ and $h, \rho \vDash \Sigma, \Gamma$ then it is not the case that $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \lightning$.*

3.6 Function and Return

In this subsection, we extend the operational semantics introduced in Subsection 3.2 with support for function calls and return statements. To this end, we have to change the format of the semantic judgment so that it additionally produces an outcome, which captures the flow of execution. Modified semantic judgements have the form: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$

SEQUENCING-3

$$\frac{\langle \Sigma, h, \rho, s_1 \rangle \Downarrow_i \langle \Sigma_1, h_1, \rho_1, Cont \rangle \qquad \langle \Sigma_1, h_1, \rho_1, s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, o \rangle}{\langle \Sigma, h, \rho, s_1; s_2 \rangle \Downarrow_i \langle \Sigma_2, h_2, \rho_2, o \rangle}$$

RETURN

$$\frac{[\![e]\!]_\rho = v}{\langle \Sigma, h, \rho, \mathsf{return}(e) \rangle \Downarrow_i \langle \Sigma, h, \rho, Ret(v) \rangle}$$

FUNCTION CALL

$$\frac{[\![e_i]\!]_\rho = v_i|_{i=1}^n \qquad \mathsf{body}(f) = s \qquad \mathsf{params}(f) = x_i|_{i=1}^n \qquad \langle \Sigma, h, [x_i \mapsto v_i|_{i=1}^n], s \rangle \Downarrow_i \langle \Sigma', h', \rho', Ret(v) \rangle \qquad \mathsf{Closed}_\Sigma(v_i)_{i=1}^n}{\langle \Sigma, h, \rho, x := f(e_1, ..., e_n) \rangle \Downarrow_i \langle \Sigma', h', \rho[x \mapsto v], Cont \rangle}$$

Figure 7. Big-Step semantics for statements - function call: $\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho', o \rangle$

signifying that the evaluation of the statement s in the heap h, store ρ, and heap typing environment Σ results in the heap h', store ρ', heap typing environment Σ', and outcome o. Outcomes are given by the following grammar:

$$o ::= Cont \mid Err \mid Ret(v) \quad (2)$$

We consider three types of outcomes: **(1)** the continuation outcome $Cont$, signifying that the execution may proceed with the next statement; **(2)** the error outcome Err, signifying that the execution generated an error and must therefore be terminated; and **(3)** the return outcome $Ret(v)$, signifying that the code of the function that is currently executing returned the value v.

Figure 7 gives a selection of the extended semantic rules.

Soundness of the Complete Type System The soundness of the complete can now be established, meaning that the type system satisfies the Type Safety and Fault Avoidance properties with respect to the extended operational semantics.

4 Small-Step Soundness

In this sectionwe prove the soundness of our type system with respect to a small-step semantics of ECMA-SL. In order to simplify the exposition, and as in the previous section, the semantics of ECMA-SL that we first present does not model function calls.

4.1 Small-Step Semantics

In this subsection we define a small-step semantics for ECMA-SL statements, ignoring for now function calls. Once again, we rely on the semantics for expressions presented in Figure 4, using the notation $[\![e]\!]_\rho = v$ to mean that the evaluation of the expression e in the store ρ results in the value v.

The small-step semantics judgements for statements are of the form $\langle \Sigma, h, \rho, s \rangle \rightarrow_i \langle \Sigma', h', \rho', s' \rangle$, meaning that the evaluation of the statement s in the heap h and store ρ leads to a statement s' in the heap h' and store ρ'. Similarly to our

FIELD ASSIGNMENT OPEN

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \tau = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \circ \qquad \Sigma' = \Sigma[l \mapsto \Sigma(l)[f \mapsto \tau]] \qquad \text{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e \rangle \rightarrow_i \langle \Sigma', h[(l,f) \mapsto v], \rho, \text{skip} \rangle}$$

FIELD ASSIGNMENT CLOSE

$$\frac{[\![e]\!]_\rho = v \qquad [\![x]\!]_\rho = l \qquad \Sigma(l,f) = Type_\Sigma(v) \qquad \lfloor \Sigma(l) \rfloor = \bullet \qquad (l,f) \in dom(h) \qquad \text{Closed}_\Sigma(v)}{\langle \Sigma, h, \rho, x.f := e \rangle \rightarrow_i \langle \Sigma, h[(l,f) \mapsto v], \rho, \text{skip} \rangle}$$

SEQUENCING COMPOSITION

$$\frac{\langle \Sigma, h, \rho, s_1 \rangle \rightarrow_i \langle \Sigma_2, h_2, \rho_2, s_1' \rangle}{\langle \Sigma, h, \rho, s_1; s_2 \rangle \rightarrow_i \langle \Sigma_2, h_2, \rho_2, s_1'; s_2 \rangle}$$

Figure 8. Small-Step semantics for statements: $\langle \Sigma, h, \rho, s \rangle \rightarrow_i \langle \Sigma', h', \rho', s \rangle$

big-step semantics, and in order to reason about the types of the objects in the heap, we have to instrument the semantics to keep track of the types of the objects created at runtime. To this end, the semantic judgement for statements additionally include the initial and final heap typing environments, respectively Σ and Σ'.

Our small-step semantics is equivalent to the big-step semantics provided in the previous chapter. Put formally, for every heap typing environments Σ and Σ', heaps h and h', stores ρ and ρ', and statement s, it holds that:

$$\langle \Sigma, h, \rho, s \rangle \Downarrow_i \langle \Sigma', h', \rho' \rangle \iff \langle \Sigma, h, \rho, s \rangle \rightarrow_i^* \langle \Sigma', h', \rho', \text{skip} \rangle \tag{3}$$

where: we use $\rightarrow_i^*$ to denote the reflexive-transitive closure of $\rightarrow_i$. From the equivalence result, it follows that the small-step semantics also enforces the *no aliasing for open objects* (NAOO) invariant. The semantic rules are given in Figure 8

Unsurprisingly, the small-step rules are analogous to the big-step rules given in the previous chapter. In fact, the set of rules that lead to a skip statement almost exactly coincide with the corresponding big-step rules. The major differences appear in the rules associated with compound statements: if, while, and sequence. For these statements, the corresponding small-step rules perform a single computation step, generating the statement to be executed next, while the big-step rules capture their complete evaluation.

4.2 Soundness - Preservation

Theorem 3 states that the proposed type system satisfies the Preservation property.

Theorem 3 (Preservation)**.** *Let g be a function and Δ a Typing Context. Let Σ and Σ' heap typing environments, Γ, Γ' and Γ_f store typing environments, h and h' heaps, ρ and ρ' stores and s and s' statements. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$, $\langle \Sigma, h, \rho, s \rangle \rightarrow_i \langle \Sigma', h', \rho', s' \rangle$ and $h, \rho \vDash \Sigma, \Gamma$. Then $\exists \Gamma' : h', \rho' \vDash \Sigma', \Gamma' \ \wedge\ g, \Delta \vdash \{\Gamma'\}\ s'\ \{\Gamma_f\}$*

4.3 Soundness - Progress

In this subsection we prove that the proposed type system satisfies the *progress property* with respect to the small-step operational semantics defined in Subsection 4.1.

In order to establish the progress property of our type system, we make use of a new auxiliary property named progress of expression typing.

Lemma 2 (Well-typed Expressions - Progress)**.** *Let e be an expression, τ_e a type, ρ a store, Σ a heap typing environment and Γ store typing environments. Suppose that $\Gamma \vdash e : \tau_e$ and $\rho \vDash_\Sigma \Gamma$. Then $\exists_v : [\![e]\!]_\rho = v$.*

Theorem 4 states that the proposed type system satisfies the Progress property.

Theorem 4 (Progress)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store, and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$ and $h, \rho \vDash \Sigma, \Gamma$. Then either $s = \text{skip}$ or $\exists \Sigma', h', \rho', s' : \langle \Sigma, h, \rho, s \rangle \rightarrow_i \langle \Sigma', h', \rho', s' \rangle$.*

4.4 Function and Return

In this subsection, we adapt our small-step semantics and soundness proofs to account for function calls and return statements.

4.4.1 Semantics

In order to extend our small-step semantics to take into account function calls, we rely on the notion of *call stack* [16]. Call stacks are used to keep track of the execution context of the calling function. Hence, when evaluating a function call, we extend the current call stack with a record that book-keeps the calling context. Conversely, when evaluating a return statement, the semantics discards the current execution context and recovers the execution context of the calling function (i.e. the function that receives the returned value) from the call stack. Formally, call stacks are generated by the following grammar:

$$cs ::= [\,] \mid (f, x, \rho, s) :: cs \tag{4}$$

Where :: denotes list concatenation and $[\,]$ the empty list. Essentially, a call stack is a list of 4-tuples of the form (f, x, ρ, s), referred to as call stack records, where: **(1)** f is the identifier of the calling function; **(2)** x is the program variable of the calling function to which the result of the current function is to be assigned; **(3)** ρ is the store of the calling function; and **(4)** s is the continuation of the calling function; that is: the part of the body of the calling function that still remains to be executed once the current function returns.

Given a call stack cs, we use the notation $stores(cs)$ to refer to the corresponding list of stores; the function $stores$ is inductively defined as follows:

$$stores(cs) = \begin{cases} [\,] & \text{if } cs = [\,] \\ \rho :: stores(cs') & \text{if } cs = (-, -, \rho, -) :: cs' \end{cases} \tag{5}$$

We are now at the position to extend our semantic judgement for statements introduced in Subsection 4.1 with support for function calls and return statements. To this end,

FUNCTION CALL

$$\frac{[\![e_i]\!]_\rho = v_i|_{i=1}^n \qquad \mathsf{Closed}_\Sigma(v_i)|_{i=1}^n \qquad \mathsf{body}(f) = s' \qquad \mathsf{params}(f) = x_i|_{i=1}^n \qquad cs' = (g, x, \rho, s) :: cs}{\langle g, \Sigma, h, \rho, cs, x := f(e_1, ..., e_n); s\rangle \rightarrow_i \langle f, \Sigma, h, [x_i \mapsto v_i|_{i=1}^n], cs', s'\rangle}$$

RETURN

$$\frac{[\![e]\!]_\rho = v \qquad cs = (f, x, \rho', s') :: cs'}{\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e); s\rangle \rightarrow_i \langle f, \Sigma, h, \rho'[x \mapsto v], cs', s'\rangle}$$

TOP LEVEL RETURN

$$\frac{[\![e]\!]_\rho = v \qquad cs = []}{\langle g, \Sigma, h, \rho, cs, \mathsf{return}(e); s\rangle \rightarrow_i \langle g, \Sigma, h, \rho[out \mapsto v], cs, \mathsf{skip}\rangle}$$

Figure 9. Small-Step semantics for statements - function call: $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle g', \Sigma', h', \rho', cs', s'\rangle$

we have to change the format of the semantic judgment to $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle g', \Sigma', h', \rho', cs', s'\rangle$, with g and cs representing the current function identifier and call stack, and g' and cs' representing the resulting function identifier and call stack.

Figure 9 gives a selection of the extended semantic rules.

4.4.2 Semantic Properties

Just as for the heap and store, we introduce here the notion of call stack satisfiability for a call stack $cs = (f, x, \rho, s)$.

Definition 9 (Call Stack Satisfiability)**.** *Given a heap typing environment Σ, a typing context Δ and a function g, a call stack cs is said to satisfy g, Δ, Σ, written $cs \vDash g, \Delta, \Sigma$, if:*

- $cs = []$ *or*
- $cs = (f, x, \rho, s) :: cs'$ *such that:*
  - $\exists_{\Gamma,\Gamma'} \forall_v\, v \vDash_\Sigma \Delta_r(g) \Rightarrow \rho[x \mapsto v] \vDash_\Sigma \Gamma \wedge f, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$
  - $cs' \vDash f, \Delta, \Sigma$

Where $\Delta_r(g)$ denotes the return type of g.

Essentially, the call stack satisfiability property guarantees that all the continuations in the current call stack are typable and that all stores in the current call satisfy the corresponding store typing environment if the extended with a value of the appropriate type.

No Aliasing for Call Stack (NACS) In order to deal with aliasing and mutation, our type system enforces the NAOO property, meaning that only closed objects can be referenced by more than one pointer. With the addition of the call stack, we have to extend this invariant to take into account the stores that form the call stack. To this end, we introduce the notion of no aliasing for call stack (NACS).

Definition 10 (No Aliasing for Call Stack)**.** *Let cs be a call stack, ρ_n a store, and Σ a heap typing environment; cs and ρ_n satisfy the no aliasing property with respect to Σ, written $NACS(\Sigma, \rho_n, cs)$, if:*

- $\mathsf{stores}(cs) = [\rho_0, ..., \rho_{n-1}]$
- $\forall_{l \in Locs}\ \forall_{x,y \in Vars}\ \forall_{i,j}\ \rho_i(x) = \rho_j(y) = l \wedge i \neq j \Rightarrow \lfloor \Sigma(l) \rfloor = \bullet$

Essentially, the NACS property means that only closed objects may be referenced by variables pertaining to different stores. In other words, if two variables x and y in different stores ρ_i and ρ_j reference the same location l, then the object pointed to by l must be closed. The proposed type system enforces the NACS invariant. However, such as with the NAOO property, we do not prove that the type system does enforce the NACS invariant directly. Instead, we instrumented the operational semantics so that it also enforces the NACS invariant and will later prove that typable programs cannot be rejected by the semantics for violating the NACS invariant.

4.4.3 Soundness

We now prove that our full type system satisfies the Progress and Preservation [30] properties with respect to the extended operational semantics.

Soundness - Preservation With the introduction of the call stack we have to consider an updated version of our Preservation theorem. Analogously to the type safety theorem for our big-step semantics presented in Subsection 3.6, with the introduction of function calls, we now also require that the global typing context Δ types the program p as hypothesis.

Theorem 5 (Soundness - Preservation)**.** *Let p be a program containing functions g and f, Δ a typing context, Σ and Σ' heap typing environments, Γ and Γ' store typing environments. Let h and h' be heaps, ρ and ρ' stores and s and s' statements. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma'\}$, $\langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle f, \Sigma', h', \rho', cs', s'\rangle$, $h, \rho \vDash \Sigma, \Gamma$, $cs \vDash g, \Delta, \Sigma$ and $\Delta \vdash p$. Then there exist two typing environments $\hat{\Gamma}$ and $\hat{\Gamma}'$ such that the following are true: $h', \rho' \vDash \Sigma', \hat{\Gamma}$; $f, \Delta \vdash \{\hat{\Gamma}\}\ s'\ \{\hat{\Gamma}'\}$; and $cs' \vDash f, \Delta, \Sigma'$.*

Soundness - Progress We finish our small-step semantics section by extending the Progress theorem to the updated transition rules. As in Theorem 5, some hypothesis have to be added. We now require that the global typing context Δ types the program p and that the existing call stack satisfies the current function g, Δ, and the heap typing environment Σ'. Formally:

Theorem 6 (Progress)**.** *Let g be a function and Δ a typing context. Let h be a heap, ρ a store, and s a statement. Suppose that $g, \Delta \vdash \{\Gamma\}\ s\ \{\Gamma_f\}$, $h, \rho \vDash \Sigma, \Gamma$, $cs \vDash g, \Delta, \Sigma$ and $\Delta \vdash p$. Then either $s = \mathsf{skip}$ or $\exists f, \Sigma', h', \rho', cs', s' : \langle g, \Sigma, h, \rho, cs, s\rangle \rightarrow_i \langle f, \Sigma', h', \rho', cs', s'\rangle$.*

5 Related Work

The research literature covers a wide variety of program analysis techniques for JavaScript, such as: type systems [11, 21], abstract interpreters [14], points-to analyses [19], program logics [18, 28], operational semantics [9, 24, 25], just to mention a few. We focus our analysis of the related work on type systems for JavaScript-like languages.

Thiemann [29] was the first to propose a type system for a subset of JavaScript. While this type system considered some of the dynamic aspects of JavaScript, such as extensible objects, dynamic function calls, and type coercions, it

also ignored various important aspects of the language, most notably JavaScript's prototype-based inheritance mechanism. Importantly, the type system proposed by Thiemann is flow-insensitive, meaning that variables and object fields are not allowed to change type over time. To overcome this issue, Anderson et al. [7] later proposed a type system that allows JavaScript objects to evolve in a controlled manner. The key idea behind this work is to classify object fields as potential or definite; potential fields can have their types change while definite fields cannot. The idea of potential/definite fields is reminiscent of our open/closed objects. These two strategies have, however, different trade-offs with both being able to typecheck legal programs that are rejected by the other.

Later, Jensen et al. [20] proposed the first sound type analysis for real JavaScript code, called TAJS. The proposed analysis is flow-sensitive, allowing the types of variables and object fields to change over time, and based on abstract interpretation [13]. The main contribution of this analysis is the design of a complex lattice to reason about unary and binary operations in JavaScript, which takes into account JavaScript's implicit type coercions.

The TypeScript programming language [8] was designed with the goal of adding optional types to JavaScript, taking opportunity of JavaScript's flexibility, while at the same time providing some of the advantages of statically typed languages, such as informative compiling errors and automatic code completion. Client-side JavaScript programs make extensive use of external APIs that are not available for static typing, thus the analysis of TypeScript programs requires the specification of interface declarations for the external libraries that a program may use. However, interface declarations are humanly written and not necessarily by the authors of the libraries, therefore leaving room for errors that can compromise the soundness of the typing process. To solve this program, Feldthaus et al. [15] proposed a method for checking the correction of TypeScript declaration files with respect to JavaScript library implementations.

More recently, some type systems were developed with the objective of enabling efficient ahead-of-time compilation for JavaScript. The first purposed work with this objective was from Choit et al. [12] which supports prototype-based inheritance, structural subtyping, and method updates. Later on, Chandra et al. [10] expanded on this work and incorporated additional annotations, thus enabling the type system to better differentiate between readable and writable object fields.

6 Conclusions

The ECMA-SL project was created with the goal of assisting with the analysis of JavaScript programs by first compiling the programs to be analysed to the ECMA-SL language. To this end, the ECMA-SL project includes a compilation tool chain from JavaScript to ECMA-SL, which has at its core a reference interpreter of the 5th version of the ECMAScript standard called ECMARef5. As the ECMAScript standard is now at its 12th version, in order for the ECMA-SL project to be relevant for the analysis of current JavaScript programs, its compilation pipeline must be adapted to the more recent versions of the ECMAScript standard. To achieve this, one must first adapt the ECMARef5 interpreter that sits at its core. However, such extension is rendered extremely difficult due to the fact that ECMA-SL is an untyped language.

This thesis contributes to the overall ECMA-SL project by designing Typed ECMA-SL, a typed version of ECMA-SL, together with a sound type system for checking Typed ECMA-SL programs. We believe that the implementation of the proposed system would make ECMA-SL substantially easier to use by statically detecting a variety of programming errors that would be, otherwise, only detected dynamically via testing.

In summary, the contributions of this thesis are the following:

Typed ECMA-SL The first contribution of this thesis is the design of Typed ECMA-SL, a typed extension of ECMA-SL [23]. Whilst doing this extension we aimed to minimize the number of extra annotations required, thus enabling already proficient developers in ECMA-SL to easily transition to Typed ECMA-SL.

Type System The second contribution of this thesis is the design of a type system for checking Typed ECMA-SL programs. The purposed type system is both flow-sensitive and sound. Flow-sensitivity allows for program variables and objects to change their types during execution. Soundness ensures that well-typed programs cannot go wrong. Soundness is particularly difficult to achieve in the setting of ECMA-SL due to the dynamicity of the language, which includes extensible objects and the dynamic deletion of object fields.

Soundness Proofs The third contribution of this thesis is the development of two soundness proofs for the proposed type system: one based on a big-step operational semantics and another one based on a small-step operational semantics. The two proofs enabled us to better understand the trade-offs between both types of semantics when proving the soundness of a type system. On the one hand, the small-step operational semantics is more involved than its big-step version, requiring the definition of call stacks and an additional invariant for constraining the ways in which call stacks can be manipulated. On the other hand, the big-step semantics has to model erroneous cases explicitly as they cannot be otherwise differentiated from non-terminating derivations.

6.1 Future Work

Implementing the type system The clear next step is to implement the proposed type system. With the implementation of the proposed type system, the ECMARef interpreter can then be extended and adapted to newer versions of the JavaScript standard.

Enabling Closed Objects to Become Open When a closed object is assigned to one and only one variable, changing its type has a local impact, therefore it might as well be treated as an open object. Thus, by introducing a more fine-grained control mechanism over object aliasing, we could potentially

re-open an object after it being closed. For instance, we envisage the introduction of an uncommit command for opening an object after closing it. This would require further instrumenting the syntax of types to book-keep the pointers to values of a specific type.

Extending the type system In the near future we would like to extend ECMA-SL with support for recursive types, union types and subtyping, as these features would be useful for the development of a typed version of the ECMARef interpreter. For instance, recursive types are essential to model object types with a recursive structure, such as abstract syntax trees.

References

[1] [n.d.]. ECMAScript® Language Specification, 12 Edition / June 2021. https://www.ecma-international.org/wp-content/uploads/ECMA-262_12th_edition_june_2021.pdf. Accessed: 2021-10-30.

[2] [n.d.]. ECMAScript® Language Specification, 5.1 Edition / June 2011. https://262.ecma-international.org/ecma-262/5.1/ECMA-262.pdf. Accessed: 2021-10-30.

[3] [n.d.]. https://github.com/tc39/test262. https://github.com/tc39/test262. Accessed: 2021-10-30.

[4] [n.d.]. JavaScript Across the World Wide Web. https://w3techs.com/technologies/details/cp-javascript/. Accessed: 2021-10-25.

[5] [n.d.]. Node.js. https://nodejs.org/en/. Accessed: 2021-10-25.

[6] [n.d.]. Node.js Increasing Usage. https://w3techs.com/technologies/details/ws-nodejs. Accessed: 2021-10-25.

[7] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. 2005. Towards Type Inference for JavaScript. In *ECOOP 2005 - Object-Oriented Programming*, Andrew P. Black (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 428–452.

[8] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 257–281.

[9] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Sergio Maffeis, Alan Schmitt, and Gareth Smith. 2014. A trusted mechanised JavaScript specification. In *In Proc. POPL*.

[10] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript (Extended Version). *arXiv e-prints*, Article arXiv:1608.07261 (Aug. 2016), arXiv:1608.07261 pages. arXiv:1608.07261 [cs.PL]

[11] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and Precise Type Checking for JavaScript. arXiv:1708.08021 [cs.PL]

[12] Wontae Choi, Satish Chandra, George C. Necula, and Koushik Sen. 2015. SJS: A Type System for JavaScript with Fixed Object Layout. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9291)*. Springer, 181–198.

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[14] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A Parallel Abstract Interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, USA, 34–45.

[15] Asger Feldthaus and Anders Møller. 2014. Checking correctness of typescript interfaces for javascript libraries. In *In Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA*. ACM, 1–16.

[16] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part i: A Multi-Language Platform for Symbolic Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 927–942. https://doi.org/10.1145/3385412.3386014

[17] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienundefined, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript Verification Toolchain. *Proc. ACM Program. Lang.* 2, POPL, Article 50 (Dec. 2017), 33 pages. https://doi.org/10.1145/3158138

[18] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. 2012. Towards a Program Logic for JavaScript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 31–44. https://doi.org/10.1145/2103656.2103663

[19] Dongseok Jang and Kwang-Moo Choe. 2009. Points-to Analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii) *(SAC '09)*. Association for Computing Machinery, New York, NY, USA, 1930–1937. https://doi.org/10.1145/1529282.1529711

[20] Simon Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript, Vol. 5673. 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[21] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[22] Gilles Kahn. 1987. Natural Semantics. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS '87)*. Springer-Verlag, Berlin, Heidelberg, 22–39.

[23] Luis Loureiro. 2021. ECMA-SL - A Platform for Specifying and Running the ECMAScript Standard.

[24] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–325.

[25] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. *ACM SIGPLAN Notices* 50 (06 2015), 346–356. https://doi.org/10.1145/2813885.2737991

[26] Gordon Plotkin. 2004. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.* 60-61 (07 2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001

[27] F. Quinaz. 2021. Precise information flow control for javascript.

[28] José Fragoso Santos, Philippa Gardner, Petar Maksimović, and Daiva Naudžiūnienė. 2017. Towards Logic-Based Verification of JavaScript Programs. In *Automated Deduction – CADE 26*, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 8–25.

[29] Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. In *Programming Languages and Systems*, Mooly Sagiv (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 408–422.

[30] Andrew K. Wright and Matthias Felleisen. 1992. A Syntactic Approach to Type Soundness. *INFORMATION AND COMPUTATION* 115 (1992), 38–94.