A Reference Implementation of ES6 Built-in Libraries

Jorge Pedreira Cardoso Brown

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Faustino Fragoso Femenin dos Santos

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. Pedro Tiago Gonçalves Monteiro

October 2022

Acknowledgments

I would firstly like to thank my supervisor, Prof. José Fragoso Santos, for his confidence in my abilities and the words of encouragement to keep me engaged and motivated. His continued support and tremendous availability were of great importance for the success of this thesis.

Secondly, I would like to thank the other students I collaborated with in this thesis: David Belchior, Nuno Policarpo, Leonor Barreiros, João Silveira, Manuel Costa, Tomás Tavares. Their help allowed us be much more effective and achieve a lot in a short amount of time.

Finally, I would like to thank my friends who always helped me when I needed it most.

Abstract

JavaScript is the *de facto* language for implementing client side Web applications. It is specified in the ECMAScript standard, a long and complex document written in English that is updated with every new iteration of the language. Despite its popularity, JavaScript is not always coherent and understandable semantically, and its dynamic paradigm makes it hard to statically analyse. ECMAScript reference interpreters are artifacts produced to reason about the language in a controlled environment. To this end, we will leverage the ECMA-SL project, a research effort at IST whose goal is to build an executable version of the standard. Currently, the ECMA-SL project comes with an interpreter, ECMARef5, for the 5th version of the standard, which is now at its 12th version. We plan to assist with the transition of ECMARef5 to the 6th version of the standard, by aiding in the implementation effort of the built-in libraries of the ECMAScript 6 Standard. Alongside other strategies employed in the ECMA-SL project, to guarantee the quality of our implementation we test it against Test262, the official ECMAScript conformance test suite.

Keywords: ECMAScript, Specification Language, Reference Interpreters, Dynamic Languages, Test262.

Resumo

O JavaScript é a linguagem *de facto* para implementar aplicações clientes na *Web*. Ela é especificada no *standard* ECMAScript, um longo e complexo documento escrito em Inglês que é atualizado com cada nova iteração da linguagem. Apesar da sua popularidade, o JavaScript nem sempre é coerente e compreensível semanticamente, e o seu dinamismo faz com que a sua análise estática seja difícil. Interpretardores de referência da linguagem ECMAScript são artifactos feitos para raciocinar sobre a linguagem num ambiente controlado. Com este fim, vamos tirar proveito do projecto ECMA-SL, um projecto de investigação no IST cujo objectivo é construir uma versão executável da especificação em oposição à versão textual feita pelo *standard*. Actualmente, o projecto ECMA-SL é composto de um interpretador, ECMARef5, para a 5ª versão do *standard* que está agora na sua 12ª versão. A nossa intenção é de apoiar com a transição do ECMARef5 para a 6ª versão do *standard*, ajudando no esforço de implementação das bibliotecas built-in do ECMAScript 6. Para garantir a qualidade da nossa implementação, conjuntamente com outras estratégias usadas no projecto ECMA-SL, testamo-la contra a *test suite* oficial de conformidade do *standard* ECMAScript, a Test262.

Keywords: ECMAScript, Linguagem de especificação, Interpretadores de referência, Linguagens dinâmicas, Test262

Contents

List of Tables

List of Figures

Chapter 1

Introduction

JavaScript is one of the most used programming languages in the world and the most commonly used to develop client-side Web applications, such as e-mail clients or online banking platforms, and so the Web has a big dependence on it. Recently it also has been gaining traction as a language for server-side scripting, specially with the Node.js [1] runtime, and even as a language for the development of various desktop applications, mainly using the Electron framework [2], such as Discord [3] and Visual Studio Code [4]. However, these applications do not all use the same JavaScript engine. Although a lot of them do use Google's V8 [5] JavaScript engine, there are other alternatives and it is critical that they all behave in the same manner. Should this not be the case, not only is it possible that some JavaScript applications do not execute properly in certain runtime environments, but it could also lead to critical security flaws. In order to mitigate this possibilities, it was deemed by Netscape [6], original creators of JavaScript, that a standardization of the JavaScript language was necessary and so the ECMAScript standard was created in collaboration with ECMA International [7].

The ECMAScript standard (ES) [8] consists of the specification of the ECMAScript language's syntax and semantics. The following of the standard's specification by every JavaScript engine, should mean that they all behave in the same manner regardless of how they handle some of the implementation-dependant aspects in their implementation. This means that developers that build applications using the ECMAScript language can be confident that their application will behave as expected, independently of the engine it is run on. The ECMAScript standard is a long document written in English that describes the behavior of the language as if it was the pseudo-code of an interpreter, giving detailed steps on how to interpret each instruction. Over the years it has evolved substantially, regularly increasing in its size and detail. Figure 1.1 illustrates this evolution and shows us that sometimes the standard can double in size in a single iteration. Despite the large amount of iterations, the standard is not easy to maintain or alter, with the process of adding features being extremely complex, requiring new features to uphold the invariants of the language's semantics and maintain backwards compatibility, guaranteeing that the behaviour of older features remains unchanged. The document is managed and maintained by the TC39 committee [9] which is composed of JavaScript developers, browser representatives, academics, etc. As of now, the committee has a well established methodology used to extend and update the standard's specification called "The TC39 Process" [10]. This process is divided into 5 stages: (1) proposal of new features for the ECMAScript language; (2) selection of the best ideas and discussion on possible solutions and what challenges they may entail; (3) describing their syntax and semantics using formal specification language; (4) refinement using feedback from the committee, implementations and users; (5) and eventually adding them to the standard in a polished state.

Naturally, not all implementations of the ECMAScript language follow the standard's described behavior exactly. They can be built using different programming languages and it may be convenient to

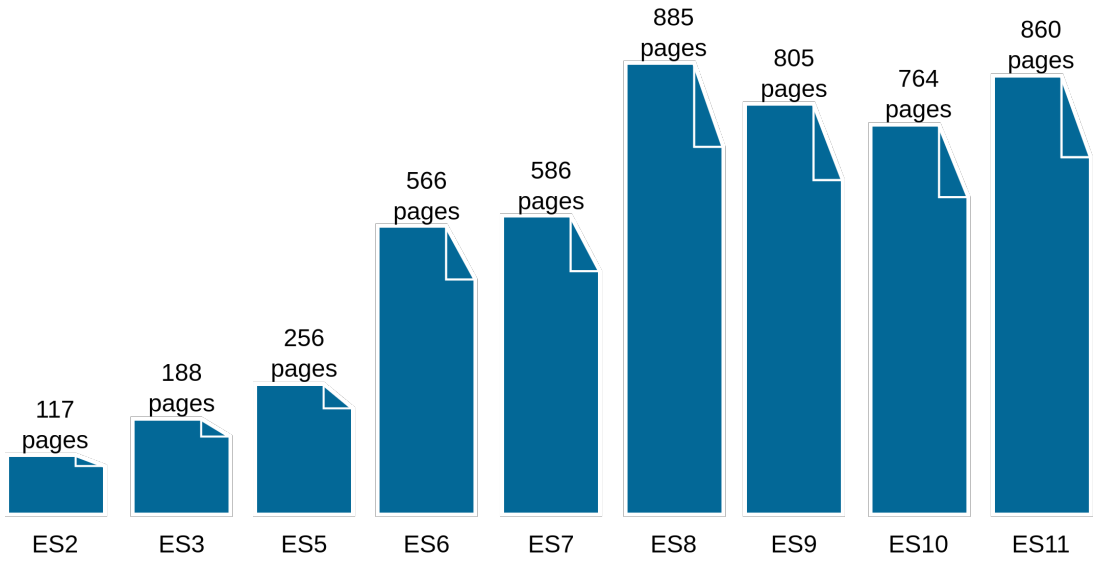


Figure 1.1: Evolution of the number of pages describing the ECMAScript standard over time.

represent some components of the standard in a different, although equivalent, manner. Most of the discrepancies are due to performance reasons. In particular, the industrial JavaScript implementations use JIT (just-in-time) compilation to guarantee the performance of their applications. If the JavaScript implementations do not follow the standard exactly, then how can it be determined that they are ECMAScript compliant? Currently, this is done through exhaustive testing. Alongside the ECMAScript standard, the TC39 committee also maintains an official test suite called Test262 [11] whose purpose is to measure the conformity of a JavaScript interpreter to the official ECMAScript specification. However, testing is an incomplete method for determining this, as there have already been multiple bugs found in JavaScript engines that were not discovered by the test suite [12].

An alternative methodology is to maintain a reference implementation that follows the standard line-by-line and use this implementation has an oracle to test the conformity of other implementations of the language. This could be done by comparing the behavior of those implementations with the behavior exhibited by the oracle on concrete programs. In this sense, multiple academic projects have cropped up with the intent of producing a reference interpreter for JavaScript [13, 14, 15, 12, 16, 17, 18]. However, most of these reference implementations do not support the standard' built-in libraries, with those that partially do it only doing it at a very small scale. In fact, we can fairly say that most ECMAScript reference interpreters ignore the language's built-ins. However, they are an essential part of the language and their testing alongside the core of the language is critical to fully understand and reason about JavaScript implementations.

With the goal of filling the void left by the absence of a complete reference interpreter the *ECMA-SL project* [19, 20, 21] was created. The main goal of the project is to maintain a complete (with built-in support) executable specification of the standard written in an intermediate language specifically designed to do so, the ECMA-SL language, which stands for ECMAScript-Specification Language. The semantics and algorithms described in the ECMAScript standard are written as if they were the pseudo-code of a reference interpreter (executable specification). The ECMA-SL language mimics this pseudo-code, making the specification of a reference interpreter, almost a work of copying the semantics already described in the English standard. From the specification of the standard in ECMA-SL, it is possible to create numerous other artifacts, namely a natural language version of the standard structured as an HTML document, similar to the original. Currently, the ECMA-SL project includes a full specification of

ES5 [22] (the 5th version of the ECMAScript standard) and a partial implementation of ES6 [8] (the 6th version of the ECMAScript standard). Also included in the ECMA-SL project, are multiple tools in development that use the interpreter for many other purposes as Figure 1.2 shows. Some of the most relevant of them are HTML2ECMA-SL [20] and ECMA-SL2English [19], whose purpose is, respectively, to convert the standard's HTML document written in English to ECMA-SL code and to convert the executable specification of the standard back to its original form in natural language.

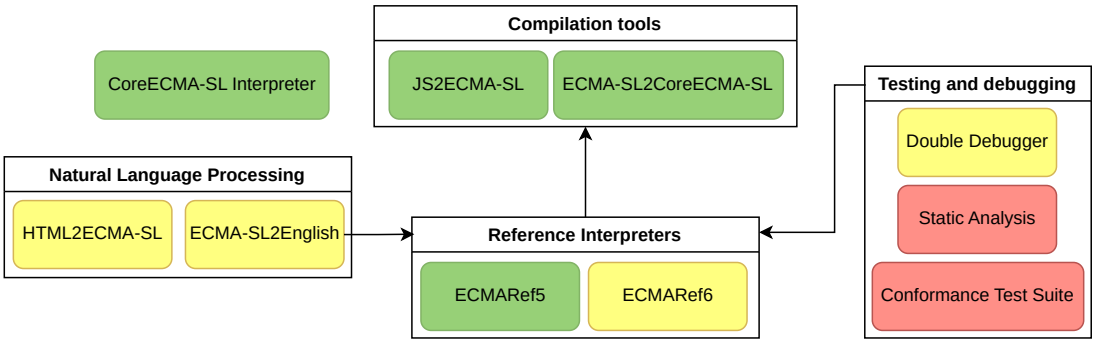


Figure 1.2: Currently available (green), in development (yellow) and future (red) tools of the ECMA-SL project. Arrows signify dependency.

The goal of this thesis is to build upon what already exists in the ECMA-SL project. We extend the ECMARef6 reference interpreter, the executable specification of ES6, increasing its coverage of the standard. This will be achieved through the implementation of built-in libraries of ES6 and their dependencies using a line-by-line strategy to guarantee that our implementation is correct. In particular, we will focus on the following built-in libraries:

- `ArrayBuffer` (Section 4.1) - the `ArrayBuffer` library introduces byte-level datatypes and operations;
- `DataView` (Section 4.2) - provides high-level abstractions to read and write bytes from `ArrayBuffer` instances, using `get` and `set` methods;
- `TypedArray` (Section 4.3) - similar to the `DataView` library, but instead provides an array-like interface to interact with `ArrayBuffer` objects;
- `Symbol` (Section 4.4) - introduces an entirely new type of property key;
- `Proxy` (Section 4.5) - provides a reflection mechanism used to override the semantics of the language.

Besides the libraries stated above, I also coordinated the implementation of all the other built-in libraries, being in charge of testing them, fixing eventual bugs, and making sure that they matched the specification. The development of all libraries was jointly done by me, and a group of 6 undergraduate students during a summer internship at INESC-ID.

From a technical point of view, the implementation of these libraries posed a few challenges. In order to successfully implement the `Symbol` library, it was necessary to change the internal representation of ECMAScript objects in ECMA-SL as the previous representation was not fit to handle `Symbol` property keys. In the implementation of the `ArrayBuffer` library, the introduction of two new datatypes, arrays and bytes, to the ECMA-SL language was required. Alongside the new types, new operators for manipulating them were also implemented, with the most important ones being capable of converting a numerical value to its representation as an array of bytes and vice-versa.

In order to demonstrate that we attained our goals, we tested our reference implementation (ECMARef6) against the official test suite, placing special emphasis on the tests that target the 6th version of the standard and the built-in libraries. Overall, we passed 93.99% out of 13253 tests relative to all the built-in libraries and 95.44% out of 1857 tests relative to the ones we have emphasized. Although we do not pass 100% of the tests, the large majority of the failing tests, fail due to missing features in ECMARef6 that are part of the core of the ECMAScript language and not its built-in libraries.

This thesis is structured in the following way: in Chapter 2 we introduce all the background knowledge pertinent to this thesis, more specifically, background on the ECMA-SL Project in Section 2.1 and on the ECMAScript standard in Section 2.2; in Chapter 3 we present and discuss other projects related to the work done in the context of this thesis, including other reference implementations; in Chapter 4 we will present the main contribution of this thesis, analyzing the standard's specification as well as the implementation in ECMARef6 of the `ArrayBuffer` (Section 4.1), `DataView` (Section 4.2), `TypedArray` (Section 4.3), `Symbol` (Section 4.4) and `Proxy` (Section 4.5) built-in libraries; in Chapter 5 present our evaluation methodology in more detail as well as our results; finally in Chapter 6 we go over some conclusions on the work performed as well as possible future endeavours.

Chapter 2

Background

This chapter introduces the necessary background for this thesis. In Section 2.1, we explain the ECMA-SL project and its main components. In Section 2.2, we give an overview of the ECMAScript standard with a special focus on: (a) the representation of ECMAScript objects, which constitute the fundamental datatype of the language (Subsection 2.2.2); (b) the ECMAScript built-in libraries (Subsection 2.2.3), which are the main focus of this thesis.

2.1 ECMA-SL Project

The ECMA-SL project [19] is a project that aims to build the most complete ECMAScript reference interpreter to date and various tools to support the analysis and specification of the ECMAScript language. The project is built around the ECMA-SL language, that stands for ECMAScript-Specification Language, and was purposefully built to define the standard's semantics.

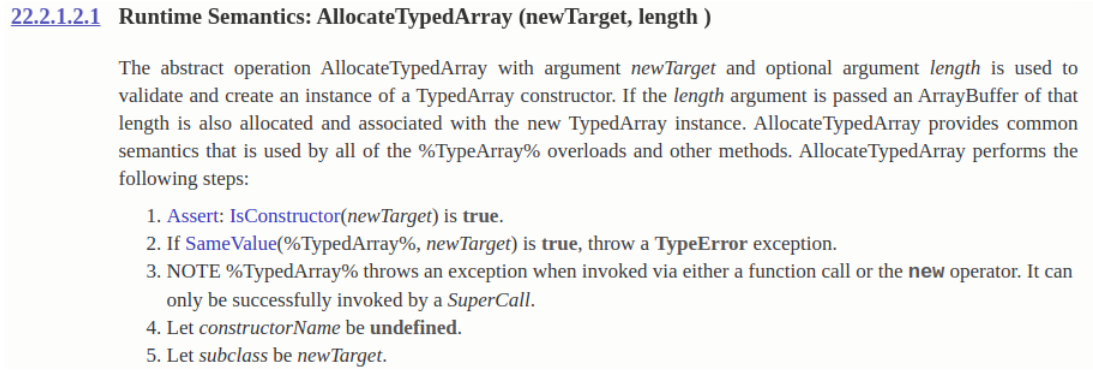
The ECMA-SL language is a *weakly-typed*, *imperative* language that has some of the dynamic properties of JavaScript like dynamic function calls and the ability to create and delete object properties and to evaluate code at runtime. The language was done with the goal of being as similar as possible to the standard's pseudo-code and therefore adopts all its meta-constructs to allow a line-by-line match of pseudo-code instructions to ECMA-SL instructions. An ECMA-SL program consists of only top-level functions, with its entry point being the top-level function named "main". Given that there are only top-level functions, each one of them has its own scope.

In Figure 2.1, we can see a side-by-side comparison of the standard's pseudo-code with the equivalent ECMA-SL code. In this example, we can see that over each line of the ECMA-SL code there is a comment with the corresponding line of the standard showing that there is indeed a line-to-line match and also making it easy to see that they are similar. In lines 10 and 12, we see that values were assigned to the variables `constructorName` and `subclass` without them being explicitly declared, because there are no variable declaration statements in the ECMA-SL language. Consequently, there is also no way to define variable types, making ECMA-SL a weakly typed language. In line 5 we can see the variable `|TypedArray|` is referenced without a previous assignment to it. This is because it is a global variable of the interpreter that had been initialized before the execution of this function. Global variables of the interpreter are surrounded by pipe characters and inaccessible to ECMAScript programs.

Based of the ECMA-SL language and to support the specification and analysis of the ECMAScript standard, the ECMA-SL project includes the following tools, which can be visualized in Figure 2.2:

1. ECMARef5 - a reference interpreter written in ECMA-SL which supports the entirety of the 5th version the ECMAScript standard, meaning that it faithfully matches each line of pseudo-code with

(a) Fragment of section 22.2.1.2.1 which describes the semantics of the allocation of `TypedArray` instances.



22.2.1.2.1 **Runtime Semantics: AllocateTypedArray (newTarget, length)**

The abstract operation AllocateTypedArray with argument *newTarget* and optional argument *length* is used to validate and create an instance of a TypedArray constructor. If the *length* argument is passed an ArrayBuffer of that length is also allocated and associated with the new TypedArray instance. AllocateTypedArray provides common semantics that is used by all of the %TypeArray% overloads and other methods. AllocateTypedArray performs the following steps:

1. Assert: IsConstructor(*newTarget*) is **true**.
2. If SameValue(%TypedArray%, *newTarget*) is **true**, throw a **TypeError** exception.
3. NOTE %TypedArray% throws an exception when invoked via either a function call or the **new** operator. It can only be successfully invoked by a *SuperCall*.
4. Let *constructorName* be **undefined**.
5. Let *subclass* be *newTarget*.

```
function AllocateTypedArray(newTarget, length) {
  /* 1. Assert: IsConstructor(newTarget) is true. */
  assert(IsConstructor(newTarget) = true);
  /* 2. If SameValue(%TypedArray%, newTarget) is true, throw a TypeError exception */
  if (SameValue(|TypedArray|, newTarget) = true) {
    throw TypeErrorConstructorInternal()
  };
  /* 3. NOTE %TypedArray% throws an exception when invoked via either a function call or the new
    operator. It can only be successfully invoked by a SuperCall. */
  /* 4. Let constructorName be undefined. */
  constructorName := 'undefined;
  /* 5. Let subclass be newTarget. */
  subclass := newTarget;
    ...
};
```

Listing 2.1: Fragment of the semantics of allocation of a `TypedArray` in ECMA-SL, as defined in the ECMARef6 reference interpreter.

Figure 2.1: Comparison between the standard's pseudo-code and the ECMA-SL language.

an ECMA-SL statement.

2. ECMARef6 - a reference interpreter of the 6th version the ECMAScript standard currently being written in ECMA-SL which will in the future also support the entire standard.

3. JS2ECMA-SL - a compiler that parses a JavaScript program into its abstract syntax tree (AST) and generates an ECMA-SL file with a single function called `buildAST` which will build the JavaScript program's AST in the ECMA-SL heap.

4. ECMA-SL2CoreECMA-SL - a compiler that transforms ECMA-SL code into its core version by swapping some meta-constructs for equivalent ones, e.g turning every loop into a while loop, making the code easier to interpret.

5. CoreECMA-SL interpreter - an interpreter of CoreECMA-SL code built in the OCaml [23] language.

6. HTML2ECMA-SL - a tool that converts a section of the standard's HTML document into the corresponding ECMA-SL code, in an attempt to automate and accelerate the process of writing the ECMAScript reference interpreters.

7. ECMA-SL2English - a tool that converts a section of the reference interpreters built in ECMAScript into the corresponding HTML matching the format and structure of the standard's HTML.

With these tools it is possible to execute JavaScript programs with one of the two reference interpreters. This process, demonstrated visually in Figure 2.3, has as its first step the building of the abstract syntax tree (AST) of the JavaScript program, via the JS2ECMA-SL tool which will produce as output the file **ast.esl**, an ECMA-SL file containing a single function called **buildAST**, that will build the JavaScript

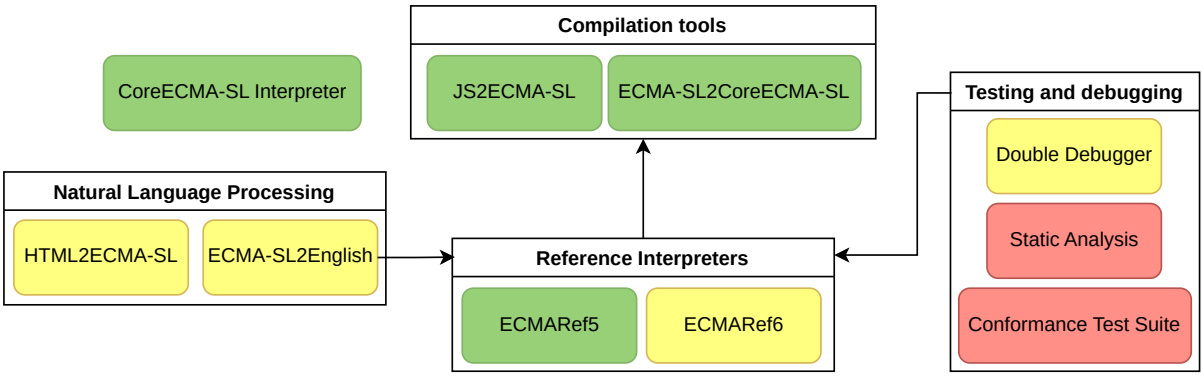


Figure 2.2: Currently available (green), in development (yellow) and future (red) tools of the ECMA-SL project. Arrows signify dependency.

program's AST in the ECMA-SL heap. At this point we have two files, **ast.esl** and **ESX_interpreter.esl**, where X is replaced by 5 or 6 depending on the ECMAScript version wanted. The **ES6_interpreter.esl** file is a bundled version of ECMARef6 that condenses the reference interpreter into a single ECMA-SL file. Both of these files are imported into a single **out.esl** file, whose code calls the reference interpreter on the result of a **buildAST** call. The next step is to compile this file into the Core version of ECMA-SL using the ECMA-SL2CoreECMA-SL tool, generating a CoreECMA-SL file, **core.cesl**. The final step is to use the CoreECMA-SL interpreter, built with the OCaml language, to run the **core.cesl** file.

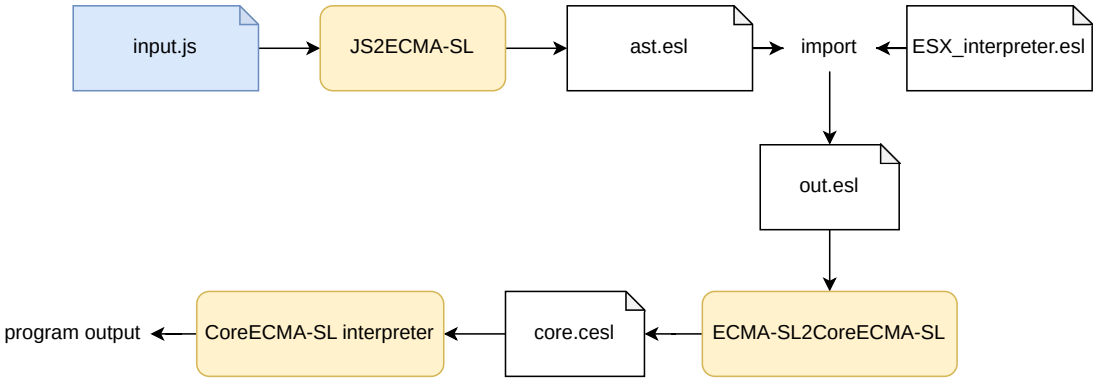


Figure 2.3: Execution pipeline of a JavaScript program.

In the two next subsections, we will give a more detailed account of the HTML2ECMA-SL and ECMA-SL2English tools as their improvement will be the main focus of this thesis.

2.2 ECMAScript Standard Overview

In this section we will take a deeper look at the ECMAScript standard, beginning with the a more general overview of its purpose and components. We will then present one of its most crucial datatypes, the object. Finally, we will give a lightweight description of some of the built-in libraries in the final subsection.

2.2.1 General Description

The ECMAScript Standard describes the various iterations of ECMAScript language's syntax and semantics. In this project, we will specifically look at the 6th version of the standard. The ECMAScript language is what the vast majority of web apps runs on and as such, all the major web browsers have

adopted the standard to a certain extent, not following completely all the specifications in order to increase their JavaScript engine's performance. This means that although their output may be identical to the expected, there are some inconsistencies between how that output is achieved between different browsers and an implementation that followed the standard in its entirety. This complete adoption of the standard into a formalization of its semantics is called a reference interpreter, which is what the ECMA-SL project aims to accomplish with ECMARef6, relative to the 6th version of the ES standard.

The standard is divided into three main components: the sections related to grammar and syntax, the sections that describe the semantics of the language's core, and the sections that describe the semantics of the built-in objects, which represent the language's standard library.

Overall, the standard describes the ECMAScript language as being object-based, with objects being collections of properties. Properties can in turn point to other objects, functions (callable objects) or primitive values. The ECMAScript language includes a prototype-based mechanism for implementing inheritance: each object has a "parent" object to which it refers to if it does not possess a property which is looked up. The behavior of an object looking up a property on its prototype and it in turn looking up that same property on its prototype until one of these objects contains the property is called *prototype chaining*. On the version of the standard studied on this project, classes were introduced, however, ECMAScript did not become a class-based language like C++ or Java. Classes serve merely as syntax sugar for the definition of constructor functions and prototype objects.

In charge of developing and improving the standard is the *TC39 committee* which is composed of JavaScript developers, ECMA members, web browser representatives, etc. This committee determines the future of the ECMAScript language choosing what new features should be added and what old features should be improved. However, this task is not easy as ECMAScript is a complex language with lots of invariants which must be guaranteed at all times and it is mandatory that the features added or modified do not unintentionally alter any previous behaviors, in order to guarantee backwards compatibility and not "break the web". To make matters worse, as web browsers do not completely follow the standard, as mentioned before, their JavaScript engines cannot be used to test these new features. This issue is something that many projects and teams have tried to solve, including the ECMA-SL project, but without much success, at least from the perspective of adoption by the committee which is still not accompanied by a reference implementation. These other projects and their reference implementations will be explored in the Related Work chapter (3).

2.2.2 ECMAScript Objects

In this subsection, we will delve deeper into what is an object in the ECMAScript standard and introduce the concept of `Property Descriptor` which is a crucial component of objects.

The ECMAScript standard defines various types which are organized into two categories:

- ECMAScript Language Types - this category represents the types available to ECMAScript programs, such as `string`, `boolean`, `null` and `object`;
- ECMAScript Specification Types - this category represents the types that are only available inside the ECMAScript interpreter and not accessible to ECMAScript programs, for example, the `Completion` and `List` types.

We will focus on one of the ECMAScript Language Types, the object type. Object values, as mentioned before, consist of collections of properties. The properties of these objects can also be split into two categories: *internal* properties; and *named* properties.

Internal properties Internal properties, identified in the standard by being encased in double square brackets (`[[ ]]`), are properties that hold values of either of the types mentioned above and are used to describe some aspect of the object that contains them, i.e. metadata, or simply as "private" properties. They are accessible only by the interpreter code and it should be impossible for JavaScript programs to directly access or modify them. All objects have two internal properties and multiple internal methods crucial for managing their properties. Their internal properties are the following:

1. `[[Prototype]]` - A value that can be either `null` or a reference to the object's prototype object that is used to implement prototype-inheritance.

2. `[[Extensible]]` - A boolean value that determines if properties can be added to the object. Even though this property can be set from true to false, the opposite cannot happen, as a non-extensible object remains so forever.

The internal methods of objects are what specifies their semantics and are called during the most fundamental operations of the language. For example, the `[[Get]]` internal method, which retrieves that value of a specified *named property*, is called every time a property of an object is accessed in ECMAScript code, be it via bracket or dot notation. Every value of the object type has a fixed set of internal methods that define its behavior, but they are polymorphic, meaning that different objects can have different algorithms for methods with the same name. Each internal method has its own default implementation and an object whose internal methods all follow their respective default implementation are called *ordinary objects*. There are, however, some objects, called *exotic objects*, that require a change from the default behavior and so have different implementations for their internal methods. For example, some objects may have a specialized internal `[[Get]]` method to alter the way properties are retrieved from them. The presence of internal properties does not mean an object is exotic, only the deviation from the default internal methods puts the object in that category. Some noteworthy exotic objects are `String` and `Array` objects. `Function` objects are an outlier, as they are the only objects with more internal methods than the ones present in ordinary objects, nevertheless, they are not exotic objects has they do not change the implementation of those methods.

Figure 2.4 shows a representation of an object created using the `Object` constructor that is assigned to a `dog` variable. We can see that the object has the `[[Prototype]]` internal property, which references `Object.prototype` since the `Object` constructor was used for its creation, and the `[[Extensible]]` internal property with the default primitive value `true`. The object also has its internal methods, which are not all represented in the diagram since an extensive listing of them is not relevant for this section. In the case of the object created in the example, all the internal methods have their default implementations as all objects instantiated via the `Object` constructor are ordinary objects.

```
1 const dog = new Object();
```

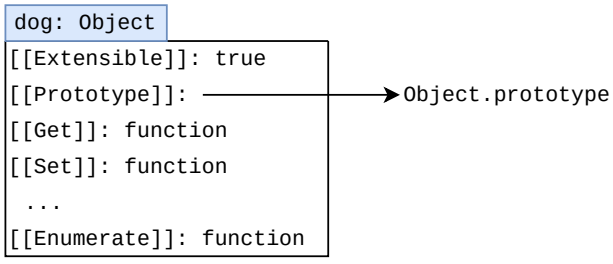


Figure 2.4: Diagram of an object and its internal properties.

Named properties Named properties are the properties in an object that will be directly accessed and created by a JavaScript program during runtime and can be of two types: *data properties* or *accessor properties*. These properties are described by a *record* called a *property descriptor*. A record is an aggregate of named fields whose values can be of any type. As there are two different types of properties, there are also two types of property descriptors to match them: *data property descriptors* and *accessor property descriptors*.

In Figure 2.5, we have a piece of JavaScript code which, once again, contains an assignment of an object to a `dog` variable. This object has a named property `race` with the value "Beagle". As discussed above, we can see that the object resulting of the evaluation of this statement has the two internal properties, `[[Prototype]]` and `[[Extensible]]`, the internal methods are omitted for the sake of clarity, and a named property which is described by a property descriptor. In this case, the "race" property would be a data property as the value was supplied and not described as a get function. The syntax for defining accessor properties will be defined ahead. An interesting note to take is that even though all named properties must be associated with a property descriptor, internal properties are not under the same restriction, meaning that they may or may not be wrapped inside property descriptors. Figure 2.5 shows a simplified version a property descriptor which will be expanded on in the following paragraphs.

```
1 const dog = {
2     race: "Beagle"
3 };
```

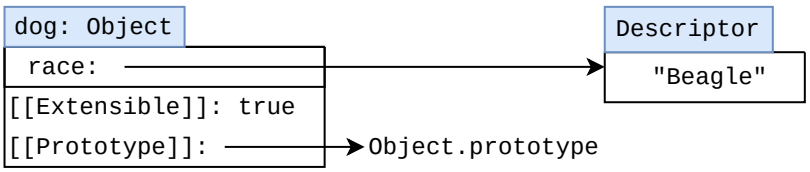


Figure 2.5: Internal representation of an ordinary object and its property.

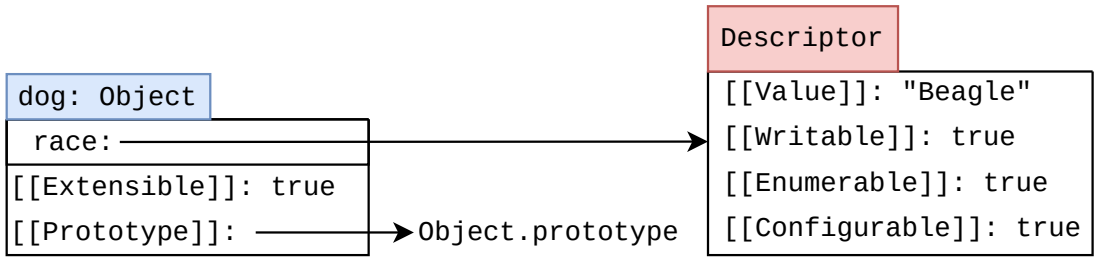
Data property descriptors As already mentioned, property descriptors are records, which are groups of named fields. Data property descriptors are four-field records that hold an explicit data value and information about the property itself. These fields are the following:

1. `[[Value]]` - The actual value of the property. This value's type can be: undefined, null, boolean, string, symbol, number or object.
2. `[[Writable]]` - A boolean value that determines if the value in the `[[Value]]` field can be changed.
3. `[[Enumerable]]` - A boolean value that determines if this property should be considered when iterating through the properties of the object.
4. `[[Configurable]]` - A boolean value that determines if the rest of the fields in the property descriptor can be altered. It also determines if the property can be removed from the object.

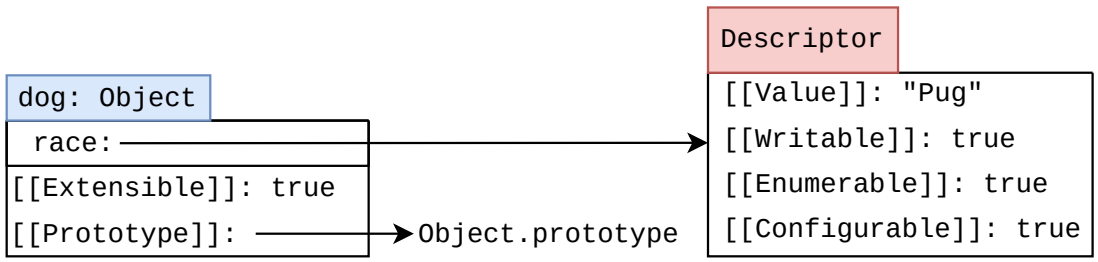
Figure 2.6 is an expansion of the example provided above in Figure 2.5. We can see that line 1 is the same as the previous example, with Figure 2.6a representing the state after the evaluation of this line of code. Here the property descriptor is complete and describes a data property, which we can assert by the presence of the fields `[[Value]]` and `[[Writable]]` in the record. In line 2, the string value "Pug" is assigned to the named property "race" of the "dog" object, which provokes a change only in the `[[Value]]` field of the property descriptor, as can be seen in Figure 2.6b. From lines 3 to 6, the values of the fields `[[Writable]]` and `[[Configurable]]` of the property descriptor are both set to false. In the

final line, an assignment to the property “race” is done once again, however, since both `[[Writable]]` and `[[Configurable]]` are set to false, the value in the `[[Value]]` field remains the same, as can be observed in Figure 2.6c. Note that, as mentioned before, the ability to change the `[[Value]]` field depends exclusively on the value of the `[[Writable]]` field. However, if the value of `[[Configurable]]` is set to true, even if the value of `[[Writable]]` is false, we can still do the assignment by first setting the `[[Writable]]` field to true. For this reason, when aiming to make a property read-only, one must set both `[[Writable]]` and `[[Configurable]]` to false.

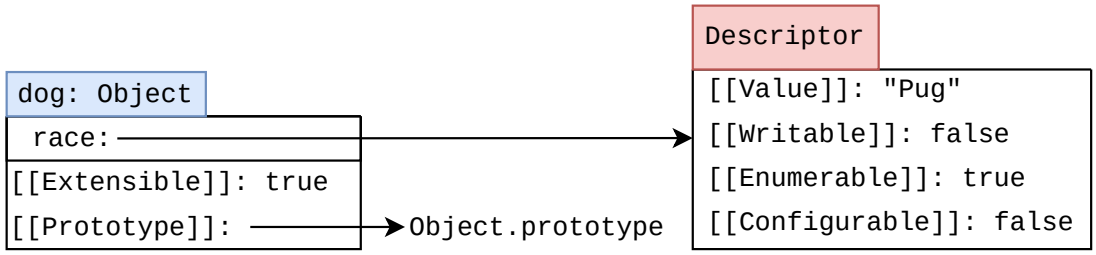
```
1  const dog = { race: "Beagle" };
2  dog.race = "Pug";
3  Object.defineProperty(dog, "race", {
4      configurable: false,
5      writable: false
6  });
7  dog.race = "Beagle";
```



(a) State after line 1.



(b) State after line 2.



(c) State after line 7.

Figure 2.6: Graphical representation of the variation of internal state caused by changes in properties.

Accessor property descriptors Accessor property descriptors also have four fields, two of which are the `[[Enumerable]]` and `[[Configurable]]` fields, which maintain the same function they had in data property descriptors. However, contrary to data property descriptors, these property descriptors do not hold an explicit data value, they hold function references to *getter* and *setter* methods, using the `[[Get]]` and `[[Set]]` fields respectively. These two fields can be defined as follows:

1. `[[Get]]` - A pointer value to a function object that will be called when an attempt is made to read this property’s value. The return value of this function will be passed on as the property’s value. It

can be undefined, in which case no call occurs. The function referenced by this field must have no formal parameters.

2. `[[Set]]` - A pointer value to a function object that will be called when an attempt is made to change this property's value. It can be undefined, in which case no call occurs. The function referenced must have exactly one formal parameter.

In Listing 2.2, we can see the definition of an object with a named accessor property called "height" with both a setter and a getter method. The result of evaluating this code snippet can be seen in Figure 2.7, where there is an object with the expected internal properties of an ordinary object and a single named property, as both methods have the same name (height). We can see that the property descriptor is an accessor property descriptor by the fact that the record possesses the fields `[[Get]]` and `[[Set]]`. In this case, the value of those fields are references to function objects. It is important to distinguish between an accessor property, which will result in user code being executed and a data property whose value is a function object, like the one in Listing 2.3. In the case of these two examples, the evaluation of the instruction `dog.height`, would resolve to 80 in the case of Listing 2.2 and a reference to a function object in the case of Listing 2.3.

Listing 2.2: Definition of an object with an accessor property.

```
1 const dog = {
2     get height() {
3         return 80;
4     },
5     set height(value) {
6         this._height = value;
7     }
8 };
```

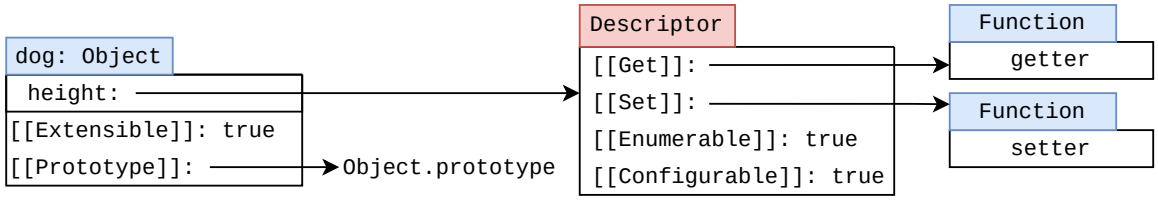


Figure 2.7: Internal state after the execution of Listing 2.2.

Listing 2.3: Definition of an object with a named data property whose value type is a reference to a function object.

```
1 const dog = {
2     height: function() {
3         return 80;
4     }
5 };
```

2.2.3 ECMAScript Built-ins

We have now seen the object type and some of its inner workings. In this subsection we will present the built-in libraries of the standard, starting by what they are and how they are integrated into the language and then we will do a brief description of some of the libraries, specifically the ones that were specifically implemented in the context of this thesis.

Firstly, it is important to make a distinction between the object datatype and type of instances. Values of the object datatype are collections of properties as presented and dissected previously. All the built-in libraries have objects as entry points which are all values of the object datatype. Some of this objects

are also constructors and create instances. These instances may or may not be values of the object datatype, but they are instances of the constructor type. For example, the `Symbol` constructor is an object that produces `Symbol` values, which are not objects, but are still of the `Symbol` type. The expression "an `Array` object" would be referring to a value of the object datatype that was instantiated using the `Array` constructor. Moving forward, the word type will usually refer to the instance type and not the datatype.

The Built-in libraries are objects that have already implemented functionality via their methods and their prototype's methods and are made available through properties of the `Global Object` which is available in all scopes of execution. All of the built-in libraries populate a property of the global object with an object of their own. Figure 2.8 shows all the built-in objects and their corresponding categories in the standard. The libraries highlighted in a blue tint, like `Symbol` and `TypedArray`, are the ones that were specifically implemented in the context of this thesis.

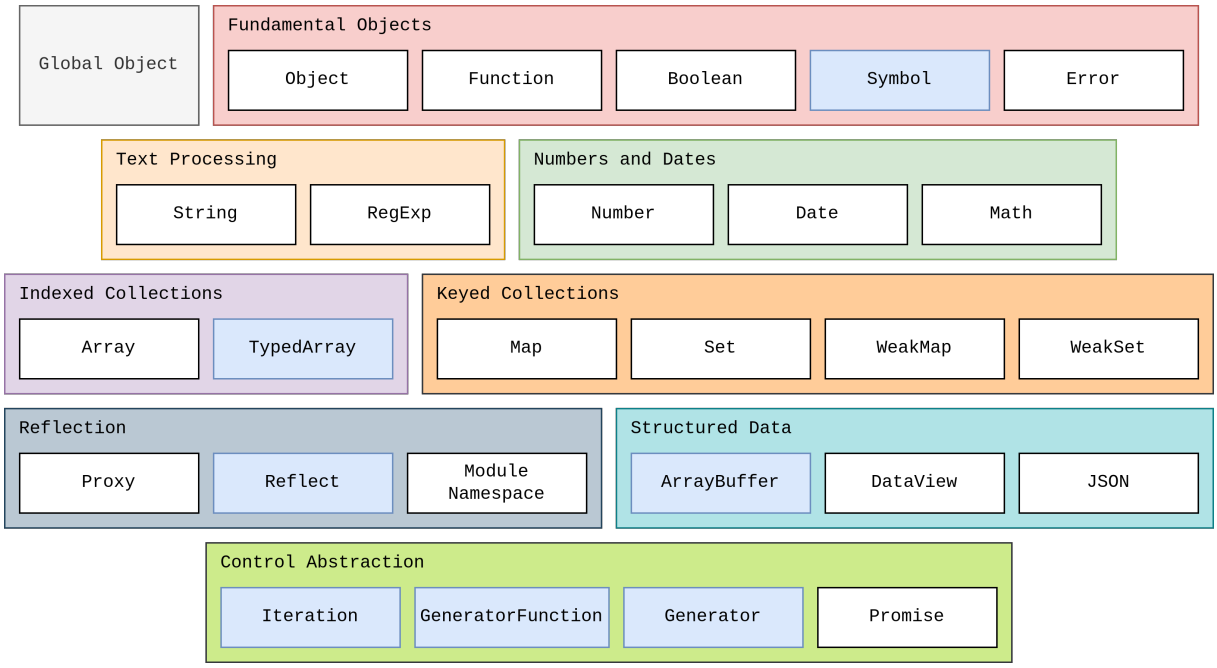


Figure 2.8: Overview of the built-in objects of ES6.

Built-in libraries can be split into two types based on the type of object that they make available through the global object:

1. Most of the built-in libraries, use a *constructor* `Function` object, which is an object that has a `[[Construct]]` internal method, to allow the creation of instances of that type. Instances created by the same constructor will share a common prototype object which will give them their functionality. Nonetheless, the constructor may still have its own methods for operations not adequate to the prototype object.

2. Some built-ins, like `JSON` and `Reflect`, use an ordinary object that does not permit the creation of instances of their types and expose their functionality only through their methods.

In Chapter 4, we look in more detail at our implementation of some of the built-in libraries highlighted before and therefore we now give a brief overview of the ones that are explored in the coming chapter:

- `ArrayBuffer` (Section 4.1) - the `ArrayBuffer` constructor produces objects that have an internal *byte array*, that can only be manipulated using `DataView` and `TypedArray` objects.

- `DataView` (Section 4.2) - the `DataView` constructor requires instances of `ArrayBuffer` to create its instances. A `DataView` object is essentially a wrapper that provides methods such as `getInt8` and `setFloat32` through its prototype, that will trigger the internal operations of the `ArrayBuffer` and allow the reading and writing of values on its internal byte array.
- `TypedArray` (Section 4.3) - similar to the `DataView` instances, `TypedArray` instances are essentially wrappers of `ArrayBuffer` instances. However, instead of the get and set methods, they provide an array-like interface to the byte-array. The `TypedArray` library actually populates the global object with multiple constructors, one for each possible element type (Int8Array, Int16Array, etc.), but their instances end up sharing the same prototype regardless. Each instance will only be able to write and read a certain amount of bytes per operation depending on the constructor used to create the instance. An instance created via the `Float32Array` constructor will read or write always 4 bytes interpreted as an IEEE 754-2008 [24] binary32 value. While one created via the `Int8Array` always reads or writes 1 byte interpreted as a signed integer.
- `Symbol` (Section 4.4) - the `Symbol` constructor produces `Symbol` values which are primitive values and can be used as keys for properties. Every `Symbol` value has an associated `string` value, however, even those who share the same `string` value must still be differentiable.
- `Proxy` (Section 4.5) - the `Proxy` constructor takes in two arguments: a target and a handler. The `Proxy` object produced will then act as the target in all regards. The only exception is when one of its internal methods is replaced by a method of the handler object. For example, if the internal `[[Get]]` method of the `Proxy` object is called, then it will search the handler object for a `"get"` method. If it finds the method, it calls it and uses it as if it was the internal method. If there is no method in the handler object, then the `[[Get]]` method of the target object is called instead. This allows the use of ECMAScript code that bypasses the target's internal method.

Chapter 3

Related Work

There have been numerous works on the development of reference interpreters for JavaScript. We will first look at the common trends amongst them, their coverage of the standard and their evaluation results. Then we discuss these works individually, looking at their most relevant design decisions and how they compare to the ECMA-SL project.

3.1 Overview

The need for a viable operation model of JavaScript was identified in 2008 with the work of Maffeis et al. [25]. Ever since, multiple other formal models of the JavaScript language have been written in various diverse languages such as Coq [26], K [27] and OCaml [23]. As these models appeared and took ideas from their predecessors, some concepts prevailed:

1. The formal model should be executable;

2. The formal model should pass the tests of the Test262 test suite;

3. The formal model should follow a line-by-line strategy in its implementation in order to be as identical as possible to the specification.

Consider Table 3.1 that shows how the various existing formal models of the language can be characterised with respect to the concepts stated above. Here we can see that more recent models tend to follow the standard line-by-line, which is a good indicator that this methodology is effective and a well-accepted approach to establishing trust in reference implementations. More recent models also tend to design their own DSLs (Domain-Specific Languages), like ECMA-SL, meant for implementing reference interpreters of the ECMAScript standard.

Although the wide adoption of these concepts results in more robust reference interpreters, there are still no models that offer significant support for the ECMAScript built-in libraries. However, their implementation is critical, for example, to reason about or test the implementation of the built-in libraries of JavaScript engines. Overall, since most ECMAScript programs use at least some of the libraries, not covering them greatly reduces the usefulness of the formal models of the language. Table 3.2 compares the various models with respect to the built-in libraries they implement, showing us that they are mostly ignored, while ECMARef6 is the most complete.

Reference interpreter	ES version	Exe.	# of tests passed	# of tests	Pass Rate	Implementation Language	L-B-L Strategy
S5 [15]	5	✓	8157	11275	72.35%	S5 Core Language	×
JSExplain [18]	5	✓	>5000	11275	44.35%	Subset of OCaml	✓
KJS [16]	5	✓	2782	11275	24.67%	K	×
JSRef [12]	5	✓	3749	11275	33.25%	Coq	✓
JS-2-JSIL [17]	5	✓	8797	11275	78.02%	JSIL	✓
ECMARef5 [19]	5	✓	9556	11275	84.75%	ECMA-SL	✓
ECMARef6	6	✓	18087	21662	83.50%	ECMA-SL	✓
JISET [28]	10	✓	18064	29878	60.46%	IR_{ES}	✓

Table 3.1: Reference interpreters and their adhesion the various strategies. (L-B-L signifies line-by-line)

Reference Intepreter	`Object`	`Function`	`Boolean`	`Symbol`	`Error`	`Number`	`Math`	`Date`	`String`	`RegExp`	`Array`	`JSON`
KJS	✓	✓	✓	×	✓	✳	×	×	✳	×	✳	×
JSRef	✓	✓	✳	×	×	✳	×	×	✳	×	×	×
JS-2-JSIL	✓	✓	✳	×	✳	✳	×	×	✳	×	×	×
ECMARef5	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	✓	✓
ECMARef6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 3.2: Built-in support by the other reference interpreters. (✳ signifies partial implementation)

3.2 Relevant Papers

3.2.1 Formalizations of the ECMAScript language's semantics

The formalization of the semantics of the ECMAScript language is a challenge that has been attempted by multiple research projects. In this subsection, we chronologically review some of the most relevant of these projects.

Maffeis et al. [25] were the first to see the need in designing an operational semantics for ECMAScript that followed the standard truthfully. Their goal was to reason about the security of JavaScript applications in the browser, more specifically, what regions of the heap a certain program could or could not reach. For example, if a secure web-page was loaded and it contained an insecure embedded advertisement could it reach the heap region allocated for the secure program? They looked to answer this question through what they coined the reachability theorem. Their operational semantics were left non-mechanised, described by semantic rules in a long text document.

Guha et al. (2010) [13] present λ_{JS}, a mechanised formalization of the semantics of the ECMAScript standard done by reducing it to a core calculus in the form of a small-step operational semantics. Their semantics cover some of ES3's most fundamental features: prototype-inheritance, extensible objects and dynamic function calls. However, features like some of the built-in libraries and the `eval` expression, that performs dynamic code execution, were not included. They put a great emphasis on the desugaring of the language's syntax, as that not only simplifies λ_{JS}, but also makes some cases where the behavior of a piece of JavaScript code is unexpected, even thought the syntax is familiar, much more apparent. An example and a common source of errors in JavaScript is the **this** keyword, which is an implicit parameter of every function and whose value is sometimes not very intuitive. As such, in λ_{JS}, **this** becomes an

explicit parameter and function calls must explicitly supply the **this** argument. The idea of desugaring is also present in the ECMA-SL project, although not in the form of desugared JavaScript but in the form of ECMA-SLCore; they still share the same goal. The correctness of the semantics was verified utilizing the Mozzila JavaScript test suite. For that, they first built a λ_{JS} interpreter and then for the applicable test cases of the test suite, desugared the original JavaScript and ran the test with the interpreter, comparing the output of their interpreter with the output of running the test on V8, SpiderMonkey and Rhino and verifying that the outputs were identical.

Politz et al. (2012) presented S5 [15] which builds upon λ_{JS} moving it from the 3rd version of the standard to the 5th and adding the semantics of accessors (getters and setters) and the eval operator. This project's reference interpreter was built with the *S5 core language*, being one of the first projects to design its own implementation language. Their reference interpreter includes some of the built-in objects that make up the ECMAScript 5 standard library. Like in λ_{JS}, there is a desugaring process, although in the S5 project it is more akin to compilation, that converts any non-implemented source-language features into ones present in the S5 core language, allowing for the execution of any ES5 program. To mechanise the semantics, a S5 core language interpreter was built in the OCaml language which makes it possible to test the reference interpreter against the Test262 conformance suite of which they passed 8157 of 11606 or 70% of tests, with the lowest performance being in the built-in objects section, which was expected given that they were not all implemented.

Park et al. (2015) developed KJS [16], the most complete formalization of JavaScript when it was released. It was done using the $\mathbb{K}$ framework [27], a powerful framework designed specifically for defining language semantics that with the definition of a language's syntax and semantics, can generate a parser, an interpreter and formal analysis tools. With KJS, the authors were able to verify that the ECMAScript 5.1 conformance test suite did not completely cover all the behaviors defined in the standard and wrote those missing tests, coming to the conclusion that most production JavaScript engines and other semantic formalizations failed these new tests. KJS and Chrome's V8 engine are the only implementations of JavaScript that completely pass all the core language tests, even among other browser engines and reference interpreters. When it comes to the language's built-in libraries, they considered that they could be built using traditional JavaScript as they had guaranteed the core semantics to be correct. As such, there was not a big effort put into implementing these libraries as it was deemed outside the scope of the project. Even though it seems like the $\mathbb{K}$ framework and KJS would be a great fit for a project like ours, it is hard to work with the framework and its language is very distant from the standard's pseudo-code, since the $\mathbb{K}$ framework was not built specifically for the ECMAScript language, which is an important aspect of the ECMA-SL approach.

3.2.2 Acquiring trust through closeness

As more reference interpreters were developed, a common method of generating trust in the implementation was to determine its closeness to the English and pseudo-code of the textual specification of the standard, mostly by using a line-by-line strategy. In this subsection, we review some of the most relevant projects in this aspect.

Bodin et al. (2014) developed JSCert [12] and JSRef for the 5th edition of the ECMAScript standard. JSCert is a mechanised specification of ES5 that uses pretty-big-step semantics and is written in the interactive proof assistant Coq. One of the goals of the JSCert was to make the argument that JSCert

is an accurate formulation of the ES5 specification. With that goal in mind, JSCert was designed to follow the ECMAScript standard as much possible, which makes it specially relevant to ECMA-SL project. Alongside JSCert, they developed JSRef a reference interpreter also written in Coq. They used an extraction technique on JSRef to generate executable OCaml code, that could be used to execute the reference interpreter. Given that they were both written in Coq, it was easy for them to verify that JSRef was correct in respect to JSCert. Given that JSCert was designed to follow the standard as closely as possible, this correctness they gave them confidence that the reference interpreter's behavior would match the standard's specification. The possession of an executable reference interpreter means that it was possible to test it against the Test262 test suite, where they passed 1796 of the 2782 of the tests that relate to chapters 8-14 of the ES5 standard, which excludes the built-in libraries. In order to measure trust in JSCert, the English prose of the standard is put side-by-side with the formal rules and their closeness if "eyeballed". In the ECMA-SL project, we also aim to measure the similarity of our mechanised semantics to the standard's prose, however our method involves the use of the HTML2ECMA-SL tool and text similarity metrics to compare the generated document with the official one, which we consider an improvement on the methodology as it is deterministic and completely objective. Regardless, the authors were able to find several bugs present in all browser implementations, the ECMA standards and the test suite. A year later, Gardner et al. [29] extended JSRef with an implementation of ES5 Arrays, using Google's own implementation in the V8 JavaScript engine [5]. They also assessed other improvements made to JSCert, such as correcting some misinterpretations of the ES5 standard, and how they and the implementation of the Array library impacted their performance against Test262.

Charguéraud et al. (2018) presented *JSExplain* [18], a reference interpreter for the 5th edition of the ECMAScript standard. This interpreter was built using a subset of the OCaml programming language in a purely functional programming style. However, they support the compilation of the OCaml source code to two different languages: JavaScript, so that the reference interpreter can be executed in a browser; and Pseudo-JS which is a custom language derived from the ECMAScript standard's pseudo-code, akin to ECMA-SL, to allow JavaScript developers to better understand the reference interpreter even if they are unfamiliar with OCaml, as this was one of the many requests the TC39 committee had for reference interpreter developers. The JSExplain reference interpreter passes over 5000 tests of the Test262 test suite and also has debugging support, more concretely, double-debugging, allowing developers to not only examine the program's state in step-by-step execution, but also to examine the reference interpreters internal state.

Jihyeok Park et al. presented JISET [28], a JavaScript IR-based semantics extraction toolchain. From the specification of ECMAScript, it can automatically synthesize parsers and AST-IR translators which allowed the authors to generate a partial implementation of ES10. These tools allow the compilation of ECMAScript programs into an intermediate language that has its own interpreter. The authors were able to synthesize parsers and compile the algorithm steps of the 10th, 9th, 8th and 7th versions of the standard to great effect, passing all 18,064 tests relative to the core of the language. However, like most of the literature, they ignore the built-in libraries, filtering out 6,532 tests related to the behaviour of these libraries. In contrast, we believe the built-in libraries to be a fundamental piece of the specification.

Summary In this section we have seen that there have been multiple attempts to formalize the semantics of the ECMAScript standard. Over the years, we see a greater emphasis on using the Test262 test suite to gain trust in the validity of reference interpreters. For this reason, there is also a great incentive to have them be executable as well. Another trust measure, repeatedly mentioned is the verbal closeness

between the reference interpreters specification and the standard, with the line-by-line approach being a common strategy to obtain it. However, most of the works focus on the core semantics of the language and do not implement the language's standard library which makes their coverage of the Test262 test suite low.

Chapter 4

Reference Implementation of the ES6 Built-in Libraries

This chapter describes the main contribution to the main goal of this thesis, which was the development of a reference implementation of the ES6 built-in libraries, in this case in the context of ECMARef6. This implementation was done in the scope of the ECMA-SL project which involves multiple students. In particular, the development of the built-in libraries had contributions from 6 students[1] coordinated by me. Besides coordinating the implementation effort, I was personally responsible for the implementation of the `TypedArray` and `Symbol` built-ins which are described in this chapter. The implementation of the `ArrayBuffer` and `DataView` libraries were done in collaboration with Nuno Policarpo, while the implementation of the `Proxy` library was done in collaboration with Tomás Tavares.

For clarity, we structure the account of our built-in implementations in the following way: first we describe how the built-in is used in practice by appealing to simple ECMAScript code snippets; then we give a high-level description of the official built-in's specification; next, we describe our ECMA-SL implementation of the built-in, focusing on its connection to the standard; and, we conclude with a small section outlining the implementation highlights.

4.1 ArrayBuffer

In this section we will discuss the standard's definition of `ArrayBuffer` objects and our implementation of them. In order to allow for the implementation of the `ArrayBuffer` built-in library we had to extend the ECMA-SL language itself. The description of this extension is given in Subsection 4.1.4.

`ArrayBuffer` objects represent what many other languages would call a `byte array`. They serve as an interface to interact with instances of the `Data Block` type. An ECMAScript `Data Block` can be thought of as a simple array of bytes. However, this interface is not available in the ECMAScript language and must be accessed via internal operations of the standard. These operations can be triggered in the ECMAScript language only through the use of `TypedArray` and `DataView` objects.

4.1.1 Examples

In this subsection, we will explain how `ArrayBuffer` objects work in practice by appealing to several ECMAScript code examples that interact with buffers either via the `DataView` interface or the `TypedArray` interface. These two interfaces will themselves be revisited in two later sections.

[1] David Belchior, Nuno Policarpo, Leonor Barreiros, João Silveira, Manuel Costa, Tomás Tavares

Consider the code snippet given in Listing 4.1. This program uses a `DataView` object to interact with an `ArrayBuffer`. To begin, an `ArrayBuffer` object is created using the `ArrayBuffer` constructor. Since the argument passed was 4, a `Data Block` of 4 bytes will be created as well. Then, the `DataView` constructor is called with the `ArrayBuffer` instance to create the `DataView` object through which we will interact with the `ArrayBuffer`. Finally, we use the `setInt32` method of the `DataView` instance to manipulate the bytes of the `Data Block`. In this specific case, since an element of the `Int32` type takes up 4 bytes, the value 9999 will be encoded into a 4 byte sequence as a signed integer. The byte sequence will be big-endian, since we omitted the last argument when calling the `setInt32` method. Passing the value `true` would change the endianness of the byte sequence.

Listing 4.1: Use of an `ArrayBuffer` object via a `DataView` object.

```
1 var ab = new ArrayBuffer(4);
2 var dv = new DataView(ab);
3 dv.setInt32(0, 9999);
```

In contrast to the example given in Listing 4.1, the code snippet given in Listing 4.2 interacts with an `ArrayBuffer` via the `TypedArray` interface. The instantiation of the objects occurs in a similar manner to the previous example except that we now use the `Int32Array` constructor instead of the `DataView` constructor. There is also a difference in the way that we use the wrapper. In the case of the `Int32Array`, we use it as an `Array`, as opposed to calling the `setInt32` method like before. It would not be wrong to think of an instance of `Int32Array` as a `DataView` wrapper that only uses the methods `getInt32` and `setInt32` of the `DataView` object. It is important to note that the APIs of these objects are not interchangeable, as `dv.getInt32(1)` is different from `ta[1]`. The expression `dv.getInt32(0)` fetches bytes 0 to 3 while `dv.getInt32(1)` fetches bytes 1 to 4. However, `ta[0]` fetches bytes 0 to 3 and `ta[1]` fetches bytes 4 to 7. The `DataView` objects allows us to read the bytes in a less structured way, while the `TypedArray` instance will force the element size to be respected. Something else to note is that it is not possible to specify endianness when using any of the `TypedArray` instances.

Listing 4.2: Use of an `ArrayBuffer` object via a `Int32Array` object.

```
1 var ab = new ArrayBuffer(4);
2 var ta = new Int32Array(ab);
3 ta[0] = 9999;
```

4.1.2 ECMAScript Specification

We will now give a detailed account of how the ECMAScript standard describes `ArrayBuffer` instances, as well as the major operations associated with them. This will give context to the next subsection where we will discuss our own implementation.

The `ArrayBuffer` built-in library makes available a constructor function in the global object. Besides the `ArrayBuffer` constructor, the only method exposed by the `ArrayBuffer` API is the method `ArrayBuffer.isView`, which is used to determine if an object passed to it is an instance of one of the common `ArrayBuffer` wrappers like `DataView` and `TypedArray`. It is not surprising that the `ArrayBuffer` interface only exposes a single method besides the constructor since interaction with `ArrayBuffer` objects always happen through either the `TypedArray` or the `DataView` interfaces, which do expose a substantially larger number of methods.

The instances created via the constructor are ordinary objects, meaning that their internal methods have the default implementation, but they do have two additional internal properties:

1. `[[ArrayBufferData]]` - a reference to the `Data Block` allocated to the `ArrayBuffer` object;
2. `[[ArrayBufferByteLength]]` - the number of bytes in the `ArrayBuffer`.

Data Blocks We have not yet given a proper description of `Data Block` values. However, they are rather simple as they are not objects, meaning that do not have any properties or fields, they are a primitive value which corresponds to a byte sequence. The fact that its metadata is present in the `ArrayBuffer` instance in the `[[ArrayBufferByteLength]]` internal property demonstrates that. In Figure 4.1, we can see an instantiation of an `ArrayBuffer` object whose internal slot `[[ArrayBufferData]]` holds the reference to a `Data Block` and `[[ArrayBufferByteLength]]` tells us that the `Data Block` is of length 6. Upon instantiation, the allocated `Data Block` of objects created via the `ArrayBuffer` constructor, have all their bytes set to `0x00`[2].

```
1 var ab = new ArrayBuffer(6);
```

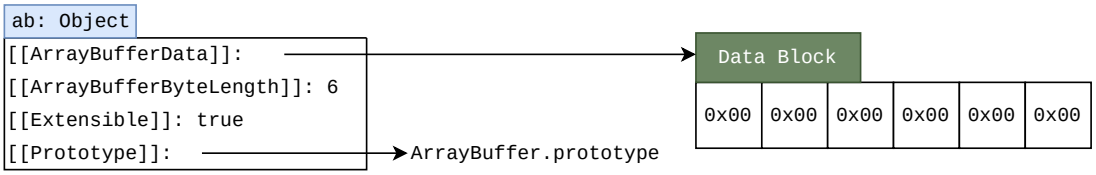


Figure 4.1: An `ArrayBuffer` object and its Data Block.

ArrayBuffer.prototype The prototype of an object is the value referenced by its `[[Prototype]]` internal property. Usually, this will be an ordinary object with a set of methods that are shared through prototype-inheritance. This object is referenced by all `ArrayBuffer` objects' `[[Prototype]]` internal property and also by the `prototype` of the `ArrayBuffer` constructor. Figure 4.2 illustrates this concept which is repeated all over the standard and is how instances can share the same properties without duplication. In this example, we have once again omitted all the internal methods from the diagram for the sake of clarity, but we now have objects with named properties, like the `ArrayBuffer` constructor in yellow and the `ArrayBuffer.prototype` object in the right, that are separated from their internal properties with a line to make the distinction between the two types of properties as clear as possible. From the example we can also see that the prototype object has only two properties:

```
1 var ab1 = new ArrayBuffer(6);
2 var ab2 = new ArrayBuffer(4);
```

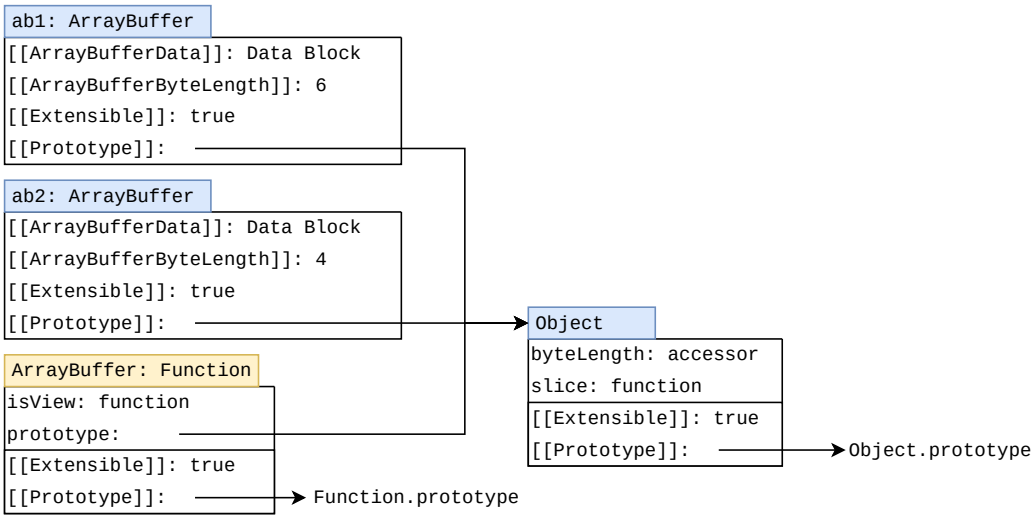


Figure 4.2: Illustration of the sharing of a prototype object by a constructor and its instances.

[2]Byte values will be represented using the hexadecimal format.

1. `byteLength` - this is an accessor property that will retrieve the value of the `ArrayBuffer`'s internal `[[ArrayBufferByteLength]]` property. It has no `[[Set]]` value so it is a read-only property.

2. `slice` - this is a data property that references a function object that was omitted from the example to keep it simple. This method will create a new `ArrayBuffer` instance whose `Data Block` will be a copy of a segment of the original one.

This means there is a lack of a method `ab.setByte(index, value)` or custom `[[Get]]` and `[[Set]]` internal methods to allow the manipulation of the bytes. As mentioned before, in order to do so, one has to create either a `DataView` or `TypedArray` object. However, unfamiliar JavaScript developers may not know that and with a requirement to do byte-level operations might think that an `ArrayBuffer` object is the best fit. Given what was discussed before, that would not be an inappropriate solution. To demonstrate this, consider the example given in Figure 4.3. This code snippet uses an `ArrayBuffer` as a typical `Array` producing unwanted results. Judging by lines 3 and 4, one might think that the first byte of the `Data Block` associated with the object was set to `0x01`. However, looking at the object diagram we can see that instead a data property descriptor was associated with the key `0` and that the `Data Block` remains with all its bytes equal to `0x00`. We can also verify this programmatically, using the instructions from line 6 onward, where we duplicate the `ArrayBuffer` object using the `slice` method and see now that accessing the value associated with the key `0` returns `undefined` instead of `1`.

```
1 var ab = new ArrayBuffer(4);
2
3 ab[0] = 1;
4 ab[0] === 1; // true
5
6 var ab2 = ab.slice(0, 4);
7
8 ab2[0] === 1; // false
9 ab2[0] === undefined; // true
```

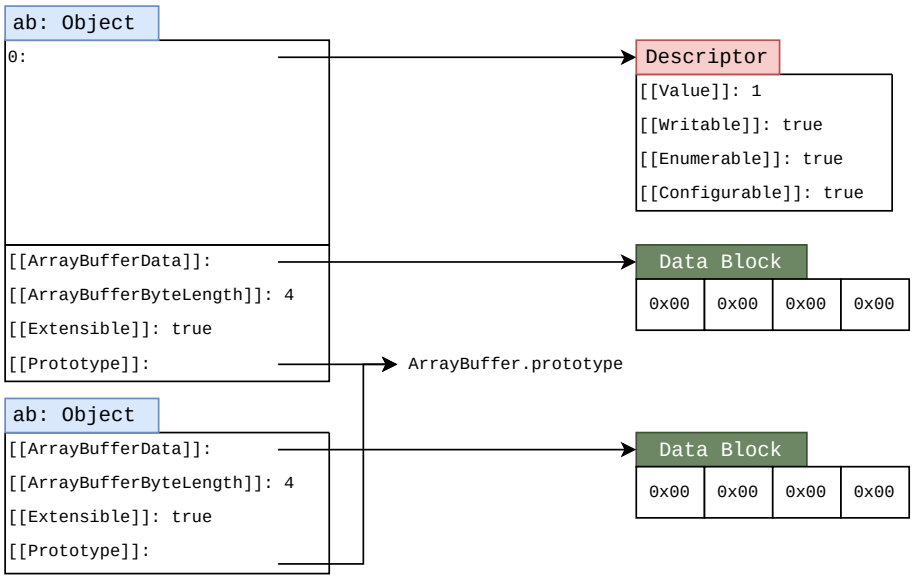


Figure 4.3: Showcase of incorrect use of an `ArrayBuffer` object.

So far, it may seem that availability of the `ArrayBuffer` constructor to JavaScript developers is redundant as there is no way of interacting with the objects created by it besides the `slice` method defined in the `ArrayBuffer.prototype` object and therefore only the aforementioned `DataView` and `TypedArray` built-in libraries should be available and they should internally create their `ArrayBuffer` instances. However, there is one use case where its availability becomes relevant. In Figure 4.4, we can see that

the ArrayBuffer object created in line 1, is used as an argument for the instantiations in lines 3 and 4, meaning that the objects instantiated will share them same buffer. In line 6, we use the setInt32 method of the DataView object to write the value 9999 onto the first 4 bytes of the buffer and lines 8 and 9, serve to demonstrate that the write operation was successful and that the buffer is indeed being shared by those two objects.

```
1  var buffer = new ArrayBuffer(4); // 32 bits
2
3  var dv = new DataView(buffer);
4  var ta = new Int32Array(buffer);
5
6  dv.setInt32(0, 9999);
7
8  dv.getInt32(0) === 9999; // true
9  ta[0] === 9999; // true
10
11 ta[1] = 44;
12
13 dv.getInt32(1) === 44; // true
14 ta[1] === 44; // true
```

Figure 4.4: Example of an ArrayBuffer object being shared.

Even though the interaction with ArrayBuffer objects must be mediated by the TypedArray and DataView built-in objects, those objects make use of the internal operations defined as part of the ArrayBuffer library. These internal operations include:

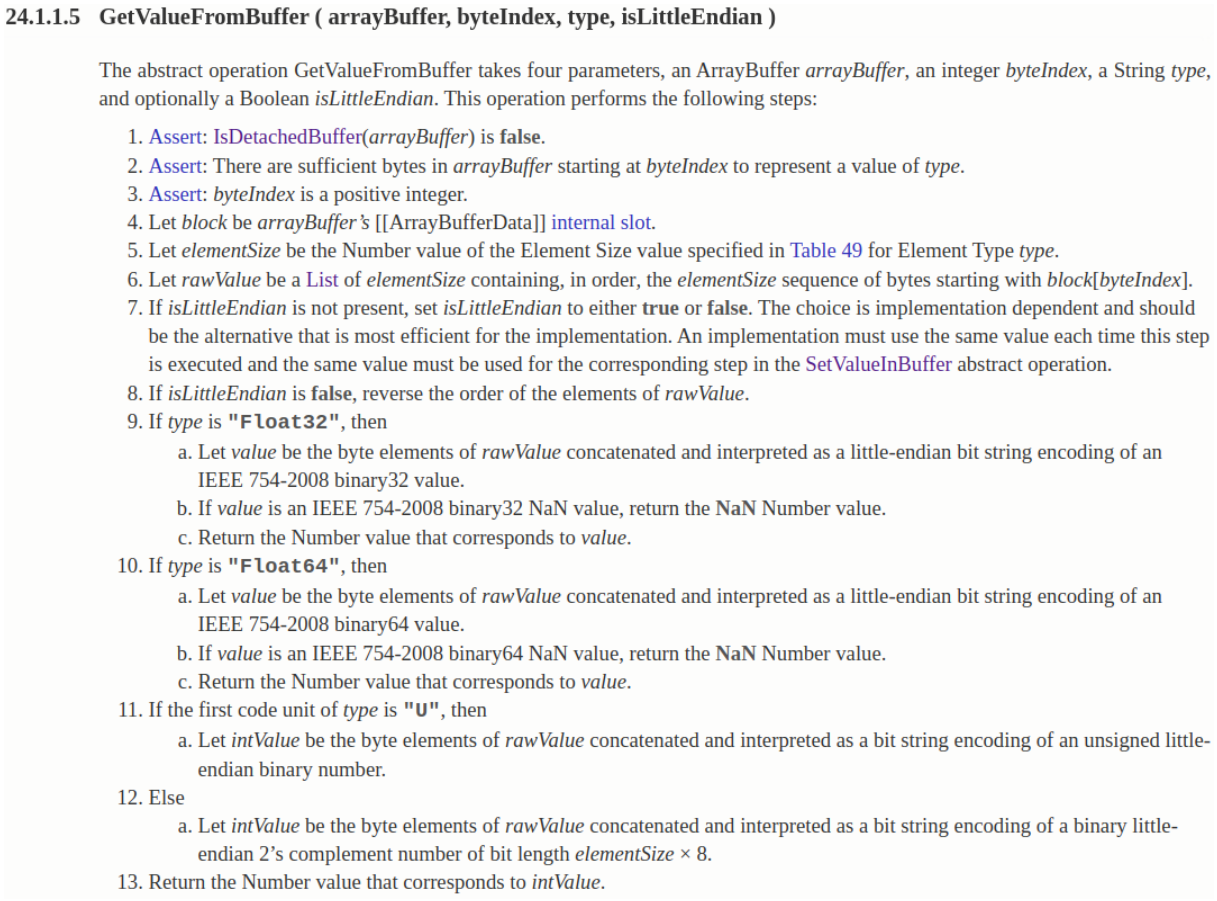
1. AllocateArrayBuffer - this operation will create an ArrayBuffer instance and a Data Block of the length passed as argument. This operation is used in the ArrayBuffer and TypedArray constructors, as well as in some other methods of TypedArray.prototype;

2. GetValueFromBuffer - this operation retrieves a value from an ArrayBuffer instance. Its parameters are an ArrayBuffer instance, a byte index, a type of value (Int8, Float32, etc.) and a flag to symbolize little-endianness. This operation derives from the type parameter the number of bytes that need to be extracted from the Data Block of the ArrayBuffer. It then uses the little-endian boolean flag to determine if it is necessary to reorder the bytes. Finally, it interprets the byte sequence according to the type parameter and return a value of the primitive number datatype;

3. SetValueInBuffer - this operation writes a value in an ArrayBuffer instance. Its parameters are an ArrayBuffer instance, a byte index, a type of value (Int8, Float32, etc.), the value to write and a flag to symbolize little-endianness. This operation derives from the type parameter the number of bytes that will be written in the Data Block of the ArrayBuffer. Next, it converts the value parameter from a number to a list of bytes, according to the type parameter. It then uses the little-endian boolean flag to determine if it is necessary to reorder the bytes. Finally, it copies the values from the list of bytes to the Data Block.

Consider the excerpt of the ECMAScript standard in Figure 4.5, that contains the standard's description of GetValueFromBuffer internal operation. Instructions 1 to 3 perform some input sanitation and instructions 4 to 8 retrieve the necessary bytes from the ArrayBuffer's Data Blocks, ordering them according to the specified endianness. The 9.a, 10.a, 11.a and 12.a instructions convert the obtained sequence of bytes to a number of the corresponding type (e.g. 9.a converts the obtained sequence of bytes to a "Float32"). After obtaining the number of the appropriate type, all the branches proceed to convert the value interpreted from the byte sequence to a Number value. However, if type is "Float64" no conversion occurs. This is because every Number value in ECMAScript is a floating-point value, more specifically a "primitive value corresponding to a double-precision 64-bit binary format IEEE 754-2008

value", which is why no conversion is necessary. If type is `"Int8"`, for instance, if the bytes read are interpreted as the value 50, this operation will actually return the value 50.0.



24.1.1.5 GetValueFromBuffer (arrayBuffer, byteIndex, type, isLittleEndian)

The abstract operation GetValueFromBuffer takes four parameters, an ArrayBuffer *arrayBuffer*, an integer *byteIndex*, a String *type*, and optionally a Boolean *isLittleEndian*. This operation performs the following steps:

1. Assert: IsDetachedBuffer(*arrayBuffer*) is **false**.
2. Assert: There are sufficient bytes in *arrayBuffer* starting at *byteIndex* to represent a value of *type*.
3. Assert: *byteIndex* is a positive integer.
4. Let *block* be *arrayBuffer*'s [[ArrayBufferData]] internal slot.
5. Let *elementSize* be the Number value of the Element Size value specified in Table 49 for Element Type *type*.
6. Let *rawValue* be a List of *elementSize* containing, in order, the *elementSize* sequence of bytes starting with *block*[*byteIndex*].
7. If *isLittleEndian* is not present, set *isLittleEndian* to either **true** or **false**. The choice is implementation dependent and should be the alternative that is most efficient for the implementation. An implementation must use the same value each time this step is executed and the same value must be used for the corresponding step in the SetValueInBuffer abstract operation.
8. If *isLittleEndian* is **false**, reverse the order of the elements of *rawValue*.
9. If *type* is **"Float32"**, then
   a. Let *value* be the byte elements of *rawValue* concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary32 value.
   b. If *value* is an IEEE 754-2008 binary32 NaN value, return the **NaN** Number value.
   c. Return the Number value that corresponds to *value*.
10. If *type* is **"Float64"**, then
    a. Let *value* be the byte elements of *rawValue* concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary64 value.
    b. If *value* is an IEEE 754-2008 binary64 NaN value, return the **NaN** Number value.
    c. Return the Number value that corresponds to *value*.
11. If the first code unit of *type* is **"U"**, then
    a. Let *intValue* be the byte elements of *rawValue* concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.
12. Else
    a. Let *intValue* be the byte elements of *rawValue* concatenated and interpreted as a bit string encoding of a binary little-endian 2's complement number of bit length *elementSize* × 8.
13. Return the Number value that corresponds to *intValue*.

Figure 4.5: Description of the `GetValueFromBuffer` abstract operation in the ES6 standard.

4.1.3 ECMA-SL Implementation

We are now at the position to fully explain our implementation of the `ArrayBuffer` library. In order to explain the representation of `ArrayBuffer` objects in ECMA-SL, we must first give a brief description of how all ECMAScript objects are represented in ECMA-SL. Following the description of the representation of ECMAScript objects, we will then focus on the representation of `ArrayBuffer` objects and `Data Block` values. We then describe how ECMAScript functions and their logic are represented in ECMA-SL, using the `ArrayBuffer` constructor and its `isView` method as examples.

ECMAScript Objects We represent ECMAScript objects using ECMA-SL objects, with a very similar schema to the ones used in the diagram examples. The main difference is that we map every ECMAScript object to two separate ECMA-SL objects: one keeping the internal properties and one keeping the named properties (the properties defined by the programmer). To better understand this encoding, consider the example given in Figure 4.6. The execution of the code-snippet in the figure generates the ECMAScript object given in Figure 2.5, which we represent in ECMA-SL as illustrated in Figure 4.6. Compared to the previous diagram we can see that internal properties have lost the double square bracket nomenclature and that the separation between named and internal properties is much more explicit. Named properties now reside in an object that is referenced by the `JSProperties` internal property and still reference `Property Descriptors` as before. The `[[Extensible]]` and `[[Prototype]]`

internal properties still reside inside the parent object but no longer have the double square brackets. Internal methods are also still inside the parent object, buy they hold `string` values now that contain the name of ECMA-SL functions that implement their behavior. In the next paragraph we discuss how ECMAScript functions are described in ECMA-SL and how these `string` values come into play.

```
1 const dog = {
2     race: "Beagle"
3 };
```

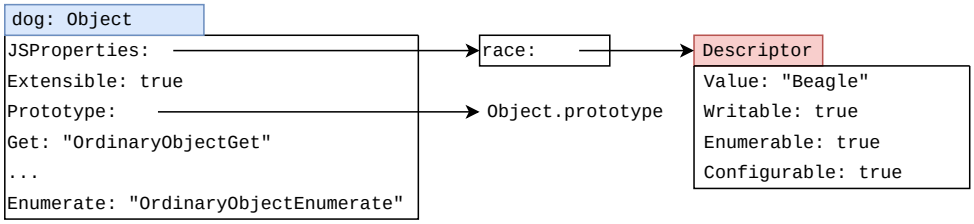


Figure 4.6: Internal representation of an object in the reference interpreter.

ArrayBuffer Objects and Data Block Values Now that we have looked at how objects are represented in our implementation, let us look at `ArrayBuffer` objects specifically. Consider Figure 4.7 that describes how `ArrayBuffer` objects are represented in ECMA-SL. Here we can see that the `[[ArrayBufferData]]` and `[[ArrayBufferByteLength]]` internal properties are present without the double square brackets once again and that the `JSProperties` object is empty, as `ArrayBuffer` instances have no named properties. The most noteworthy difference is the representation of the `Data Block`, held by the `ArrayBufferData` property which is now an array of bytes. This array is an ECMA-SL array that has nothing to do with ECMAScript arrays.

```
1 var ab = new ArrayBuffer(6);
```

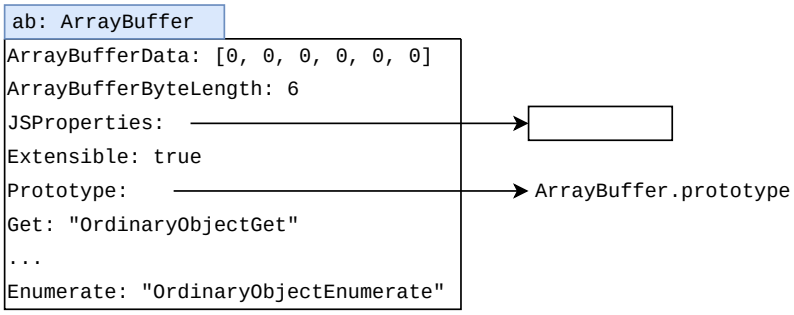


Figure 4.7: Internal representation of an `ArrayBuffer` object in the reference interpreter.

ECMAScript Functions and the ArrayBuffer Constructor Another aspect of the language whose representation is worth describing in more detail are the function objects. ECMAScript function objects are ordinary objects with one extra internal method `[[Call]]` that will make it execute its code. Function objects that are constructors also have a `[[Construct]]` internal method that is called instead of the `[[Call]]` method in `new` expressions. Their code is stored in an internal property called `[[ECMAScriptCode]]`. Since ECMAScript functions are also objects, they are represented in ECMA-SL through ECMA-SL objects. These objects have the extra internal methods `Call` and `Construct`, as well as the `Code` property that holds the code of the function. The disparity in the property name is a fragment from ES5 and ECMARef5 where function objects used the `[[Code]]` property instead. In

our implementation, this property holds the name of the ECMA-SL function that captures its respective logic. Consider Figure 4.8 where we illustrate the internal representation of the `ArrayBuffer` constructor. In the constructor object, we can see that the `Call`, `Construct` and `Code` properties are all present. We can also see that the `JSProperties` object has properties that point to its `prototype` property and `isView` method. The `isView` descriptor points to another function object that also has the `Call` and `Code` properties, with the value of the `Code` property being `"ArrayBufferIsView"`.

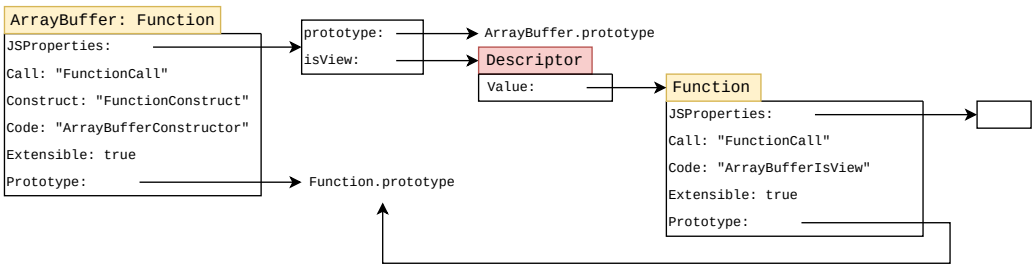


Figure 4.8: Internal representation of the `ArrayBuffer` constructor and its `isView` method in the reference interpreter.

Consider now Figure 4.9 where we can see the comparison between the `ArrayBufferIsView` ECMA-SL function mentioned in the `Code` property of the `ArrayBuffer.isView` method and its ES6 counterpart. Starting from the top, we can spot a few differences:

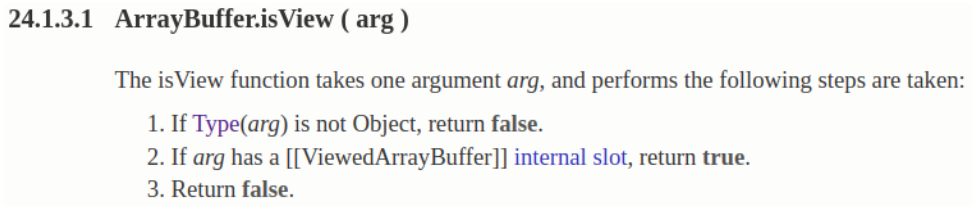
1. Starting with the name, we can see that the naming in the standard makes it obvious that `isView` is a method of `ArrayBuffer` being using dot notation. In ECMA-SL, function names cannot have dots in them and so it is impossible to use the same nomenclature.

2. When it comes to the parameters, we can also see that the ECMA-SL version has many more than its counterpart. This is because all ECMA-SL functions that can be called through ECMAScript code, like the methods of the built-ins objects, share the same parameters. In the scope of this section, the only that are significant are the `this` and `args` parameters. The `this` parameter will be a reference to the object that holds the method, in this case the `ArrayBuffer` constructor. The `args` parameter is a list of all the arguments used in the function call. In fact, the first line of the ECMA-SL function body retrieves the `arg` argument from the list.

Contrary to the function signature, the body of the ECMAScript and ECMA-SL functions holds many similarities. One can easily check that the ECMA-SL code mimics the pseudo-code of the standard almost exactly, as we had discussed in Section 2.1 regarding the ECMA-SL project.

This kind of structuring of the objects and their properties and methods is identical in other the built-in objects, so this subject will not be touched upon in further explanations unless it is necessary.

Internal Operations Finally, we come to the implementation of the internal operations that dictate the core of the `ArrayBuffer` behavior. We will once again be comparing the standard's description with the ECMA-SL implementation with an example focused on the `GetValueFromBuffer` operation showed earlier. This comparison is given in Figure 4.10 where the ECMA-SL excerpt does not have the initial instructions of the operation as, once again, they are matched with equivalent ECMA-SL statements with no nuances. However, we can see that instructions 9.a, 10.a, 11.a and 12.a are not perfectly matched. The reason for that is that instruction is not typical and cannot be matched by a standard ECMA-SL one-liner. It was actually necessary to extend the ECMA-SL language with these new operators (`float32_from_le_bytes`, `float64_from_le_bytes`, `uint_from_le_bytes`, `int_from_le_bytes`) in order to achieve this functionality, as previously to the introduction of the `ArrayBuffer` library, there



24.1.3.1 ArrayBuffer.isView (arg)

The isView function takes one argument *arg*, and performs the following steps are taken:

1. If Type(*arg*) is not Object, return **false**.
2. If *arg* has a [[ViewedArrayBuffer]] internal slot, return **true**.
3. Return **false**.

(a) Standard description of the `ArrayBuffer.isView` method.

```
function ArrayBufferIsView(global, this, NewTarget, strict, args) {
  arg := getOptionalParam(args, 0);
  /* 1. If Type(arg) is not Object, */
  if (!(Type(arg) = "Object")) {
    /* return false */
    return false
  };
  /* 2. If arg has a [[ViewedArrayBuffer]] internal slot, */
  if ("ViewedArrayBuffer" in_obj arg) {
    /* return true */
    return true
  };
  /* 3. Return false. */
  return false
};
```

(b) Implementation of the `ArrayBuffer.isView` method in the reference interpreter.

Figure 4.9: Comparison between the standard description of the `ArrayBuffer.isView` method its implementation in the reference interpreter.

was no need for byte-level operations in ECMA-SL. Remember also, that we mentioned that the values needed to be converted to floating-point numbers since `number` values are all 64-bit floating-point values. The reason why there are no explicit conversions in our implementation is because they happen inside the newly added operators. To avoid the possibility of a stray integer value in the interpreter, it was deemed safer that the operators do the conversion themselves. The semantics and implementation of these new operators are discussed further in the next subsection.

4.1.4 Extending ECMA-SL

In this subsection we introduce the extensions that were made to the ECMA-SL language in order to support the `ArrayBuffer` built-in library. This extension was made through two aspects: (1) arrays were introduced alongside operators to create, read and write values to them; (2) the byte type and byte-level operators were introduced that allow the conversion of a number value to a byte-array and vice-versa.

Before the introduction of the `ArrayBuffer` built-in library and the `Data Block` type, there were no ECMAScript datatypes that required ECMA-SL arrays for their representation. Although ECMA-SL lists already existed, since they are immutable they would not be an adequate representation of the `Data Block` type, as changing the value of one of the bytes would require an entirely new list to be created. The operators made available to the array type in ECMA-SL can be seen in Table 4.1 alongside their parameters and semantics.

Alongside the array type and its operators, it was also necessary to create the *byte* type and operators that could turn an ECMA-SL floating-point value into an ECMA-SL array of bytes and vice-versa. These operators only take in and return float values because in ECMAScript all `Number` values are floats. Trickling down this concept into the ECMA-SL language itself, means that there is a lesser likelihood of existing stray integer values. A list of these operators, their parameters and semantics can be seen in Table 4.2. Note that although there are operators for big-endian (`be_bytes` suffix) and little-endian (`le_bytes` suffix) byte ordering, in the implementation of the reference interpreter it is only necessary to use operators of one kind, regardless of what the endianness of the byte sequence is supposed to

9. If *type* is **"Float32"**, then
    a. Let *value* be the byte elements of *rawValue* concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary32 value.
    b. If *value* is an IEEE 754-2008 binary32 NaN value, return the **NaN** Number value.
    c. Return the Number value that corresponds to *value*.
10. If *type* is **"Float64"**, then
    a. Let *value* be the byte elements of *rawValue* concatenated and interpreted as a little-endian bit string encoding of an IEEE 754-2008 binary64 value.
    b. If *value* is an IEEE 754-2008 binary64 NaN value, return the **NaN** Number value.
    c. Return the Number value that corresponds to *value*.
11. If the first code unit of *type* is **"U"**, then
    a. Let *intValue* be the byte elements of *rawValue* concatenated and interpreted as a bit string encoding of an unsigned little-endian binary number.
12. Else
    a. Let *intValue* be the byte elements of *rawValue* concatenated and interpreted as a bit string encoding of a binary little-endian 2's complement number of bit length *elementSize* × 8.
13. Return the Number value that corresponds to *intValue*.

(a) Final instructions of the standard's description of the `GetValueFromBuffer` operation.

```
function GetValueFromBuffer(arrayBuffer, byteIndex, type, isLittleEndian) {
  /*
    ...
  */
  if (type = "Float32") {
    value := float32_from_le_bytes(rawValue);
    if (is_NaN value) {
      return NaN
    };
    return value
  };
  if (type = "Float64") {
    value := float64_from_le_bytes(rawValue);
    if (is_NaN value) {
      return NaN
    };
    return value
  };
  if (s_nth(type, 0) = "U") {
      intValue := uint_from_le_bytes(rawValue, elementSize)
  } else  {
      intValue := int_from_le_bytes(rawValue, elementSize)
  };
  return intValue
};
```

(b) Implementation of the `GetValueFromBuffer` operation in the reference interpreter.

Figure 4.10: Comparison between the standard description of the `GetValueFromBuffer` operation and its implementation in the reference interpreter.

be. The code of the reference interpreter will reorder the ECMA-SL array according to the required endianness before (in the `GetValueFromBuffer` internal operation) or after (in the `SetValueInBuffer` internal operation) using any of these operators.

4.2 DataView

In this section we will do a deep-dive on `DataView` objects, one of the possible ways to interact with `ArrayBuffer` objects, as discussed in Section 4.1.

A `DataView` object behaves much like an `ArrayBuffer` wrapper that provides high-level operations to JavaScript users. Like a relational database view can make it easier to interface with a complex table, `DataView` instances provide the same level of abstraction for `ArrayBuffer` instances. These operations are available through a set of `getX` and `setX` methods where the `X` is replaced by one of the existing element types (`Int8`, `Uint8`, `Float32`, etc.).

Operator	Parameters	Semantics
`array_make`	• length integer value • init value	Creates a new array with the specified length with all its elements equal to the init value
`a_len`	• array	Returns the number of elements of the array
`a_nth`	• array • index integer value	Retrieves the element of the array in the specified position
`a_set`	• array • index integer value • value	Sets the element of the array in the specified position to the specified value

Table 4.1: Array operators added to ECMA-SL and their respective semantics.

Listing 4.3: Standard use of a `DataView` object.

```
1 var dv = new DataView(new ArrayBuffer(8));
2 dv.setInt8(0, 9);
3 dv.setInt16(4, 9);
4 dv.getInt32(0);
```

4.2.1 Examples

Consider the example in Listing 4.3, where we explore a portion of the API made available through `DataView` objects. Firstly, a `DataView` object is created using a new `ArrayBuffer` object of length 8. Next, we use the `setInt8` method to write the value `0x09` onto the first byte of the buffer. All the methods of `DataView` objects take in a byte index as the first argument that specifies where to start the read or write operation. Consequently, the next expression (`dv.setInt16(4, 9)`) writes the value `0x09` once again, but now starting at the 4th byte. Since the `Int16` type occupies two bytes, the `0x09` value will be spread across the 4th and 5th byte. The 4th byte will have its value changed to `0x09` while the 5th will remain `0x00` because these operations, by default, are big-endian (recall that in big-endian order the most significant digits are stored on the right). However, we can use one extra boolean argument to specify endianness. The expression `dv.setInt16(4, 9, true)` orders the bytes in a little-endian fashion, meaning that the value of the 4th byte would be `0x00` and the value of the 5th would be `0x09`. Finally, we use one of the `get` methods, specifically `getInt32`. Since the `Int32` type occupies 4 bytes and the byte index specified was 0, this method will retrieve the 4 first bytes of the buffer and interpret them as a signed integer. Considering the previous statements, the byte sequence expected to be retrieved from the `Data Block` is `0x09000000`. Interpreting this sequence as a big-endian signed integer, we obtain the value 9.

4.2.2 ECMAScript Specification

We will now go over the ECMAScript standard's specification of the `DataView` built-in library. The library is exposed via the `DataView` constructor in the global object. The constructor has no methods of its own, meaning its use is limited to the creation of new `DataView` objects. These objects allow ECMAScript programs to call the `get` and `set` methods mentioned before through their prototype.

DataView objects `DataView` objects are used as `ArrayBuffer` wrappers and interact with their bytes via method calls. They are created via the `DataView` constructor which has three parameters: (1) a non-optional `ArrayBuffer` object, that if missing, will result in a `TypeError` being thrown; (2) an optional `number` value for the byte offset that defaults to 0 if not present; and (3) an optional `number` value for the

Operator	Parameters	Semantics
`float_to_byte`	• float value	Converts a float value between -128 and 127 to a byte value
`float32_from_le_bytes`	• byte sequence	Interprets a little-endian sequence of 4 bytes as a Float32
`float32_from_be_bytes`	• byte sequence	Interprets a big-endian sequence of 4 bytes as a Float32
`float32_to_le_bytes`	• float32 value	Converts a Float32 value to a little-endian sequence of 4 bytes
`float32_to_be_bytes`	• float32 value	Converts a Float32 value to a big-endian sequence of 4 bytes
`float64_from_le_bytes`	• byte sequence	Interprets a little-endian sequence of 8 bytes as a Float64
`float64_from_be_bytes`	• byte sequence	Interprets a big-endian sequence of 8 bytes as a Float64
`float64_to_le_bytes`	• float64 value	Converts a Float64 value to a little-endian sequence of 8 bytes
`float64_to_be_bytes`	• float64 value	Converts a Float64 value to a big-endian sequence of 8 bytes
`int_from_le_bytes`	• byte sequence • length of sequence	Interprets the little-endian sequence of bytes as a signed integer
`int_from_be_bytes`	• byte sequence • length of sequence	Interprets the big-endian sequence of bytes as a signed integer
`int_to_le_bytes`	• float value • length of sequence	Converts the float value to an integer and encodes it as a little-endian sequence with the length specified
`int_to_be_bytes`	• float value • length of sequence	Converts the float value to an integer and encodes it as a big-endian sequence with the length specified
`uint_from_le_bytes`	• byte sequence • length of sequence	Interprets the little-endian sequence of bytes as an unsigned integer
`uint_from_be_bytes`	• byte sequence • length of sequence	Interprets the big-endian sequence of bytes as an unsigned integer
`uint_to_le_bytes`	• float value • length of sequence	Converts the float value to an unsigned integer and encodes it as a little-endian sequence with the length specified
`uint_to_be_bytes`	• float value • length of sequence	Converts the float value to an unsigned integer and encodes it as a big-endian sequence with the length specified

Table 4.2: Byte operators added to ECMA-SL and their respective semantics.

byte length that defaults to the length of the buffer it is not present. The values passed to the constructor will be stored in internal properties exclusive to `DataView` objects. Their internal methods, however, have the default implementation meaning they are ordinary objects. The extra internal properties of these objects are the following:

1. `[[DataView]]` - which functions as a flag so that functions that expect an argument to be a `DataView` object can verify if it is;
2. `[[ViewedArrayBuffer]]` - which is a reference to the `ArrayBuffer` object being wrapped;
3. `[[ByteOffset]]` - which holds an integer value that represents the byte offset associated with the view, meaning that if the offset is 5, for instance, the first 5 bytes of the `ArrayBuffer` are ignored;
4. `[[ByteLength]]` - which is the byte length of the view, meaning that view may ignore the final bytes of the `ArrayBuffer` if the view's `[[ByteLength]]` internal property is smaller than the buffer's `[[ArrayBufferByteLength]]` internal property.

In order to visualize this object structure, we present Figure 4.11. Here we see that the internal property `[[ViewedArrayBuffer]]` was populated with an `ArrayBuffer` object, as required. The optional arguments were not passed, so the `[[ByteOffset]]` was set to 0 and the `[[ByteLength]]` was set to the same value as the `ArrayBuffer`'s `[[ArrayBufferByteLength]]` internal property.

```
1 var ab = new ArrayBuffer(4);
2 var dv = new DataView(ab);
```

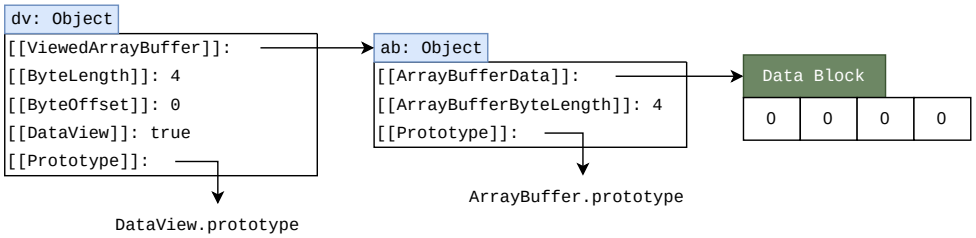


Figure 4.11: A `DataView` object after instantiation.

To make clearer the role of the `[[ByteOffset]]` and `[[ByteLength]]` properties, consider Figure 4.12. In this example, we create 3 `DataView` objects which share the same `ArrayBuffer` of length 4. We can see that the `DataView` object created in line 2 with no offset and length arguments, uses their default values, which are respectively set to 0 and 4 (to match the entire length of the `DataBlock` of the `ArrayBuffer`). In line 4, we create a `DataView` object with offset 1 and length 2, meaning that operations using this object will only be capable of accessing the second and third bytes of the `Data Block`. We can corroborate this via the value returned by the `getInt8` method in line 5, where although we used the argument 0, the value returned was 2 and not 1, since the second byte was retrieved instead of the first one. The code in line 6 throws a `RangeError` exception since the sum of the offset and length equals 5 which is larger than the value of the `[[ArrayBufferByteLength]]` internal property of the `ArrayBuffer`.

DataView.prototype The prototype object of `DataView` is where most of its functionality resides. This object exposes a series of `get` and `set` methods, shared across all `DataView` objects, to handle instances of `ArrayBuffer`. As these methods perform the same operation, but with differing element types, to eliminate duplicated logic, they all call an internal operation that takes the element type as a parameter. Read methods call `GetViewValue` while write methods call `SetViewValue`. Consider Figure 4.13 where this shared behavior is visible. In particular, we can see that both the `getUInt16` and `getUInt32`

```
1 var ab = new ArrayBuffer(4);
2 var dv1 = new DataView(ab);
3 dv1.setInt32(0, 0x01020304);
4 var dv2 = new DataView(ab, 1, 2);
5 dv2.getInt8(0); // 2
6 var dv3 = new DataView(ab, 1, 4); // RangeError
```

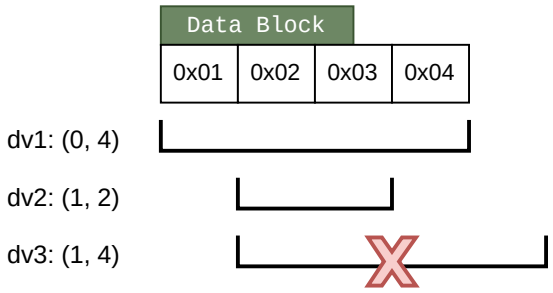


Figure 4.12: Visualization of the implications of the `[[ByteOffset]]` and `[[ByteLength]]` internal properties of `DataView` objects.

call the internal method `GetViewValue`, with the only difference being the last argument passed onto the `GetViewValue` internal operation. The `getUnit16` method passes the `string` `"Uint16"`, while the `getUnit32` method passes the `string` `"Uint32"`.

24.2.4.11 DataView.prototype.getUint16 (byteOffset [, littleEndian])

When the **`getUint16`** method is called with argument *byteOffset* and optional argument *littleEndian* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, let *littleEndian* be **false**.
3. Return GetViewValue(*v*, *byteOffset*, *littleEndian*, **`"Uint16"`**).

24.2.4.12 DataView.prototype.getUint32 (byteOffset [, littleEndian])

When the **`getUint32`** method is called with argument *byteOffset* and optional argument *littleEndian* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, let *littleEndian* be **false**.
3. Return GetViewValue(*v*, *byteOffset*, *littleEndian*, **`"Uint32"`**).

Figure 4.13: ES6 description of the `getUint16` and `getUint32` methods of `DataView.prototype` object.

Internal Operations Considering that all the methods of the `DataView.prototype` object call an internal operation we know that it is in these operations that the wrapped `ArrayBuffer` objects are accessed. The `GetViewValue` and `SetViewValue` operations begin by performing some input sanitation and then proceed to call the `GetValueFromBuffer` and `SetValueInBuffer` operations discussed in the previous section. In fact, the `GetViewValue` and `SetViewValue` operations are so similar that only differ in their final instruction. This can be seen in Figure 4.14, where in the bottom, there is a note explicitly saying that the algorithms are identical up to their final instruction. The final line of `GetViewValue` would instead call `GetValueFromBuffer(buffer, bufferIndex, type, isLittleEndian)`.

4.2.3 ECMA-SL Implementation

In this subsection we explain how the `DataView` built-in library was implemented in ECMARef6. Firstly, we will look at the structure of `DataView` objects. We will then look at the structure of the `DataView` constructor and prototype objects. We will conclude the subsection by looking at the internal implementation of a method and an internal operation of the `DataView` library.

24.2.1.2 SetViewValue (view, requestIndex, isLittleEndian, type, value)

The abstract operation SetViewValue with arguments *view*, *requestIndex*, *isLittleEndian*, *type*, and *value* is used by functions on DataView instances to store values into the view's buffer. It performs the following steps:

1. If Type(*view*) is not Object, throw a **TypeError** exception.
2. If *view* does not have a [[DataView]] internal slot, throw a **TypeError** exception.
3. Let *numberIndex* be ToNumber(*requestIndex*).
4. Let *getIndex* be ToInteger(*numberIndex*).
5. ReturnIfAbrupt(*getIndex*).
6. If *numberIndex* ≠ *getIndex* or *getIndex* < 0, throw a **RangeError** exception.
7. Let *isLittleEndian* be ToBoolean(*isLittleEndian*).
8. Let *buffer* be the value of *view*'s [[ViewedArrayBuffer]] internal slot.
9. If IsDetachedBuffer(*buffer*) is **true**, throw a **TypeError** exception.
10. Let *viewOffset* be the value of *view*'s [[ByteOffset]] internal slot.
11. Let *viewSize* be the value of *view*'s [[ByteLength]] internal slot.
12. Let *elementSize* be the Number value of the Element Size value specified in Table 49 for Element Type *type*.
13. If *getIndex* +*elementSize* > *viewSize*, throw a **RangeError** exception.
14. Let *bufferIndex* be *getIndex* + *viewOffset*.
15. Return SetValueInBuffer(*buffer*, *bufferIndex*, *type*, *value*, *isLittleEndian*).

NOTE The algorithms for GetViewValue and SetViewValue are identical except for their final steps.

Figure 4.14: ES6 description of the `SetViewValue` operation.

DataView Objects `DataView` objects have no named properties like `ArrayBuffer` objects, so their `JSProperties` object is not populated. They have the same internal properties mentioned in the specification: `[[ViewedArrayBuffer]]`; `[[DataView]]`; `[[ByteOffset]]`; and `[[ByteLength]]`. Consider Figure 4.15, where a `DataView` object is created to wrap an `ArrayBuffer` of 4 bytes. As there are no properties like the `Data Block` that are not matched by an ECMA-SL primitive value, the structure of the internal representation and the one in the specification is almost identical.

```
1 var ab = new ArrayBuffer(4);
2 var dv = new DataView(ab);
```

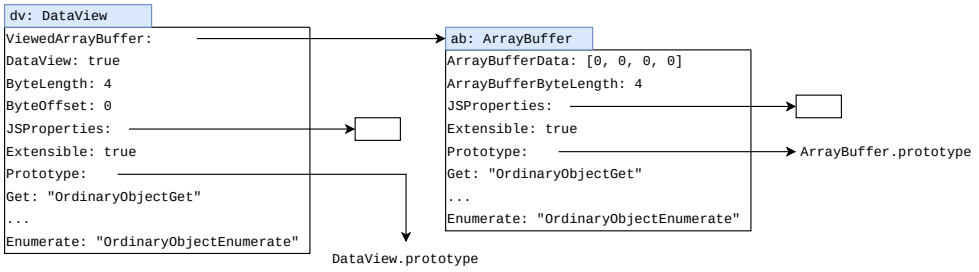


Figure 4.15: Internal representation of a `DataView` object.

DataView constructor The `DataView` constructor object has the internal properties of a typical function object, such as `[[Call]]` and `[[Construct]]`, and its `[[Code]]` internal property has the name of the ECMA-SL function that contains the implementation of the `DataView` constructor. In terms of named properties, the constructor has only the `prototype` property which will point to the prototype object of all `DataView` instances. An illustration of the constructor and its properties can be seen in Figure 4.16, where we can see that the object's only contribution to the library is the construction of `DataView` objects as there are no methods associated with it.

DataView.prototype As we discussed before and considering the lack of methods in the constructor, the prototype object is expected to hold most of the functionality of this library. To do that, its `JSProperties` object is filled with all the different `get` and `set` methods of the different element types. Consider Figure 4.17 that shows the graph of ECMA-SL objects connected to the prototype object.

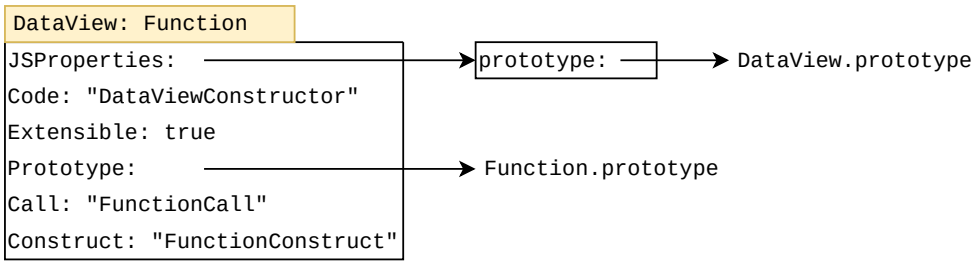


Figure 4.16: Internal representation of the `DataView` constructor.

The prototype object has the `getInt8` property of its `JSProperties` object pointing to a `property descriptor` that in turn points to a `Function` object. This `Function` object's `Code` internal property contains the name of the ECMA-SL function that implements the `getInt8` method. The remaining methods of the prototype were omitted to make a more concise diagram, but they will also be present in the `JSProperties` object and will have their own descriptors and corresponding `Function` objects.

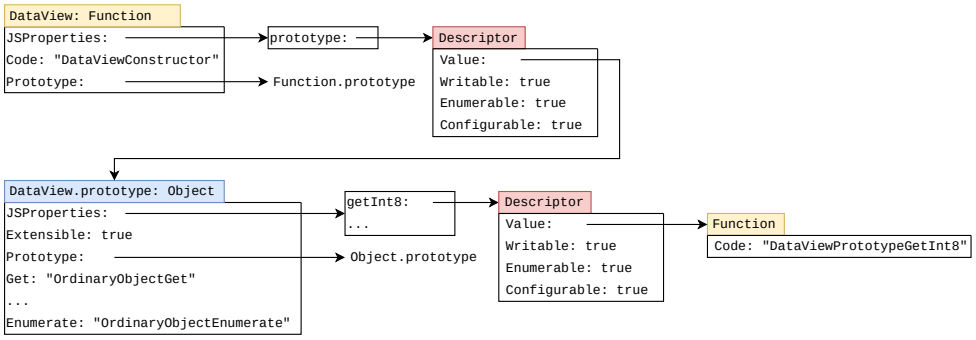
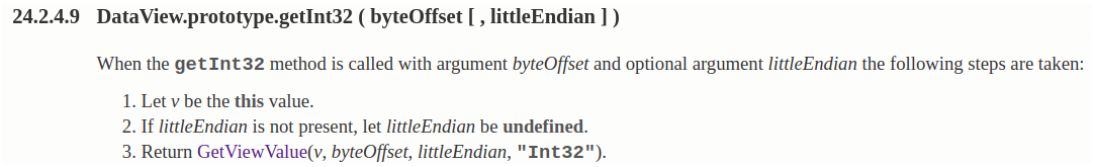


Figure 4.17: Internal representation of the `DataView.prototype` object.

As most of the ECMA-SL code shown so far in this document, the statements in the code of the ECMA-SL functions that implement the methods of `DataView.prototype` mimic the standard's pseudo-code. To make that comparison, consider Figure 4.18 where we compare the standard's description of the `getInt32` method to our own implementation in ECMARef6. In our implementation of the method, we were once again able to keep the similarity between our statements and the pseudo-code instructions of the standard.

Internal Operations Contrary to methods, the internal operations of the library have no ECMAScript `Function` objects and exist only as ECMA-SL functions. Consider Figure 4.19 where we explore if in the internal operations our implementation remains truthful to the standard. We can see that the strategy of matching line-by-line keeps paying dividends as the implementation is little more than a copy of the specification. Although it would be possible to add a `sanitizeInputs` function to our implementation to reduce duplicated logic in the `GetViewValue` and `SetViewValue` operations, it would break the line-by-line philosophy we employed in the rest of the implementation so we opted to not to do so.

Overall, the similarities between the standard's specification and our implementation in terms of data structures and algorithms, contribute to establish a sense of trust and security in our implementation of the `DataView` built-in library.



24.2.4.9 DataView.prototype.getInt32 (byteOffset [, littleEndian])

When the **getInt32** method is called with argument *byteOffset* and optional argument *littleEndian* the following steps are taken:

1. Let *v* be the **this** value.
2. If *littleEndian* is not present, let *littleEndian* be **undefined**.
3. Return GetViewValue(*v*, *byteOffset*, *littleEndian*, **"Int32"**).

(a) Standard description of the `getInt32` method of the `DataView.protoype` object.

```
function DataViewPrototypeGetInt32(global, this, NewTarget, strict, args) {
  byteOffset := getOptionalParam(args, 0);
  littleEndian := getOptionalParam(args, 1);
  /* 1. Let v be the this value. */
  v := this;
  /* 2. If littleEndian is not present, */
  if (littleEndian = null) {
    /* let littleEndian be undefined */
    littleEndian := 'undefined
  };
  /* 3. Return GetViewValue(v, byteOffset, littleEndian, "Int32"). */
  return GetViewValue(v, byteOffset, littleEndian, "Int32")
};
```

(b) Implementation of the `getInt32` method of the `DataView.prototype` object in the reference interpreter.

Figure 4.18: Comparison between the standard description of the `getInt32` method and its implementation in the reference interpreter.

4.3 TypedArray

In this section we present both the ECMAScript specification as well as the ECMARef6 implementation of the `TypedArray` built-in library. The `TypedArray` library provides another way to interface with `ArrayBuffer` objects and their byte arrays.

4.3.1 Examples

Typical usage of `TypedArray` objects is similar to the usage of `Array` objects. It is possible to use bracket notation and integer values to index these objects. As demonstration, consider the example in Listing 4.4. An instance of `TypedArray` is first created via the `Int8Array` (all the constructors made available by the `TypedArray` library are listed in the next subsection). Using the argument 4, makes it so the constructor code allocates its own `ArrayBuffer` instance to use. In line 2, we assign the value 1 to the property 0 of the object. In another object, this would associate a `property descriptor` with the key 0. However, the semantics of `TypedArray` objects make it so this assignment writes in its underlying buffer instead. Since the object is of the `Int8Array` type, it interprets the value 1 as a signed integer of 1 byte and change the value of the first byte of the buffer accordingly. In line 3, we see more parallels with the `Array` API of ECMAScript, as we use the `map` method to create a new `Int8Array` object with the value of all its bytes doubled.

Listing 4.4: Example of the use of a `TypedArray` object.

```
1 var ta = new Int8Array(4);
2 ta[0] = 1;
3 var doubled = ta.map(x => x * 2);
```

4.3.2 ECMAScript Specification

The previous built-in libraries we analyzed, exposed a single object, namely a constructor. In the case of the `TypedArray` library, various constructors are exposed through the global object. Each constructor

24.2.1.1 GetViewValue (view, requestIndex, isLittleEndian, type)

The abstract operation GetViewValue with arguments *view*, *requestIndex*, *isLittleEndian*, and *type* is used by functions on DataView instances is to retrieve values from the view's buffer. It performs the following steps:

1. If Type(*view*) is not Object, throw a **TypeError** exception.
2. If *view* does not have a [[DataView]] internal slot, throw a **TypeError** exception.
3. Let *numberIndex* be ToNumber(*requestIndex*).
4. Let *getIndex* be ToInteger(*numberIndex*).
5. ReturnIfAbrupt(*getIndex*).
6. If *numberIndex* ≠ *getIndex* or *getIndex* < 0, throw a **RangeError** exception.
7. Let *isLittleEndian* be ToBoolean(*isLittleEndian*).
8. Let *buffer* be the value of *view*'s [[ViewedArrayBuffer]] internal slot.
9. If IsDetachedBuffer(*buffer*) is **true**, throw a **TypeError** exception.
10. Let *viewOffset* be the value of *view*'s [[ByteOffset]] internal slot.
11. Let *viewSize* be the value of *view*'s [[ByteLength]] internal slot.
12. Let *elementSize* be the Number value of the Element Size value specified in Table 49 for Element Type *type*.
13. If *getIndex* +*elementSize* > *viewSize*, throw a **RangeError** exception.
14. Let *bufferIndex* be *getIndex* + *viewOffset*.
15. Return GetValueFromBuffer(*buffer*, *bufferIndex*, *type*, *isLittleEndian*).

(a) Standard description of the `GetViewValue` internal operation.

```
function GetViewValue (view, requestIndex, isLittleEndian, type) {
  /* 1. If Type(view) is not Object, throw a TypeError exception. */
  if (!(Type(view) = "Object")) {
    throw TypeErrorConstructorInternal()
  };
  /* 2. If view does not have a [[DataView]] internal slot, throw a TypeError
    exception. */
  if (!( "DataView" in_obj view)) {
    throw TypeErrorConstructorInternal()
  };
  /* 3. Let numberIndex be ToNumber(requestIndex). */
  numberIndex := ToNumber(requestIndex);
  /* 4. Let getIndex be ToInteger(numberIndex). */
  getIndex := ToInteger(numberIndex);
  /* 5. ReturnIfAbrupt(getIndex). */
  @ReturnIfAbrupt(getIndex);
  /* 6. If numberIndex != getIndex or getIndex < 0, throw a RangeError exception.
     */
  if ((!(numberIndex = getIndex)) ||| (getIndex < 0.)) {
    throw RangeErrorConstructorInternal()
  };
  /* 7. Let isLittleEndian be ToBoolean(isLittleEndian). */
  isLittleEndian := ToBoolean(isLittleEndian);
  /* 8. Let buffer be the value of view's [[ViewedArrayBuffer]] internal slot. */
  buffer := view.ViewedArrayBuffer;
  /* 9. If IsDetachedBuffer(buffer) is true, throw a TypeError exception. */
  if (IsDetachedBuffer(buffer) = true) {
    throw TypeErrorConstructorInternal()
  };
  /* 10. Let viewOffset be the value of view's [[ByteOffset]] internal slot. */
  viewOffset := view.ByteOffset;
  /* 11. Let viewSize be the value of view's [[ByteLength]] internal slot. */
  viewSize := view.ByteLength;
  /* 12. Let elementSize be the Number value of the Element Size value specified
    in Table 49 for Element Type type. */
  elementSize :=  getElementSize(type);
  /* 13. If getIndex +elementSize > viewSize, throw a RangeError exception. */
  if ((getIndex + int_to_float(elementSize)) > viewSize) {
    throw RangeErrorConstructorInternal()
  };
  /* 14. Let bufferIndex be getIndex + viewOffset. */
  bufferIndex := getIndex + viewOffset;
  /* 15. Return GetValueFromBuffer(buffer, bufferIndex, type, isLittleEndian). */
  return GetValueFromBuffer(buffer, int_of_float(bufferIndex), type,
    isLittleEndian)
};
```

(b) Implementation of the `GetViewValue` internal operation in the reference interpreter.

Figure 4.19: Comparison between the standard description of the `GetViewValue` operation and its implementation in the reference interpreter.

produces objects that represent byte sequences of different element types. The existing element types and their size are listed in Table 4.3.

Types	Int8	UInt8	UInt8C	Int16	UInt16	Int32	UInt32	Float32	Float64
Size in bytes	1	1	1	2	2	4	4	4	8

Table 4.3: Element types and their byte size.

TypedArray Objects Considering the large number of different constructors made available by the library, it may seem odd that we keep referencing the objects built from the constructors as `TypedArray` objects, instead of `Int8Array` objects for example. However, regardless of the constructor used to create the objects they all share the same prototype object and internal methods, meaning that they can be used in the same way without any issues. As we will see further into this section, the only difference between the different types of `TypedArray` is how many bytes they handle at a time and how those bytes are interpreted.

Integer Indexed Exotic Objects In the previous paragraph, we mentioned that all `TypedArray` objects share the same internal methods. These methods are not the default methods, however, meaning that `TypedArray` objects are exotic objects. More concretely, they are `Integer Indexed` exotic objects. The standard defines `Integer Indexed` *exotic objects* as "an exotic object that performs special handling of integer index property keys". To accommodate this functionality, `integer indexed` exotic objects, and by extension `TypedArray` objects, have 4 additional internal properties:

1. `[[ViewedArrayBuffer]]` - a reference to an `ArrayBuffer` object;
2. `[[ByteOffset]]` - this property serves the same purpose has the `[[ByteOffset]]` property of `DataView` objects. It indicates the a certain amount of bytes of the buffer should be skipped. It must be 0 to take full advantage of the size of the buffer;
3. `[[ArrayLength]]` - the number of elements that the buffer can contain. Given that different elements have different byte sizes, `[[ArrayLength]]` $= \frac{bufferSize - byteOffset}{elementSize}$, with $bufferSize$ being the size of the buffer in bytes and $byteOffset$ being the value of `[[ByteOffset]]`. For instance, if the element is `Int16` which takes up 2 bytes and the buffer has 4 bytes in length, the `[[ArrayLength]]` property has the value 2 if the offset is 0 and 1 if the offset is 2;
4. `[[TypedArrayName]]` - the name of the `TypedArray` constructor that instantiated this object.

Since the default implementation of most of the internal methods is based around reading and writing the keys and values of named properties, some of them had to be changed to instead interact with the `ArrayBuffer` when the property name is a number. When the property name is not a number, the methods fallback to the default implementation. The following is a brief description of what the altered internal methods perform when the property name is a number:

1. `[[GetOwnProperty]]` - fetches a value from the buffer and wraps it in a new `data property descriptor`. Since the value is not stored in a named property, a new `property descriptor` must be created every time this method is called;
2. `[[HasProperty]]` - returns true if the index is in the bounds of the array;
3. `[[DefineOwnProperty]]` - if the `property descriptor` has its `[[Writable]]`, `[[Enumerable]]` and `[[Configurable]]` equal to true, then it writes the value in the `[[Value]]` field in the buffer;

4. [[Get]] - retrieves a value from the buffer;
5. [[Set]] - writes a value in the buffer;
6. [[OwnPropertyKeys]] - includes the keys from 0 to [[ArrayLength]] before any of the keys of its named properties. This makes it so a for statement can be used to iterate over its values.

Consider the code-snippet and object diagram in Figure 4.20 that displays the structure of an instance of TypedArray, created through an Int16Array object. An object created by any of the other constructors would have the same structure but different values for the [[ArrayLength]], [[TypedArrayName]] and [[Prototype]] properties. The various constructors of TypedArray behave in different ways depending on the first argument passed. Passing a number, equates to passing the length of the array which means two things: (1) the [[ArrayLength]] property will have the same value as the one that was passed; and (2) that a new ArrayBuffer object will be created with the number of bytes necessary to support the array's length. In our example, since Int16 elements occupy 2 bytes and length of the array is 2, the buffer has 4 bytes. This newly created ArrayBuffer is referenced by the [[ViewedArrayBuffer]] internal property. The [[ByteOffset]] property is set to 0 by default, but it can be changed if the constructor is used in a different manner, which will be discussed later. Finally, the [[TypedArrayName]] property has the value "Int16Array" since the object was created by the constructor with that name.

```
1 var ta = new Int16Array(2);
```

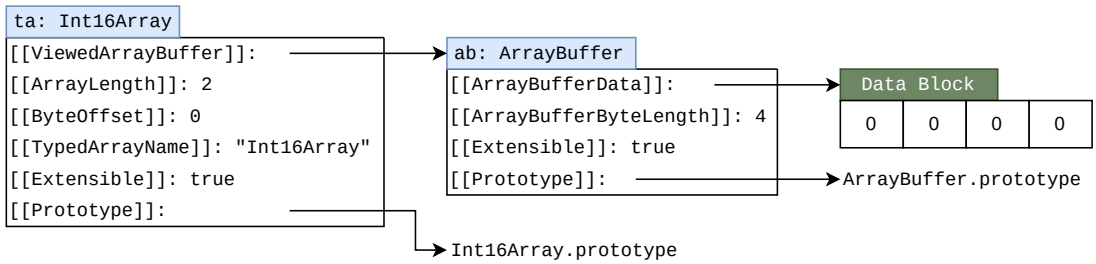


Figure 4.20: Caption

We now present some details of the internal methods of TypedArray objects and how they interact with their buffers. The methods that do read operations, [[GetOwnProperty]] and [[Get]], have in their instructions a call to an internal operation named IntegerIndexedElementGet while the ones that do write operations, [[DefineOwnProperty]] and [[Set]], call IntegerIndexedElementSet. These operations are similar in concept to the GetViewValue and SetViewValue of the DataView library. Consider the excerpt of the standard in Figure 4.21, that describes the IntegerIndexedElementGet operation, where, like in GetViewValue, we start by performing some input sanitation. Then the internal [[TypedArrayName]] property is used to determine the element type and size. With the element size, calculations are done to determine the byte index to use when accessing the buffer. Finally, we use the GetValueFromBuffer operation, as we did in GetViewValue, to obtain the value. All higher-level methods of the TypedArray objects that access the buffer, do so through their internal methods and therefore, these internal operations.

TypedArray constructors As we have established, there is a constructor for each type of element, but there is one more constructor that we have not yet mentioned. That is because this constructor is not available through the global object and is not meant to be used directly in ECMAScript programs. It is named %TypedArray% in the standard and it acts a super constructor of all other types of TypedArray. The way to achieve this in ECMAScript is by making the [[Prototype]] internal property of

9.4.5.8 IntegerIndexedElementGet (O, index)

The abstract operation IntegerIndexedElementGet with arguments *O* and *index* performs the following steps:

1. Assert: Type(*index*) is Number.
2. Assert: *O* is an Object that has [[ViewedArrayBuffer]], [[ArrayLength]], [[ByteOffset]], and [[TypedArrayName]] internal slots.
3. Let *buffer* be the value of *O*'s [[ViewedArrayBuffer]] internal slot.
4. If IsDetachedBuffer(*buffer*) is **true**, throw a **TypeError** exception.
5. If IsInteger(*index*) is **false**, return **undefined**
6. If *index* = −0, return **undefined**.
7. Let *length* be the value of *O*'s [[ArrayLength]] internal slot.
8. If *index* < 0 or *index* ≥ *length*, return **undefined**.
9. Let *offset* be the value of *O*'s [[ByteOffset]] internal slot.
10. Let *arrayTypeName* be the String value of *O*'s [[TypedArrayName]] internal slot.
11. Let *elementSize* be the Number value of the Element Size value specified in Table 49 for *arrayTypeName*.
12. Let *indexedPosition* = (*index* × *elementSize*) + *offset*.
13. Let *elementType* be the String value of the Element Type value in Table 49 for *arrayTypeName*.
14. Return GetValueFromBuffer(*buffer*, *indexedPosition*, *elementType*).

Figure 4.21: ES6 description of the `IntegerIndexedElementGet` internal operation.

the `TypedArray` constructors point to the `%TypedArray%` constructor. This relationship forms an intricate network of constructor and prototype objects that we will present in an iterative manner. `TypedArray` constructors have the following two properties that are not present in typical constructors:

1. `BYTES_PER_ELEMENT` - this is a named property that holds the byte size of the element represented by the constructor. In the `Float64Array` object, this property's value is 8, as `Float64` elements have a byte size of 8;

2. `[[TypedArrayConstructorName]]` - this is an internal property that holds a `string` value with the name of the constructor. In the `Float64Array` object, this property's value is `"Float64Array"`.

To explore the network of constructor and prototype objects, let us first consider the diagram in Figure 4.22 that shows the representation of the `Int32Array` constructor. `Property descriptors` were omitted from the diagram to highlight the most important aspects. The `Int32Array` constructor has the named property `BYTES_PER_ELEMENT` with value 4 and `[[TypedArrayConstructorName]]` with value `"Int32Array"`, as expected. It also has the named property `prototype`, present in all constructors, that points to an ordinary object `Int32Array.prototype`. Unlike most constructors however, its internal `[[Prototype]]` property does not reference the `Function.prototype` object. It instead points to the super `%TypedArray%` constructor.

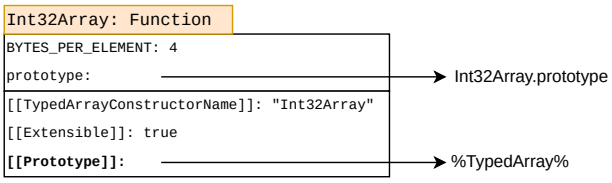


Figure 4.22: Representation of the `Int32Array` object.

Expanding the network objects, we now add the `%TypedArray%` constructor, as well as the `Int8Array` and `Int16Array` constructors, as shown in Figure 4.23. They all have a `[[BYTES_PER_ELEMENT]]` property with the appropriate value and a `prototype` property that points to the respective prototype object. The `[[Prototype]]` internal property is equal in all constructors and points to the super constructor, meaning they all share its `from` and `of` methods.

The code of the `%TypedArray%` constructor is also called by the code of the other constructors, as the excerpt of the standard in Figure 4.24 shows. When writing the standard, to avoid duplication of text, the authors use the word "TypedArray" italicized to symbolize any of the `TypedArray` constructors

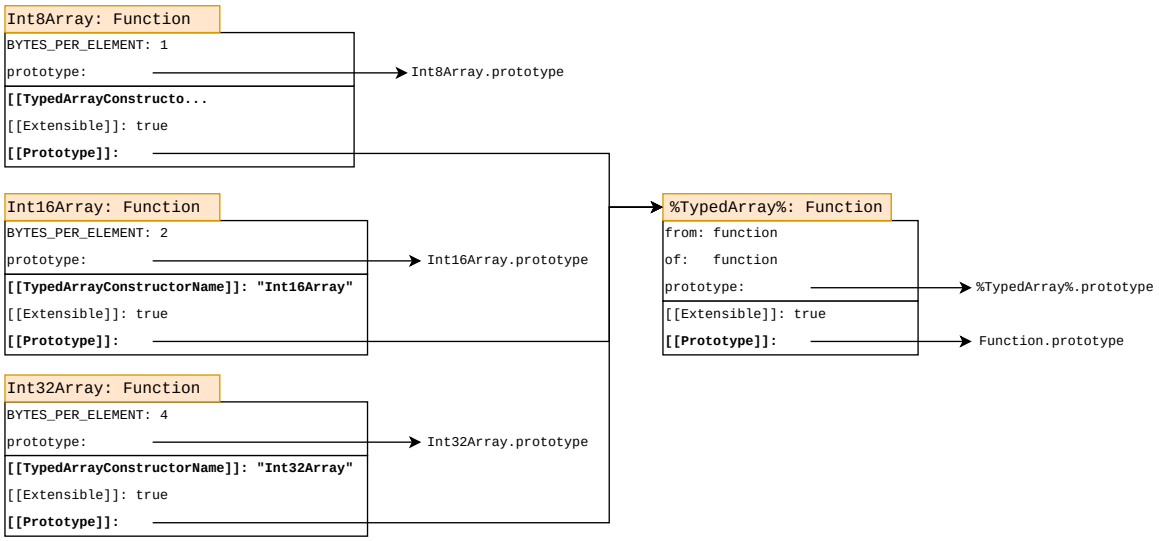


Figure 4.23: Representation of the `Int8Array`, `Int16Array`, `Int32Array` and `%TypedArray%` object.

that are not `%TypedArray%`. The code of these objects obtains the `%TypedArray%` constructor by calling their own `[[GetPrototypeOf]]` internal method. With it, it then calls the `Construct` operation which executes the code of the super constructor to build the appropriate `TypedArray` object. In order to know what is the appropriate type, it uses the `NewTarget` value it receives as argument. This value represents the constructor function that started the chain of constructor calls. If the ECMAScript expression `new Int8Array(20)` was executed, then the value of `NewTarget` is the `Int8Array` constructor. With this, the super constructor can obtain the element type and size from the `[[TypedArrayConstructorName]]` property of the `NewTarget`.

22.2.4.1 *TypedArray(... argumentsList)*

A *TypedArray* constructor with a list of arguments *argumentsList* performs the following steps:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *here* be the active function.
3. Let *super* be *here*.[[GetPrototypeOf]]().
4. ReturnIfAbrupt(*super*).
5. If IsConstructor (*super*) is **false**, throw a **TypeError** exception.
6. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that invoked the active function.
7. Return Construct(*super*, *argumentsList*, NewTarget).

Figure 4.24: ES6 description of the *TypedArray* constructors.

%TypedArray%.prototype Object Although there are multiple types of `TypedArray`, the logic they use is identical. For this reason, their prototype chain culminates in a single prototype object shared amongst all `TypedArray` instances, commonly referred to in the standard as `%TypedArrayPrototype%` or `%TypedArray%.prototype`. It is this object that holds all the methods available to `TypedArray` objects. A fragment of this prototype-chain can be visualized in Figure 4.25. The execution of the code-snippet creates two `TypedArray` objects: one `Int16Array` and one `Int32Array`. Because they were created using different constructors, the `[[Prototype]]` property of these objects points to different objects. These objects have a copy of the `BYTES_PER_ELEMENT` named property of the constructors and their own `[[Prototype]]` property. Contrary to the `ta1` and `ta2` objects, that have different prototypes, the `Int16Array.prototype` and `Int32Array.prototype` objects have the same one. This object is the `%TypedArray%.prototype` object. This prototype-chain, makes all the named properties of the

%TypedArray%.prototype object available to all objects that come before it in the chain, including all the TypedArray objects. The pattern observed here of the immediate prototype of TypedArray instances having a single named property BYTES_PER_ELEMENT and %TypedArrayPrototype% as its prototype holds true for all other types of TypedArray that were not mentioned and not just for the Int16Array and Int32Array types.

```
1 var ta1 = new Int16Array(2);
2 var ta2 = new Int32Array(1);
```

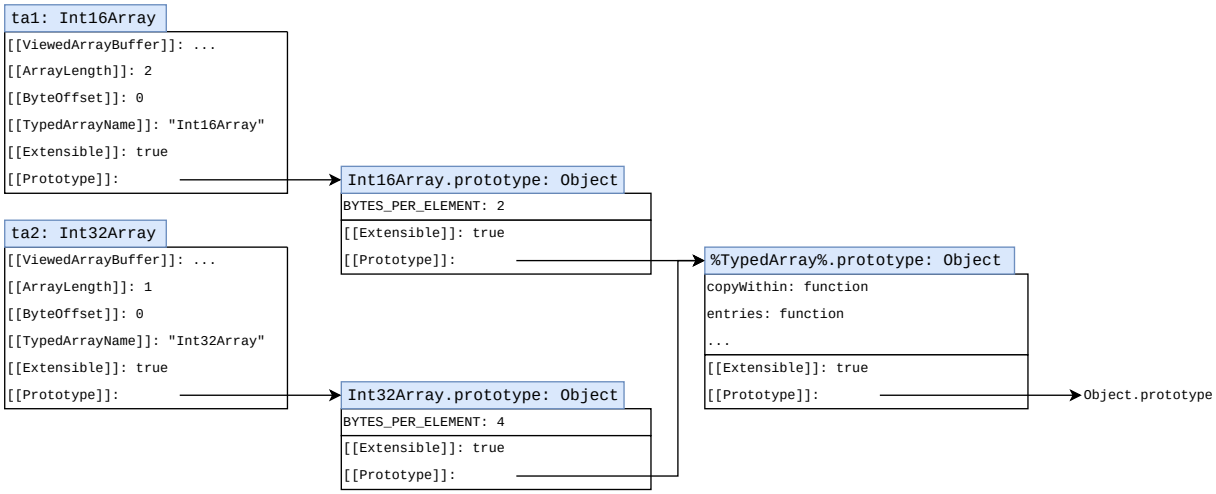


Figure 4.25: Representation of the prototype-chain of Int16Array and Int32Array objects.

Not only do the different types of TypedArray share the same methods, there are also some algorithms shared between the Array and the TypedArray libraries. Figure 4.26 shows the standard's description of the reverse method, where we can see that simply by changing the way in which we look up the length of the object, in this case by accessing the [[ArrayLength]] internal property instead of calling [[Get]]("length"), the reverse algorithm becomes reusable. The ease with which some algorithms can be reused comes from the fact that TypedArray and Array objects both specialize in the handling of properties whose keys are integer indexes.

22.2.3.21 %TypedArray%.prototype.reverse ( )

%TypedArray%.**prototype.reverse** is a distinct function that implements the same algorithm as **Array.prototype.reverse** as defined in 22.1.3.20 except that the **this** object's [[ArrayLength]] internal slot is accessed in place of performing a [[Get]] of **"length"**. The implementation of the algorithm may be optimized with the knowledge that the **this** value is an object that has a fixed length and whose integer indexed properties are not sparse. However, such optimization must not introduce any observable changes in the specified behaviour of the algorithm.

This function is not generic. ValidateTypedArray is applied to the **this** value prior to evaluating the algorithm. If its result is an abrupt completion that exception is thrown instead of evaluating the algorithm.

Figure 4.26: Section 22.2.3.21 of the standard which describes the reverse method of the TypedArray prototype.

To showcase the full depth of the prototype-chain in a single diagram, we present Figure 4.27. Explaining this diagram would be to duplicate our explanation of the previous two diagrams, so we refrain from it. However, since this diagram is a narrower (contains only the Int16 type), but deeper representation of the chain, it might be a clearer to visualize it.

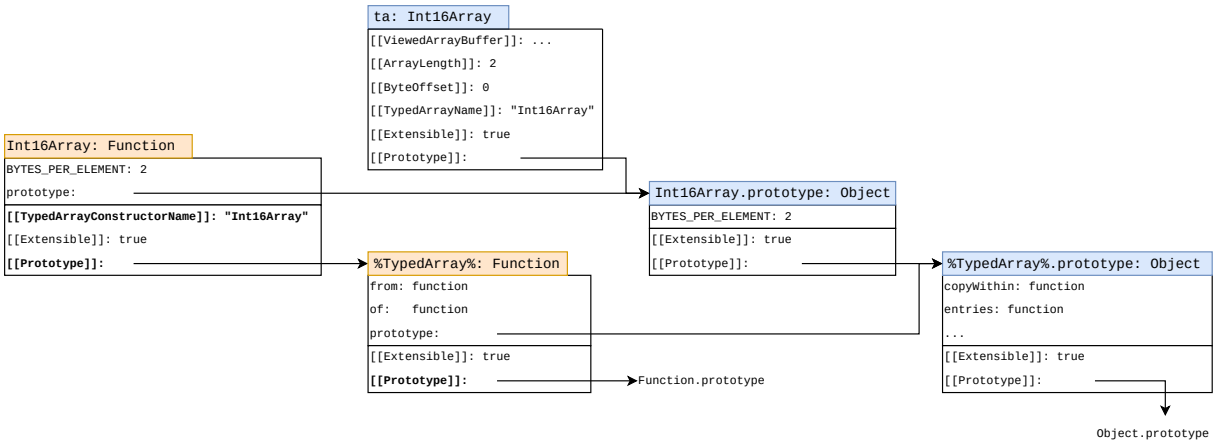


Figure 4.27: Diagram representing the full prototype-chain using `Int16Array` as a starting point.

4.3.3 ECMA-SL Implementation

In the coming subsection, we present our own implementation of the `TypedArray` library. When it comes to the internal representation of the `TypedArray` objects, much like the `DataView` objects, we achieve an almost identical object structure to the one described in the standard. For this reason, our focus is shifted towards the implementation of the chain of constructors.

TypedArray Constructors The way we implemented the various constructor objects does not differ greatly from standard's description, however, we managed to use only one ECMA-SL function for the `TypedArray` constructors besides `%TypedArray%`. Consider the diagram in Figure 4.28 that contains the representation of the `Int8Array`, `Int16Array`, `Int32Array` and `%TypedArray%` constructors. The first three have the `JSProperties` object populated with the `BYTES_PER_ELEMENT` and `prototype` properties that point to their respective `property descriptors`. The latter has different properties in its `JSProperties` object, namely the `from` and `of` method, which point to `data property descriptors` that were omitted to make the diagram more digestible. The most important property in all the constructors is the `Code` property which has the same value in the `Int8Array`, `Int16Array`, `Int32Array` constructors. The `%TypedArray%` constructor has its own `Code` value, `"TypedArrayConstructor"`, because it is meant to be called by all the other constructors via the execution of the `TypedArraySubConstructor` ECMA-SL function. This is how the various `TypedArray` types share instantiation logic.

However, we did not achieve this behavior as cleanly as we would have liked. Consider Figure 4.29, which contrasts our `TypedArraySubConstructor` function against the standard's description. The first difference comes in the form of the name, that we cannot match as font styling is not possible in ECMA-SL. Still in the signature of the function, we can see that the parameters are different as well. All ECMA-SL functions exposed in function objects, have this same signature except for the `here` parameter. Looking at line 8 of our implementation, helps to understand this extra parameter. Since ECMARef6 was built on top of ECMARef5 and the ES5 had no mention of "the active function" in any instruction there was no need to support this functionality in the reference interpreter. To date, the ECMARef6 reference interpreter still does not have this feature, but there is work being done on it by other students. Nevertheless, it leaves our implementation a bit "hacky", as only in this specific function call, an extra argument is passed. The `here` parameter will contain the function object whose code is currently being executed. As with previous comparisons between the standard and our implementation, the rest of the statements of the function mimic the standard's pseudo-code instructions.

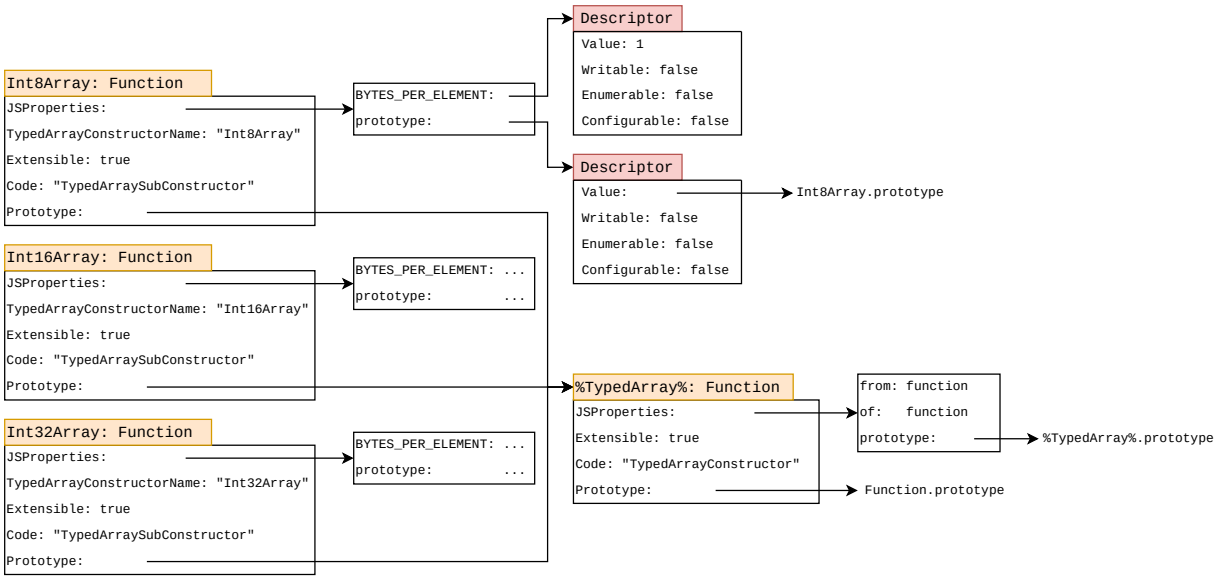


Figure 4.28: Internal representation of the chain of `TypedArray` constructors.

4.4 Symbol

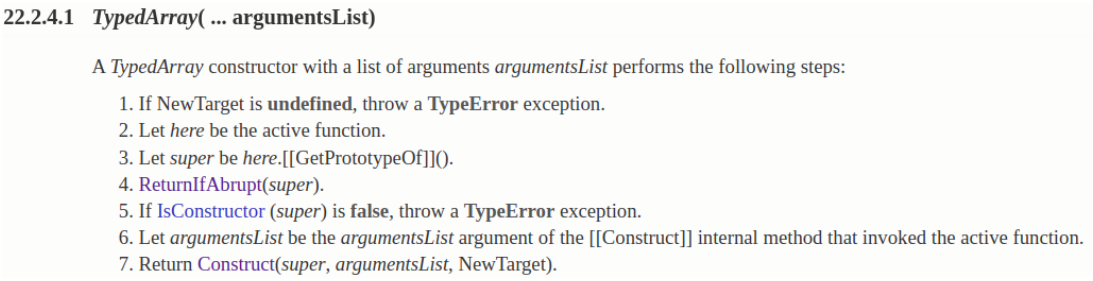
In this section, we present the ECMAScript specification and our ECMA-SL implementation of the `Symbol` built-in library. The purpose of the `Symbol` library and its objects is to provide another type of property key besides `string` values.

4.4.1 Examples

To demonstrate the use of `Symbol` values consider Listing 4.5. In this code-snippet, we see the use of the `Symbol` constructor to create two separate `Symbol` values. Although the arguments used in their creation were identical, they are still distinguishable. In line 4 we create an ECMAScript object using the object literal expression and then use our `Symbol` values `sym1` and `sym2` to add some properties to the newly created object. We associate the `string` `"xpto"` with the property key `sym1`. On the left-hand side of the assignment expression, we use bracket notation, as that is the only way to use `Symbol` values. The final instruction of our code-snippet does another assignment, this time using the other `Symbol` value. Since both the `Symbol` values were created with the same arguments, one may think that this last instruction essentially overwrites what was done in the previous one, by replacing `"xpto"` with `"abc"`. However, since `Symbol` values are always unique from each other, the last instruction will instead create a new property associated with the key `sym2`. In the end, we end up with an object we two properties: one associated with the `sym1` key; and another associated with the `sym2` key.

Listing 4.5: Example of the creation and use of `Symbol` values as property keys.

```
1 var sym1 = Symbol("example");
2 var sym2 = Symbol("example");
3
4 var obj = {};
5
6 obj[sym1] = "xpto";
7 obj[sym2] = "abc";
```



22.2.4.1 ***TypedArray*(... argumentsList)**

A *TypedArray* constructor with a list of arguments *argumentsList* performs the following steps:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *here* be the active function.
3. Let *super* be *here*.[[GetPrototypeOf]]().
4. ReturnIfAbrupt(*super*).
5. If IsConstructor (*super*) is **false**, throw a **TypeError** exception.
6. Let *argumentsList* be the *argumentsList* argument of the [[Construct]] internal method that invoked the active function.
7. Return Construct(*super*, *argumentsList*, NewTarget).

(a) Standard description of the *TypedArray* constructor.

```
1  function TypedArraySubConstructor(ctx, this, NewTarget, strict, argumentsList,
       here) {
2    /* 1. If NewTarget is undefined, */
3    if (NewTarget = 'undefined) {
4      /* throw a TypeError exception */
5      throw TypeErrorConstructorInternal()
6    };
7    /* 2. Let here be the active function. */
8    /* TODO: Instruction not yet implemented. */
9    /* 3. Let super be here.[[GetPrototypeOf]](). */
10   super := {here.GetPrototypeOf}(here);
11   /* 4. ReturnIfAbrupt(super). */
12   @ReturnIfAbrupt(super);
13   /* 5. If IsConstructor (super) is false, */
14   if (IsConstructor(super) = false) {
15     /* throw a TypeError exception */
16     throw TypeErrorConstructorInternal()
17   };
18   /* 6. Let argumentsList be the argumentsList argument of the [[Construct]]
19      internal method that invoked the active function. */
20   argumentsList := argumentsList;
21   /* 7. Return Construct(super, argumentsList, NewTarget). */
22   return Construct(ctx, null, super, argumentsList, NewTarget)
23 }
```

(b) Implementation of the *TypedArray* constructor in the reference interpreter.

Figure 4.29: Comparison between the standard description of the *TypedArray* constructor and its implementation in the reference interpreter.

4.4.2 ECMAScript Specification

The entry point of the `Symbol` library in the ECMAScript specification is the `Symbol` constructor. This constructor creates `Symbol` values, which are primitives, unlike most constructors that create objects. In this subsection we first explain the difference between `Symbol` values and objects and how each is created and used. Afterwards, we present the `Symbol` constructor and its named properties, which are the main contribution of the `Symbol` library to the ECMAScript language. Finally, we dive into some of the methods available to `Symbol` objects through their prototype object.

Symbol Values A `Symbol` value is a "primitive value that represents a unique, non-String Object property key". They are immutable and, even though they are not objects, have an internal property called `[[Description]]` that can be either `undefined` or a `string` value. In order to visualize their representation consider Figure 4.30. We start by creating two `Symbol` values using the same `Symbol("example")` expression. Note that the `new` keyword is not used when creating `Symbol` values and using it will throw a `TypeError` exception. Looking at the diagram, they look indistinguishable, except for the fact that they are two separate identical values. However, the `sym1 !== sym2` comparison in line 3 evaluates to `true`. This is because `Symbol` values are unique and therefore the result of the equality and inequality operators always indicates that two `Symbol` values are different. Even if they possess the same `[[Description]]` as in this example.

```
1 var sym1 = Symbol("example");
2 var sym2 = Symbol("example");
3 sym1 !== sym2; // true
```

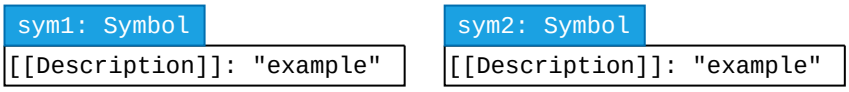


Figure 4.30: Two `Symbol` primitive values.

Symbol Objects As we have seen, `Symbol` values are primitive values and not the typical objects used in the rest of the built-in libraries. They have no `[[Prototype]]` internal property or named properties, so what is supposed to happen when the expression `Symbol("xpto").toString()` is evaluated? Using dot notation to access a property creates what is called a `Property Reference`. These have two components: (1) the expression before the dot, called the *base*; and (2) the expression after the dot, called the *reference name*. When the `base` of a `property reference` is not an object, it is converted to one by calling the `ToObject` internal operation of the ECMAScript standard. The `ToObject` internal operation is the only way to create `Symbol` objects. To compare `Symbol` values and objects, consider the code-snippet and diagram in Figure 4.31. In the first line of code, we create a `Symbol` value, using the constructor without the `new` keyword, with `"example"` as the value of `[[Description]]`. In the second line, we create the `Symbol` object by executing a type conversion. While using the `Object` constructor with the `new` keyword creates an object, calling it as a function makes it so the argument is converted to an object by using the `ToObject` internal operation. In the diagram, we can see that the conversion to an object, creates a wrapper that stores the `Symbol` value in its `[[SymbolData]]` internal property. This wrapper is an ordinary object that also has a `[[Prototype]]` property that allows the `Symbol("xpto").toString()` expression to execute successfully.

```
1 var symValue = Symbol("example");
2 var symObject = Object(sym);
```

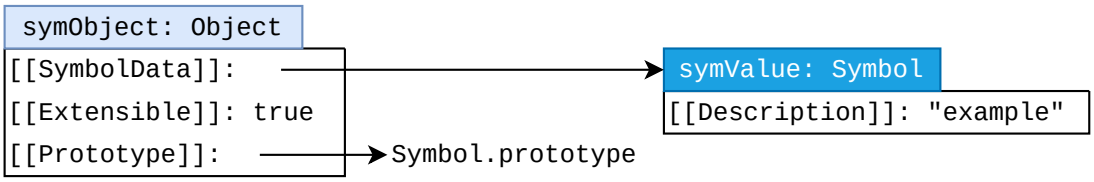


Figure 4.31: Representation of a `Symbol` object (left) and `Symbol` value (right).

Symbol Constructor The `Symbol` constructor, like all other constructors, has two important components: (1) the structure of the object that represents the constructor function with all its internal and named properties; and (2) the pseudo-code description of its behaviour. Consider the ES6 excerpt in Figure 4.32 that contains the descriptions of the `Symbol` constructor function. As we had noted, using the `new` keyword will make it so the value of `NewTarget` is not `undefined`, causing a `TypeError` exception to be thrown. Then the `description` argument is converted to a `string` value using the `ToString` internal operation. Finally, the `string` is used to create the `Symbol` value that is returned.

Although the `Symbol` constructor allows the creating of new `Symbol` values, most ECMAScript programs do not need to create new ones and just use ones that are made available through the named properties of the constructor. Consider the object diagram in Figure 4.33 where the constructor is displayed alongside some of its named properties, namely `iterator` and `toPrimitive`. These properties' descriptors are immutable, as all their properties that are not `[[Value]]` are set to `false`. Their `[[Value]]` property points to `Symbol` values that are created before any ECMAScript code is executed.

19.4.1.1 Symbol ([description])

When **Symbol** is called with optional argument *description*, the following steps are taken:

1. If NewTarget is not **undefined**, throw a **TypeError** exception.
2. If *description* is **undefined**, let *descString* be **undefined**.
3. Else, let *descString* be ToString(*description*).
4. ReturnIfAbrupt(*descString*).
5. Return a new unique Symbol value whose [[Description]] value is *descString*.

Figure 4.32: ES6 description of the `Symbol` constructor function.

There are 11 of these `Symbol` values made available through the named properties of the constructor `Symbol`. They do not allow any behavior that was not possible before, but it makes it unequivocal the type of property being accessed. For example, the `TypedArray` prototype object makes an iterator method available through the `TypedArray.prototype[Symbol.iterator]` property instead of the `TypedArray.prototype.iterator` property. But if it did use the `iterator string` value as the property key, it would still retain its functionality, changing only the way it is accessed.

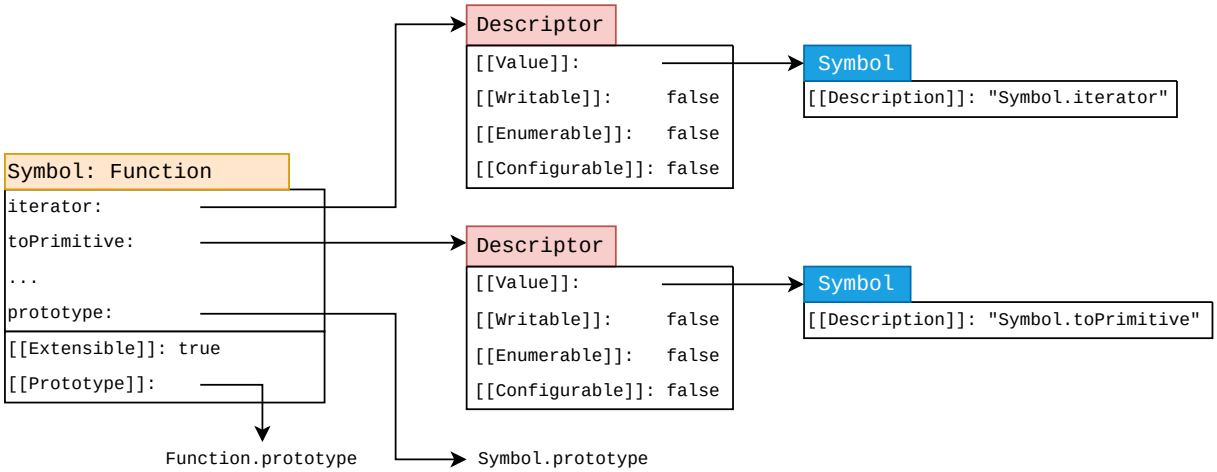


Figure 4.33: Representation of the `Symbol` constructor object.

Symbol.prototype Object Since `Symbol` values have no prototype and `Symbol` objects are only relevant in niche cases, the `Symbol.prototype` object does not have many methods. It has the three following methods:

1. `toString` - Returns the `string` value resultant from the concatentation of `"Symbol ("`, the symbol's `[[Description]]` and `")"`.
2. `valueOf` - If the `this` value is a `Symbol` object it returns `this.[[SymbolData]]`. If the `this` value is a `Symbol` value, it simply returns that value.
3. `Symbol.toPrimitive` - This method does the exact same as the `valueOf` method.

4.4.3 ECMA-SL Implementation

We are now in a position to introduce and explain our implementation of the `Symbol` built-in library in ECMARef6. First, we present the internal representation of `Symbol` values and objects. To conclude, we

explain the changes that needed to be made to the structure of ECMA-SL objects to support the use of `Symbol` values as property keys.

Symbol Values In ECMARef6, `Symbol` values are represented using ECMA-SL objects. Contrary to the ECMA-SL objects used to represent ECMAScript objects, `Symbol` values in ECMA-SL do not have a `JSProperties` property. Recall that a characteristic of `Symbol` values was that they were unique and compared through their values and not the value of their `[[Description]]`. In ECMA-SL, two `Symbol` values can be compared correctly using the `=` operator. However, it was still necessary for us to add the `ID` property to distinguish between `Symbol` values. The `ID` property holds an integer value that is different for every `Symbol` value. Why this was necessary, will be addressed later in this subsection. As an illustration of the structure just described, consider the object diagram in Figure 4.34. Here, the `Symbol` values created can be distinguished by their `ID` property, which has different values.

```
1 var symValue1 = Symbol("example");
2 var symValue2 = Symbol("example");
3 symValue1 === symValue2; //false
```

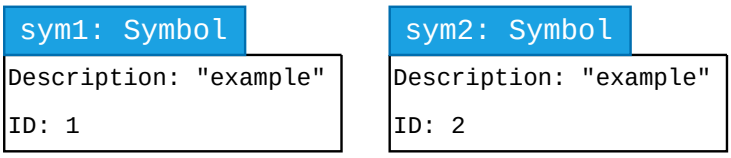


Figure 4.34: Symbol values are unique and distinguished by their `ID` property.

Symbol objects In the case of `Symbol` objects, they now have their own `JSProperties` property, although it is not populated. The object structure of `Symbol` objects can be seen in Figure 4.35, where we can see that the `SymbolData` property points to an ECMA-SL object that represents a `Symbol` value and its `Prototype` property points to the `Symbol.prototype` object.

```
1 var symValue = Symbol("example");
2 var symObj = Object(symValue);
```

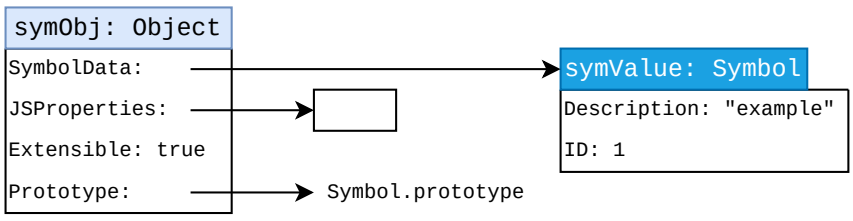


Figure 4.35: Internal representation a `Symbol` object and value.

Symbol Values As Property Keys Of particular importance is the actual use of symbols. Symbols are meant to be used as property keys alongside strings. Since the `Symbol` built-in library was introduced in ES6, the previous version of ECMARef, ECMARef5, had no support for `Symbol` values. Therefore, it was necessary to adapt the reference interpreter to accommodate this new property-key type.

ECMA-SL objects are collections of string-value pairs, meaning that it was not possible to just use the `Symbol` values as keys for our internal objects. Our first alternative was to just use the `string` value in the `Description` property of `Symbol` values. An example of this configuration can be seen in Figure 4.36 where an Object `a` and a `Symbol` value `sysValue` with `Description` `"example"` are created. In line 4, we assign the value `"b"` to object `a`, using the `string` value `"example"` as property key. The effect of

this line can be seen in the JSProperties object, where there is a property descriptor with value "b" mapped to the property example. In line 5, we do a similar process but use symValue as the property key and set the value to "c". We can see that another property descriptor with Value "c" is associated with the property Symbol(example).

```
1 var a = {};
2 var symValue = Symbol("example");
3
4 a.example = "b";
5 a[symValue] = "c";
```

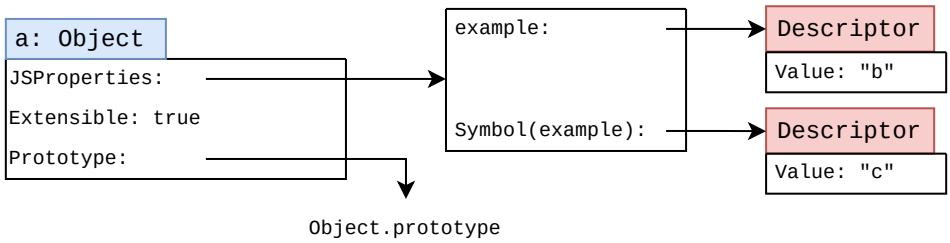


Figure 4.36: Object property assignment using string and Symbol values.

However, this represents two big problems:

1. how do we distinguish between Symbol value keys with the same Description value;
2. how can we distinguish between the string "Symbol(example)" and the a Symbol value with Description "example" when they are both used as property keys.

The example in Figure 4.37 shows this exact scenario where the assignment done in line 6, is overridden by the ones in line 7 and 8, which is not the intended behavior. The intended behavior is that the assignments in lines 6, 7 and 8 all create their own property descriptors.

```
1 var a = {};
2 var symValue1 = Symbol("example");
3 var symValue2 = Symbol("example");
4
5 a.example = "b";
6 a[symValue1] = "c";
7 a[symValue2] = "d";
8 a["Symbol(example)"] = "e";
```

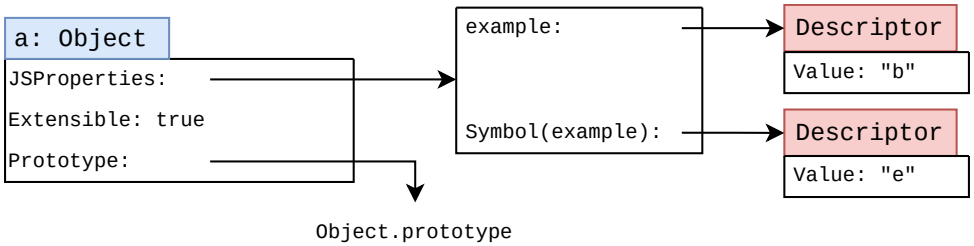


Figure 4.37: Internal property storage design that does not allow for proper integration of Symbol values as property keys.

In order to solve these issues, a new property called JSPropertiesSymbols was added to our ECMA-SL representation of ECMAScript objects. This property is meant to hold the named properties of the ECMAScript object that use Symbol value keys, while JSProperties now holds only the named properties that use string value keys. This solves the issue of using string values that match the Description of Symbol values. However, at this stage, this solution still suffers from collisions of Symbol values with identical Description values. To fix this issue, instead of the properties of the

JSPropertiesSymbols object being the Description of the symbols, they are now the Symbols' ID. The ID property of Symbol values is converted to a string and used as a key (recall that ECMA-SL objects are string-value pairs) to allow the distinction between all symbols, since each has a unique ID. This is illustrated in Figure 4.38, where the a object now has a JSPropertiesSymbols property that points to an object that maps the IDs 1 and 2 of the symbols to the appropriate property descriptors. Although the current solution is enough to handle any reading or writing of properties usingSymbol values in objects, an additional property [[SymbolKeys]] had to be added. This property allows the mapping of ID values to the correspondent Symbol values, enabling the retrieval of both the Symbol and string property keys of an object when methods like Object.keys() are called.

```
1  var a = {};
2  var symValue1 = Symbol("example");
3  var symValue2 = Symbol("example");
4
5  a.example = "b";
6  a[symValue1] = "c";
7  a[symValue2] = "d";
8  a["Symbol(example)"] = "e";
```

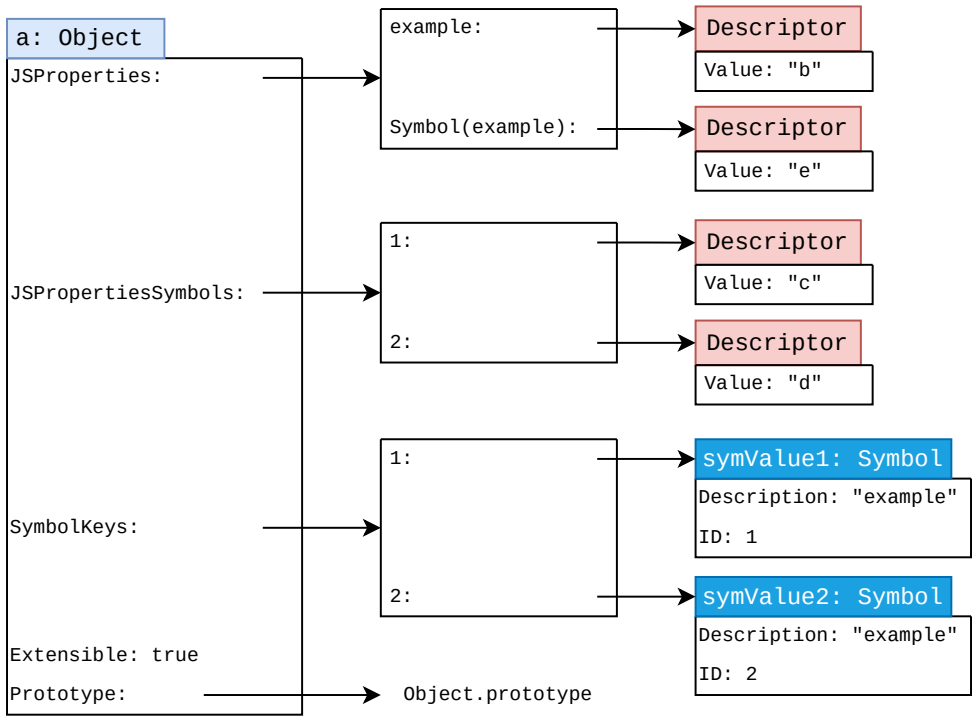


Figure 4.38: Internal property storage design that allows for proper Symbol property keys integration.

4.5 Proxy

In this section we explore the Proxy built-in library. A Proxy object is an exotic object composed of two other objects: a target and a handler. The internal methods of the Proxy object allow the partial substitution of the internal methods of the target object by ECMAScript user-code present in the methods of the handler object.

4.5.1 Examples

Like most built-ins, the entry point of the `Proxy` library is its constructor. The `Proxy` constructor takes in two arguments: a target object and an handler object. Consider the ECMAScript code in Listing 4.6 where, in the first line, a `target` object is created with the property `a` equal to `"xpto"`. The next statement is the declaration the `handler` variable which references another object. This object has a single property named `get` whose value is a function that returns the `"not xpto"` value. The next statement declares the `proxy` variable which is initialized with a `Proxy` object created using the `Proxy` constructor with the objects in the `target` and `handler` variables as arguments. The assignment in the line 9 is where we can start exploring the possibilities created by the `Proxy` library. The assignment statement does not modify the named properties of the `proxy` object, but instead those of the `target` object, meaning that after line 9 the `target` object has two named properties: `a` and `example`. So far, we have seen that the `Proxy` object can pass on its duties to its `target` object. However this does not occur when accessing the `a` property of `proxy`. In this case, the expression causes the `[[Get]]` internal method of the `Proxy` object to run, that does not have the default behavior. Instead of retrieving its named property `a` as expected, it first checks if its `handler` object has a method named `get` and if it does, the code of that method takes over the execution of the `[[Get]]` internal method of the `Proxy` object. This means that the `proxy.a` expression of the final line of the code-snippet evaluates to `"not xpto"`, as that is what the `get` method of the `handler` object returns.

Listing 4.6: Example of the use of a `Proxy` object to override the property retrieval semantics of an object.

```
1  var target = { a: "xpto" };
2
3  var handler = {
4      get: function () { return "not xpto"; }
5  };
6
7  var proxy = new Proxy(target, handler);
8
9  proxy.example = 123;
10 proxy.a;
```

4.5.2 ECMAScript Specification

Proxy Objects A `Proxy` object is an exotic object that allows the partial substitution of some of the internal methods of a given object by ECMAScript user-code. All proxy objects have two internal slots:

1. `[[ProxyTarget]]` - the object whose internal methods are to be replaced;
2. `[[ProxyHandler]]` - the object whose methods hold the ECMAScript code that is called instead of the internal methods of the target object.

Contrary to all other ECMAScript objects, `Proxy` objects do not have a `[[Prototype]]` internal property. The `[[Prototype]]` of a `Proxy` object is, by proxy, in most cases the `[[Prototype]]` of their target object. Consider the visualization of this uncommon object structure in Figure 4.39 where we can check the lack of the `[[Prototype]]` and `[[Extensible]]` internal properties as well as the lack of any named properties in the `Proxy` object.

Besides the extra internal properties of `Proxy` objects, their most important aspect is their internal methods which differ from the internal methods of ordinary objects. All these methods have the following execution pattern:

```
1  var target = { a: "xpto" };
2
3  var handler = {
4      get: function () { return "not xpto"; }
5  };
6
7  var proxy = new Proxy(target, handler);
```

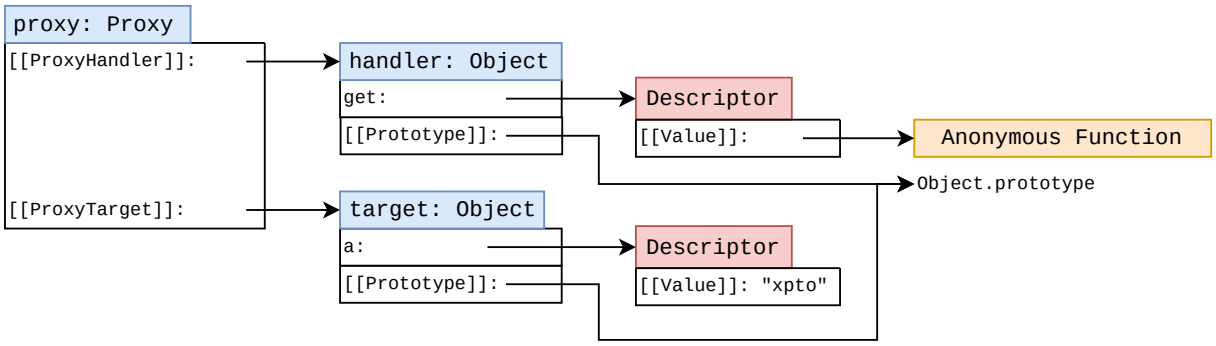


Figure 4.39: Representation of a `Proxy` object and its handler and target objects.

1. Fetch both the `[[ProxyTarget]]` and `[[ProxyHandler]]` internal properties and make sure that the handler is not `null`;
2. Attempt to retrieve the appropriate method from the handler object, for example, in the execution of the `[[Get]]` internal method, the method with key `get` is fetched;
3. If there was no method in the `handler` object, then there is no substitution to be made. In this occurrence, the `Proxy` internal method uses the `target`'s internal method and finishes execution, returning the result if necessary. For example, in the execution of the `[[Get]]` method, the `target.[[Get]]` method would be called;
4. Else, if the `handler` object possesses the appropriate method, then that method is called with the adequate arguments;
5. Since unknown ECMAScript code was executed, it is necessary to check if some applicable invariants are being upheld;
6. If all necessary invariants are upheld, then the method finishes, returning the appropriate value.

To verify this execution flow, consider Figure 4.40 that contains the ECMAScript standard's description of the `[[Get]]` internal method of `Proxy` exotic objects. Instructions 2 to 5 correspond to the first step, where the internal properties of the `Proxy` object are accessed and tested. The second step, which corresponds to the method retrieval, is done in instructions 6 and 7 using the `GetMethod` function which returns either a function or undefined. The third step is done in instructions 8 and 8.a which call the `[[Get]]` method of the target if the result of the `GetMethod` call was undefined. Instructions 9 and 10 correspond to step 4 where the handler's method is finally called. Step 5 takes place from instruction 11 to 13 where the invariants are enforced. Conveniently, in the "NOTE" section below the final instruction, the standard lists all the invariants that are being enforced in this method. Finally, the sixth step occurs in the final instruction.

Proxy Constructor Besides the ability to create `Proxy` objects, the `Proxy` constructor exposes an additional method that creates revocable `Proxy` objects, called `revocable`. While using the constructor in a typical `new Proxy(target, handler)` expression creates a regular `Proxy` object, using the

9.5.8 [[Get]] (P, Receiver)

When the [[Get]] internal method of a Proxy exotic object *O* is called with property key *P* and ECMAScript language value *Receiver* the following steps are taken:

1. Assert: IsPropertyKey(*P*) is **true**.
2. Let *handler* be the value of the [[ProxyHandler]] internal slot of *O*.
3. If *handler* is **null**, throw a **TypeError** exception.
4. Assert: Type(*handler*) is Object.
5. Let *target* be the value of the [[ProxyTarget]] internal slot of *O*.
6. Let *trap* be GetMethod(*handler*, **"get"**).
7. ReturnIfAbrupt(*trap*).
8. If *trap* is **undefined**, then
    a. Return *target*.[[Get]](*P*, *Receiver*).
9. Let *trapResult* be Call(*trap*, *handler*, «*target*, *P*, *Receiver*»).
10. ReturnIfAbrupt(*trapResult*).
11. Let *targetDesc* be *target*.[[GetOwnProperty]](*P*).
12. ReturnIfAbrupt(*targetDesc*).
13. If *targetDesc* is not **undefined**, then
    a. If IsDataDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Writable]] is **false**, then
        i. If SameValue(*trapResult*, *targetDesc*.[[Value]]) is **false**, throw a **TypeError** exception.
    b. If IsAccessorDescriptor(*targetDesc*) and *targetDesc*.[[Configurable]] is **false** and *targetDesc*.[[Get]] is **undefined**, then
        i. If *trapResult* is not **undefined**, throw a **TypeError** exception.
14. Return *trapResult*.

NOTE [[Get]] for proxy objects enforces the following invariants:

- The value reported for a property must be the same as the value of the corresponding target object property if the target object property is a non-writable, non-configurable own data property.
- The value reported for a property must be **undefined** if the corresponding target object property is a non-configurable own accessor property that has **undefined** as its [[Get]] attribute.

Figure 4.40: ECMAScript standard's description of the `[[Get]]` internal method of `Proxy` exotic objects.

`Proxy.revocable(target, handler)` method returns an object with two named properties: a property named `proxy` which contains a newly created `Proxy` object; and a property named `revoke` that contains a `Function` object with a reference to the `Proxy` object in a `[[RevocableProxy]]` internal property. Calling this function sets the value of the `[[ProxyTarget]]` and `[[ProxyHandler]]` properties of the `Proxy` object to `null`, making it unusable. Since `Proxy` objects do not have a `[[Prototype]]` internal property, the constructor does not have a `prototype` named property. This means that there is no `Proxy` prototype object either.

Advanced Example Recall the example given by Figure 4.3, where an `ArrayBuffer` object was being used incorrectly as the user was trying to manipulate the underlying `Data Block` using bracket notation which is incorrect since the `ArrayBuffer` object is not an exotic object with specialized `[[Get]]` and `[[Set]]` internal methods that allow it to be used in that way. However, as we have been discussing, the purpose of `Proxy` objects is to substitute an object's internal method with an ECMAScript function. This means that using a `Proxy` and proper handler objects, we can wrap an `ArrayBuffer` object and make it so the previous incorrect example now works properly.

That is exactly what the code-snippet and diagram in Figure 4.41 demonstrate, as we start out by creating an `ArrayBuffer` object and wrapping it with a `DataView` instance that is used in our custom `[[Get]]` and `[[Set]]` methods. In line 4, we define the function `isInteger` which determines if a `string` contains only digits. We then create the `Proxy` object in line 8, using the `ArrayBuffer` as *target* and an object created inline as *handler*. The handler object has a `get` method which is called in the execution of the `[[Get]]` internal method of the `Proxy` object. This method receives as arguments: the `target`, which is the value of the `[[ProxyTarget]]` internal property; the property name used; and a receiver which is

not relevant to this example. The method starts by determining if the property name used is a `string` containing only digits, and therefore, that can be converted to an integer. The reason why the expression `typeof p == "number"` is not used instead is because property names can only be `string` or `Symbol` values. If the property name can be converted to an integer, then it is converted and the `getInt8` method of the `DataView` object that is wrapping the `ArrayBuffer` is called to get the value. If the property name cannot be converted to an integer, then the `ArrayBuffer` is accessed using standard bracket notation on the `target`. The handler object also has a set method that is called during the execution of the `[[Set]]` internal method of the `Proxy` object. This method receives the same arguments as the get method, with the addition of the value used for the assignment. It also starts by attempting the conversion of the property name into an integer, but proceeds to call the `setInt8` method of the `DataView` object instead if it succeeds in the conversion. Should the conversion fail, then once the assignment is done in the typical fashion using bracket notation on the `target`. Finally, the method `returns true`, since `set` methods are meant to return a boolean flag to represent success. In line 22, we perform an assignment on the `Proxy` object, that triggers the call of the `[[Set]]` internal method which in turn, calls the `set` method of the handler object and write in the `Data Block` of the `ArrayBuffer`. Finally, the last two lines, test the value of the first byte of the `Data Block`. First, using the `DataView` object, which guarantees that the 100 value was written in the block and not as a named property in the `Proxy` or its `target`. Then, using the `Proxy` object itself, which confirms that the substitution for the `get` method of the handler object was successful and returned the expected value.

4.5.3 ECMA-SL Implementation

We now present the ECMARef6 implementation of the `Proxy` library. Since the `Proxy` objects and their internal methods are the most important aspect of the library, our focus will be on the implementation of those components.

Proxy Objects Most of the ECMAScript objects in ECMARef6 reference interpreter are created using the `NewECMAScriptObject` ECMA-SL function, that initializes the `JSProperties`, `JSPropertiesSymbols` and `SymbolKeys` properties. However, `Proxy` objects do not and cannot have any named properties making this initialization redundant. Therefore, `Proxy` objects are creating using an ECMA-SL object literal which creates an ECMA-SL object with no properties. This object is then populated with its internal methods and the `ProxyTarget` and `ProxyHandler` properties. Consider Figure 4.42 where it is possible to visualize our representation of `Proxy` objects. We can see that the `Proxy` object on right, lacks the usual internal properties of ECMAScript objects and that its internal methods are all different from their defaults.

Internal Methods Much like in the case of the `DataView` built-in library, the `Proxy` library does not add anything that was not previously supported by the reference interpreter. Therefore, our implementation did not require extending the ECMA-SL language or changing the structure of all ECMAScript objects, like the `Symbol` library did, which allowed us to copy the specification. As we can see in Figure 4.43, both the specification and our implementation of the `[[Enumerate]]` internal method follow the pattern mentioned above and there is a high degree of resemblance between the instructions of both of them.

```
var ab = new ArrayBuffer(4);
var dv = new DataView(ab);

function isInteger(str) {
    return typeof str == 'string' ? /^\d+$/.test(str) : false;
}

var abProxy = new Proxy(ab, {
    get: function (t, p) {
        return isInteger(p) ? dv.getInt8(parseInt(p)) : t[p];
    },
    set: function (t, p, v) {
        if (isInteger(p)) {
            dv.setInt8(parseInt(p), v);
        } else {
            t[p] = v;
        }
        return true;
    }
});

abProxy[0] = 100;
dv.getInt8(0) === 100; // true
abProxy[0] === 100; // true
```

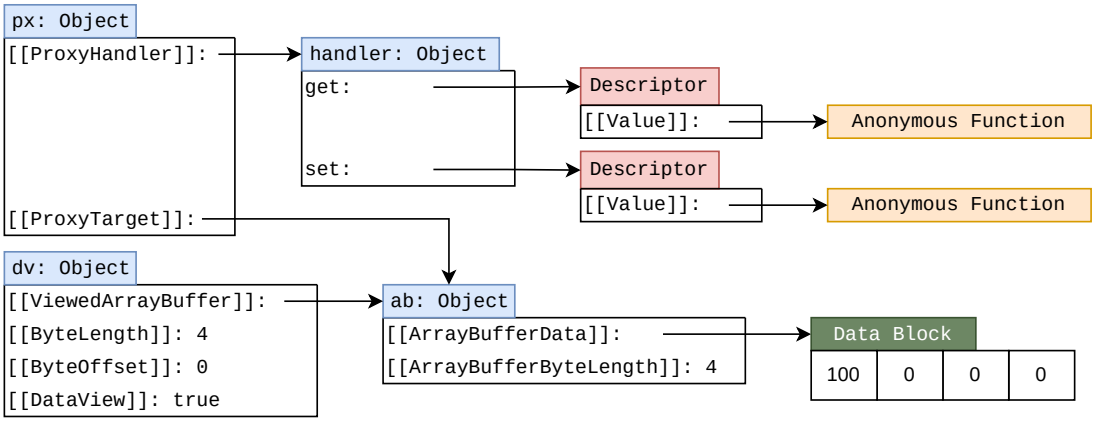


Figure 4.41: ECMAScript code implementation and diagram of an `ArrayBuffer` wrapped using a `Proxy` object to allow utilization of bracket notation to read and write bytes.

4.6 Reflect

In this section we will introduce the `Reflect` built-in library and why some of its functionality seems a bit redundant.

The `Reflect` built-in library is meant to allow ECMAScript code to more directly call internal methods of objects. While all internal methods are indirectly called via nested function calls, without the `Reflect` library it is impossible to call them in isolation. However, as we will see there are some cases where the `Reflect` library is not necessary.

4.6.1 Examples

Listing 4.7: Example of the use of the `Reflect` object.

```
var obj = {
    a: "xpto"
};

var iterator = Reflect.enumerate(obj);

var result = iterator.next();
```

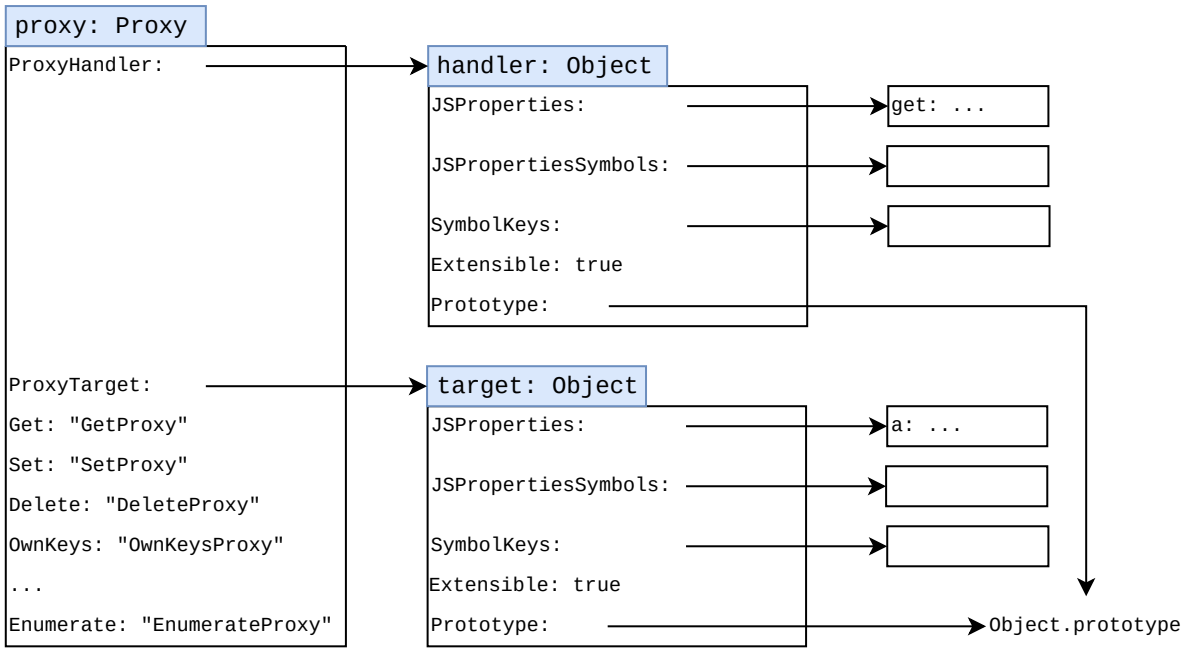


Figure 4.42: Internal representation of a `Proxy` object and its `target` and `handler` objects in ECMARef6.

4.6.2 ECMAScript Specification

In the ECMAScript specification, the `Reflect` library is accessed through the `Reflect` property of the global object. The value of this property is an ordinary object in contrast to most other built-in libraries that provide constructors. This means there are no `Reflect` objects or a `Reflect.prototype` object. As such, all the functionality of this library is accessed through the methods of `Reflect` object. Each of its methods represents one of the internal methods of objects present in the standard. Like most reflection features of other languages, these methods allow ECMAScript programs to directly call the internal methods instead of them being indirectly called via the constructs of the language. To that effect, the `Reflect` object has 14 methods, with 12 of them matching the 12 internal methods of non-function objects and the other 2 being matching the `[[Call]]` and `[[Construct]]` internal methods of functions. Table 4.4 matches all the methods of the `Reflect` object with its corresponding internal method. Much like the internal methods of `Proxy` objects, there is common pattern of behavior in the `Reflect` methods. This pattern is the following:

1. Validation of the inputs. For example, when calling `Reflect.construct` the first argument must be a constructor or a `TypeError` exception will be thrown;

2. Conversion of inputs to internal data types. For example, when calling the `Reflect.construct` method, the second argument is an `Array` of the arguments to be passed on to the constructor function. However, the `Construct` internal operation which this method calls, receives those arguments in a `List` instead of an `Array` object. Since, `List` values cannot be created by the ECMAScript program calling `Reflect.construct`, the array must be converted;

3. The internal method or an internal operation that calls the internal method is called;

4. Similarly to the inputs, the outputs of internal methods can sometimes be values of types internal to the interpreter and therefore need to be converted to ECMAScript equivalent. For example, the `[[OwnPropertyKeys]]` internal function called by the `Reflect.ownKeys` method returns a `List` value with the property keys which needs to be converted to an ECMAScript `Array` object.

9.5.11 [[Enumerate]] ()

When the [[Enumerate]] internal method of a Proxy exotic object *O* is called the following steps are taken:

1. Let *handler* be the value of the [[ProxyHandler]] internal slot of *O*.
2. If *handler* is **null**, throw a **TypeError** exception.
3. Assert: Type(*handler*) is Object.
4. Let *target* be the value of the [[ProxyTarget]] internal slot of *O*.
5. Let *trap* be GetMethod(*handler*, **"enumerate"**).
6. ReturnIfAbrupt(*trap*).
7. If *trap* is **undefined**, then
    a. Return *target*.[[Enumerate]]().
8. Let *trapResult* be Call(*trap*, *handler*, «*target*»).
9. ReturnIfAbrupt(*trapResult*).
10. If Type(*trapResult*) is not Object, throw a **TypeError** exception.
11. Return *trapResult*.

NOTE [[Enumerate]] for proxy objects enforces the following invariants:

- The result of [[Enumerate]] must be an Object.

(a) Standard description of the `Proxy.[[Enumerate]]` internal method.

```
function EnumerateProxy (O) {
    /* 1. Let handler be the value of the [[ProxyHandler]] internal slot of O. */
    handler := O.ProxyHandler;
    /* 2. If handler is null, throw a TypeError exception. */
    if (handler = 'null) {
        throw TypeErrorConstructorInternal()
    };
    /* 3. Assert: Type(handler) is Object. */
    assert(Type(handler) = "Object");
    /* 4. Let target be the value of the [[ProxyTarget]] internal slot of O. */
    target := O.ProxyTarget;
    /* 5. Let trap be GetMethod(handler, "enumerate"). */
    trap := GetMethod(handler, "enumerate");
    /* 6. ReturnIfAbrupt(trap). */
    @ReturnIfAbrupt(trap);
    /* 7. If trap is undefined, then */
    if (trap = 'undefined) {
        /* a. Return target.[[Enumerate]](). */
        return {target.Enumerate}(target)
    };
    /* 8. Let trapResult be Call(trap, handler, [target]). */
    trapResult := Call(null, null, trap, handler, [target]);
    /* 9. ReturnIfAbrupt(trapResult). */
    @ReturnIfAbrupt(trapResult);
    /* 11. If Type(trapResult) is not Object, throw a TypeError exception. */
    if (!(Type(trapResult) = "Object")) {
        throw TypeErrorConstructorInternal()
    };
    /* 12. Return trapResult. */
    return trapResult
};
```

(b) Implementation of the `Proxy.[[Enumerate]]` internal method in the reference interpreter.

Figure 4.43: Comparison between the standard description of the `Proxy.[[Enumerate]]` internal method and its implementation in the reference interpreter.

Consider the ECMAScript standard excerpt in Figure 4.44, where we can observe the aforementioned pattern in the description of the `Reflect.getOwnPropertyDescriptor` method. This method has two parameters, `target` and `propertyKey`, and its purpose is to retrieve the `property descriptor` associated with the `propertyKey` key of the `target` object. In the first line, it checks if the `target` argument is an object. If it is not an object, then a `TypeError` exception is thrown because only objects have internal methods. In the next line, the `propertyKey` argument is coerced into a `string` or `Symbol` value using the `ToPropertyKey` internal operation, since those are the types accepted by `[[GetOwnProperty]]`. In line 4, the call to `[[GetOwnProperty]]` is done and the `property descriptor` returned is stored in the

Internal method	`Reflect` method
`[[Call]]`	`apply`
`[[Construct]]`	`construct`
`[[DefineOwnProperty]]`	`defineProperty`
`[[Delete]]`	`deleteProperty`
`[[Enumerate]]`	`enumerate`
`[[Get]]`	`get`
`[[GetOwnProperty]]`	`getOwnPropertyDescriptor`
`[[GetPrototypeOf]]`	`getPrototypeOf`
`[[HasProperty]]`	`has`
`[[IsExtensible]]`	`isExtensible`
`[[OwnPropertyKeys]]`	`ownKeys`
`[[PreventExtensions]]`	`preventExtensions`
`[[Set]]`	`set`
`[[SetPrototypeOf]]`	`setPrototypeOf`

Table 4.4: Internal methods and their `Reflect` object method counterparts.

`desc` variable. This value cannot be returned as is, since ECMAScript programs have no mechanisms to deal with `property descriptors` directly. Consequently, the `FromPropertyDescriptor` internal operation is used to create an object whose named properties mimic the fields of the `descriptor` passed. This value is now a standard ECMAScript object and can be returned.

26.1.7 Reflect.getOwnPropertyDescriptor (target, propertyKey)

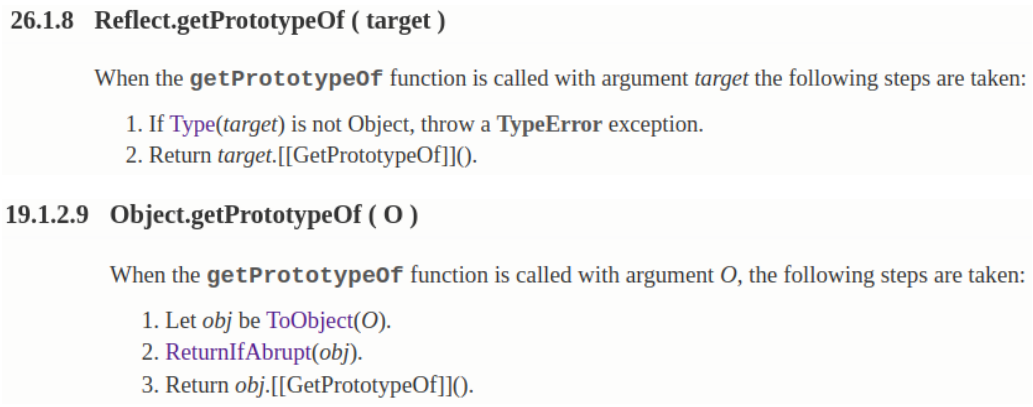
When the **`getOwnPropertyDescriptor`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. If Type(*target*) is not Object, throw a **TypeError** exception.
2. Let *key* be ToPropertyKey(*propertyKey*).
3. ReturnIfAbrupt(*key*).
4. Let *desc* be *target*.[[GetOwnProperty]](*key*).
5. ReturnIfAbrupt(*desc*).
6. Return FromPropertyDescriptor(*desc*).

Figure 4.44: ECMAScript standard's description of the `Reflect.getOwnPropertyDescriptor` method.

Duplicated Behavior With the introduction of the `Reflect` library, some of the functionality already present in the standard, was made redundant or repeated. When of those cases is the `getPrototypeOf` method of the `Reflect` object. As we can see in Figure 4.45, the equivalent method for the `Object` built-in object is almost identical. The only difference being that the `Reflect` version will throw a `TypeError` exception if the argument is not an object and the `Object` version will coerce the argument into an object instead. They both then return the result of calling the `[[GetPrototypeOf]]` internal method.

The reason why we consider at least one of these methods redundant is because a simple ECMAScript instruction before a call to any of these methods would equalize their functionality. One could argue that the existence of methods this similar could even be a source of confusion for developers. Listing 4.8 and Listing 4.9, demonstrate how with an extra line of ECMAScript code it is possible to obtain



26.1.8 Reflect.getPrototypeOf (target)

When the **`getPrototypeOf`** function is called with argument *target* the following steps are taken:

1. If Type(*target*) is not Object, throw a **TypeError** exception.
2. Return *target*.[[GetPrototypeOf]]().

19.1.2.9 Object.getPrototypeOf (O)

When the **`getPrototypeOf`** function is called with argument *O*, the following steps are taken:

1. Let *obj* be ToObject(*O*).
2. ReturnIfAbrupt(*obj*).
3. Return *obj*.[[GetPrototypeOf]]().

Figure 4.45: ECMAScript Standard specification of the `getPrototypeOf` method of the `Reflect` and `Object` built-in objects.

the behavior of one method while using the other one. In the first one, the use of the `Object` constructor without the `new` keyword is used to perform a type conversion and guarantee that the value of "obj" is an object. In the second snippet, we perform a type check using the `typeof` operator to guarantee that `Object.getPrototypeOf` is never called with a non-object argument. The `getPrototypeOf` method is used as an example here, but there are other methods that fall into the same pattern.

Listing 4.8: Equivalent to the `Object.getPrototypeOf` method.

```
1 var obj = Object(notObj);
2 Reflect.getPrototypeOf(obj);
```

Listing 4.9: Equivalent to the `Reflect.getPrototypeOf` method.

```
1 if (typeof obj != "object") throw TypeError("Must be an object");
2 Object.getPrototypeOf(obj);
```

Regardless of how redundant these methods may or may not be, after considering the end goal of the `Reflect` built-in library and some constraints of the standard as a whole it is possible to understand that the existence of these almost repeated methods is mandatory.

Considering that the objective of the `Reflect` library is to provide a access to an object's internal methods, the most obvious way to supply that functionality is by mapping each internal method to a method of an object accessible at the JavaScript level, in this case, the `Reflect` object. This is the route taken by the authors of the ECMAScript standard and Table 4.4 shows how the methods were mapped. However, applying this thought process in a consistent manner, does mean that some functionality will end up being duplicated.

Why not add the `Reflect.getPrototypeOf` method and remove the `Object.getPrototypeOf` then? This would solve the issue of duplicated functionality, however, as we mentioned in Section 2.2, there is a high priority attributed to maintaining backwards compatibility when expanding the ECMAScript standard. Since the `Object` built-in library existed in ES5 with all the conflicting methods and the `Reflect` built-in library was a new addition to ES6, this sort of change would make all ECMAScript code that used the `Object` built-in methods fail when executed.

4.6.3 ECMA-SL Implementation

We can now present our implementation of the `Reflect` built-in library in ECMARef6. Much like the other libraries we implemented, here we were able to maintain the similarity between the specification and the implementation.

Consider Figure 4.46 where we compare our implementation of the `Reflect.getOwnPropertyDescriptor` method to the specification. In the ECMA-SL implementation, the arguments have to be retrieved from the `args` list using the `l_nth` operator, as we had discussed before. The other significant difference occurs in the call of the internal method. In ECMA-SL, the `this` value in methods is not implicit and therefore must be passed as the first argument.

26.1.7 Reflect.getOwnPropertyDescriptor (target, propertyKey)

When the **`getOwnPropertyDescriptor`** function is called with arguments *target* and *propertyKey*, the following steps are taken:

1. If Type(*target*) is not Object, throw a **TypeError** exception.
2. Let *key* be ToPropertyKey(*propertyKey*).
3. ReturnIfAbrupt(*key*).
4. Let *desc* be *target*.[[GetOwnProperty]](*key*).
5. ReturnIfAbrupt(*desc*).
6. Return FromPropertyDescriptor(*desc*).

```
function ReflectGetOwnPropertyDescriptor(global, this, NewTarget, strict, args) {
  target := l_nth(args, 0);
  propertyKey := l_nth(args, 1);
  /* 1. If Type(target) is not Object, */
  if (!(Type(target) = "Object")) {
    /* throw a TypeError exception */
    throw TypeErrorConstructorInternal()
  };
  /* 2. Let key be ToPropertyKey(propertyKey). */
  key := ToPropertyKey(propertyKey);
  /* 3. ReturnIfAbrupt(key). */
  @ReturnIfAbrupt(key);
  /* 4. Let desc be target.[[GetOwnProperty]](key). */
  desc := {target.GetOwnProperty}(target, key);
  /* 5. ReturnIfAbrupt(desc). */
  @ReturnIfAbrupt(desc);
  /* 6. Return FromPropertyDescriptor(desc). */
  return FromPropertyDescriptor(desc)
};
```

Figure 4.46: ECMAScript standard's description and ECMA-SL implementation of the `Reflect.getOwnPropertyDescriptor` method.

4.7 Other Built-in Libraries

My role in the development of ECMARef6 was to coordinate the implementation of all the built-in libraries. In that capacity, I was in charge of their evaluation and correction. Meaning that I tested and corrected bugs across **all** the built-in libraries, whose code totals 26275 LoC.

Besides the built-in libraries described in this chapter, I was the main developer of the `Reflect` library. The `Reflect` library exposes no constructors or prototypes, just an ordinary object. This object has one method for each of the internal methods present in ECMAScript objects. Every one of these methods receives an object as its first argument and calls its internal method that matches the `Reflect` method. For instance, the expression `Reflect.getPrototypeOf(new Int8Array())` would call the `[[GetPrototypeOf]]` internal method of the `Int8Array` object passed as argument and return the value obtained. Since objects are the only type with internal methods, if a value of another type is used as the first argument, then the method call will throw a `TypeError` exception. I did not include a

description of the implementation of this library in this thesis due to time constraints.

Chapter 5

Evaluation

This chapter presents the evaluation of our implementation of the ES6 built-in libraries. In a nutshell, we evaluate our implementation by testing it against Test262, the ECMAScript official test suite. Even though we focus the evaluation on the built-in libraries discussed in Chapter 4, we present our testing results for all built-in libraries.

5.1 Test262

The evaluation process of the ECMARef6 reference interpreter is straightforward given the existence of Test262 [11]. Since our main goal is to conform to the ECMAScript standard and the test suite's purpose is to test the conformity of an implementation to the standard, we can evaluate the extent and correctness of our implementation quantitatively, by checking the number of passing tests for each implemented library and contrasting that with the total number of tests for that library.

The tests that compose the Test262 test suite are JavaScript files with a set of instructions and the necessary assertions to verify that the state produced by the execution matched the tests' expectations.

Figure 5.1 is an example of a test file, which we can see is just a JavaScript file. It is also important to note the three distinct sections of the test file: the *copyright* section which has information related to the authors of the test; the *metadata* section where some important characteristics of the test are defined; and the *body* section where the JavaScript code resides. When it comes to ES6 tests, the metatadata section has a key-value structure with following keys:

- `es6id`: this value refers to the section of the standard targeted by the test;
- `description`: a succinct description of the feature being tested;
- `info`: information about the specific pseudo-code instruction of the standard that captures the core functionality being tested;
- `includes`: a collection of JavaScript files that need to be evaluated before executing the test code;
- `features`: the features of the standard that are being tested.

For instance, the test given in Figure 5.1 tests the behavior defined in Section 22.2.2.2 of the standard, which defines the `%TypedArray%.of` function. More concretely, the instruction in the 4th line is supposed to throw a `TypeError` exception since ""of" cannot be invoked as a function". On line 20, there is an `assert` object and a call to its `throws` method neither of which is defined in the standard and therefore should not be accessible since they are not defined before line 20. This means that they are defined

```
1  // Copyright (C) 2016 the V8 project authors. All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3  /*---
4  es6id: 22.2.2.2
5  description: >
6    "of" cannot be invoked as a function
7  info: |
8    22.2.2.2 %TypedArray%.of ( ...items )
9
10   ...
11   3. Let C be the this value.
12   4. If IsConstructor(C) is false, throw a TypeError exception.
13   ...
14 includes: [testTypedArray.js]
15 features: [TypedArray]
16 ---*/
17
18 var of = TypedArray.of;
19
20 assert.throws(TypeError, function() {
21   of();
22 });
```

Figure 5.1: Example of a test file of the Test262 suite.

somewhere else, more concretely, the Test262 *harness*. The harness is a collection of JavaScript files composed of function and variable definitions, some of which must be executed before the test, more specifically the ones mentioned in the `includes` value of the `metadata` section.

5.1.1 Test Selection

Although, we use the Test262 suite to perform our evaluation, some of its tests are meant to target features of the newer versions of ECMAScript. Naturally, some tests targeting ES12, the newest version of the standard, are expected to fail when run against ECMARef6. Including these tests in our evaluation would pollute our results and prevent us from getting a clear idea of how well our implementation performed relative to the version of the standard that we target. This means that in order to get a correct assessment of the state of our implementation we need to filter out a portion of these tests.

Selecting tests is not trivial because not all of them come annotated with a flag that indicates their version. Up to 2016, tests included a flag indicating whether they target version 5 (`es5id`) or version 6 (`es6id`). These flags have deprecated. Hence, if a test comes with either the `es5id` or the `es6id` flags, then it should be included. However, the opposite does not hold. There might be tests without these flags that should also be included. The test filtering problem is a highly complex problem that cannot be systematically addressed in the context of this thesis. Our solution was to filter the unlabeled tests manually. More concretely, our selection methodology for unlabeled tests was simple: when testing our implementation, tests that failed were analyzed to determine the source of the error. If the error source was a missing feature that the test case expected to be present, but was only introduced in an ECMAScript version posterior to ES6, then the test was discarded as it was not applicable to ECMARef6. In theory, this approach means that although we discard some tests relative to the built-in libraries, these target more recent versions of the ECMAScript standard and our selection still contains all the tests meant for the 6th version.

Table 5.1 presents the results for the test selection process. For each built-in library, we show its corresponding section in the standard, the number of tests in the current test suite and the number of selected tests. It is worth noting that there is a smaller selection ratio in the `TypedArray`, `ArrayBuffer` and `DataView` sections relative to the other ones. This is due to the introduction of the `BigInt` primitive type in ES11 that added the element types `Int64` and `Uint64` to byte-level operations. This resulted in the addition of new `TypedArray` constructors and `DataView` methods that needed to be tested.

ES6 Section	# of tests in the Test262 suite	# of tests selected
18 - 26	22996	13253
19.4 (Symbol)	92	81
22.2 (TypedArray)	2012	962
24.1 (ArrayBuffer)	150	79
24.2 (DataView)	505	327
26.1 (Reflect)	153	152
26.2 (Proxy)	311	259

Table 5.1: Number of tests selected for each built-in library.

5.2 Evaluation Pipeline

The test execution pipeline is similar to the one explained in section 2.1 except that the JavaScript file would be the test that we intend to run and the output of the `CoreECMA-SL` interpreter needs to be evaluated to determine the outcome of the test. There are 4 possible test outcomes which are determined by the exit code of the `CoreECMA-SL` interpreter:

- 0: **Ok** - the test passed;
- 1: **Fail** - the test failed because some of the assertions made in the test file were not true;
- 2: **Error** - the test failed because there was an internal error in the ECMARef6 interpreter, such as accessing a property of an undefined value or calling an internal function that does not exist or with the incorrect number of arguments;
- 3: **Unsupported** - the test failed because it requires some feature which is currently not implemented, such as one of the built-in libraries or a language feature like template literals, and so is expected to fail.

As discussed in Section 5.1, the Test262 harness must be executed before the body of the test so that the auxiliary testing functions can be defined. To fulfill this requirement we simply prepend the harness's code to the test's code.

Putting it all together, our testing pipeline, illustrated in Figure 5.2, starts with the concatenation of the harness and test to be executed. That JavaScript file will then be parsed and compiled to ECMA-SL so that the ECMARef6 interpreter can be imported into it, creating the out.esl file of the diagram. This file is then compiled to CoreECMA-SL using the ECMA-SL2CoreECMA-SL tool, so that the code can be evaluated by the `CoreECMA-SL` interpreter and test's outcome determined.

5.3 Results

Considering that the measure we are using to evaluate the results obtained during this thesis is the conformity to the ECMAScript standard using the Test262 test suite, the various test outcomes described in Section 5.2 can be considered irrelevant, as fail, error and unsupported test results all represent the same result in the evaluation context, that the interpreter does not conform.

Grouping all the negative test results and looking specifically at the tests that are related to the built-in libraries, we can do a proper assessment of the work performed during this thesis. Consider Table 5.2 that summarizes the results across all the built-in libraries of the ECMAScript standard and the ones where I was personally involved. Here we can observe that although we employ strategies to guarantee

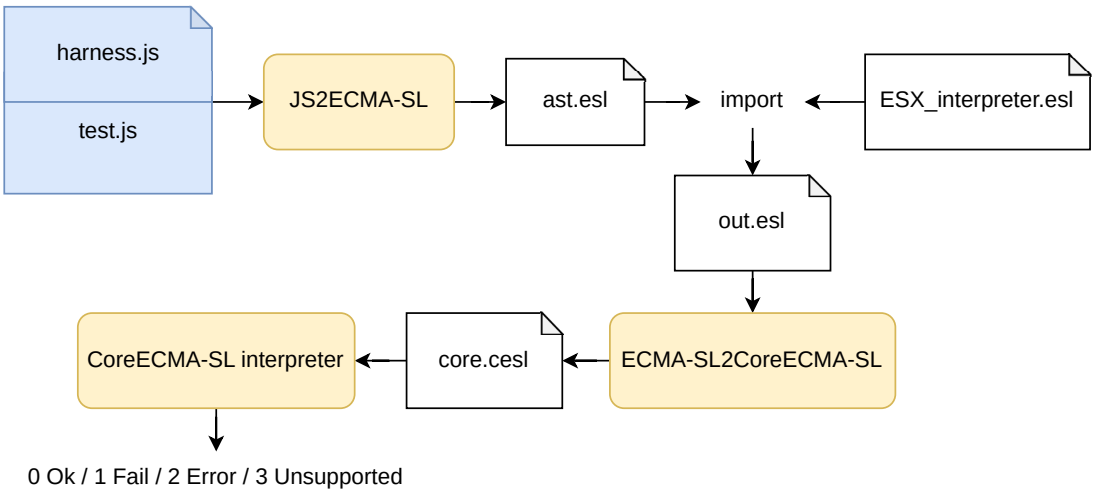


Figure 5.2: Test execution pipeline.

that the reference interpreter is as close to the specification as possible, there are still some errors present. It is important to split these errors into categories based on where they occurred. There are 3 possible categories they can be placed into:

1. The error comes from a flaw in the implementation of the built-in library being tested. This type of error is the least common, but the most serious.
2. The error comes from a flaw in the implementation of one of the dependencies of the built-in library being tested. For instance, a `DataView` test that fails due to a flaw in the `ArrayBuffer` implementation.
3. The error comes from the lack of an implementation of one of the dependencies of the built-in library being tested.

The reasons for these errors are the following:

- `Symbol` - The errors in the `Symbol` tests fall into the 3rd category. These stem from the use of ECMARef5 code inside of ECMARef6. ES6 introduced the concept of `Realm` which is something that encapsulates all ECMAScript code. It contains the global object and the language's intrinsic objects. The tests that are failing test if the different `Symbol` values that are named properties of the `Symbol` constructor are equal across all realms. Since they are not yet supported by ECMARef6, the tests fail;
- `ArrayBuffer` - The first error is equal to the `Symbol` errors, suffering from the lack of the `Realm` implementation. The other test falls into the 2nd category and fails because there is an error in the implementation of the constructor of ECMAScript classes. Classes that extend other classes but do not define their own constructor. For example, in `class A extends B{}` class A gets a default constructor that simply passes all the arguments it receives to the constructor of B. The default constructor function in this scenario currently does not provide the appropriate arguments to the constructor of the `super` class it extends. Using the constructor with the expression `new A()`, would make it call the B constructor with one argument with value `undefined`, the equivalent of `B(undefined)`, instead of using no arguments;
- `DataView` - There are two separate reasons why some of the `DataView` tests fails: (1) one of the tests fails because of the lack of a `Realm` implementation; (2) the other 2 tests fail because there is

an error in the representation of the `NaN` value in ECMA-SL. These 2 tests are the only ones that fail because of errors in the implementation done in the context of this thesis.

- `TypedArray` - In the case of the `TypedArray` tests, the errors come from the `NaN` and `Realm` issues mentioned above;

- `Reflect` - The `Reflect` tests that are failing have to do with the way properties are defined in ECMA-SL objects. Internal methods of ECMAScript objects like `[[OwnKeys]]`, should return the keys sorted by creation order. ECMA-SL objects do not keep track of the creation of their properties and so the test cannot pass. Currently, this functionality is being added to the ECMA-SL representation of ECMAScript objects by another student;

- `Proxy` - The `Proxy` tests are all failing because of different reasons: (1) one test uses a `Bound Function` which are not yet completely implemented; (2) another test uses an `import` statement which the reference interpreter does not yet support; (3) another test uses a `generator` function which the reference interpreter does not yet support; (4) and the final test fails because of the lack of property order preservation, as in the `Reflect` tests.

Section	# of tests	Passed	Failed	Passed Percentage
19.4 (Symbol)	81	68	13	83.95%
22.2 (TypedArray)	959	942	17	98.22%
24.1 (ArrayBuffer)	79	77	2	97.46%
24.2 (DataView)	327	324	3	99.08%
26.1 (Reflect)	152	149	3	98.02%
26.2 (Proxy)	259	255	4	96.53%
19.1 (Object)	2942	2926	16	99.46%
19.2 (Function)	399	378	21	94.74%
19.3 (Boolean)	51	51	0	100.00%
19.5 (Error)	41	41	0	100.00%
20.1 (Number)	348	303	45	87.07%
20.2 (Math)	341	337	4	98.83%
20.3 (Date)	750	741	9	98.80%
21.1 (String)	1014	973	41	95.96%
21.2 (RegExp)	1410	885	525	62.77%
22.1 (Array)	2701	2675	26	99.04%
23.1 (Map)	156	154	2	98.72%
23.2 (Set)	197	196	1	99.49%
23.3 (WeakMap)	93	91	2	97.85%
23.4 (WeakSet)	79	78	1	98.73%
24.3 (JSON)	150	138	12	92.00%
25.1 (Iteration)	4	4	0	100.00%
25.2 (GeneratorFunction)				Not Implemented
25.3 (Generator)				Not Implemented
25.4 (Promise)	384	375	9	97.66%
26.3 (Module Namespace)				Not Implemented
Total	13253	12457	793	93.99%

Table 5.2: Test results of the built-in libraries.

5.4 Evaluation short-comings

Besides the goal of our interpreter conforming the ECMAScript standard, there is also the goal of it being capable of being used as a reference. To evaluate our success in the pursuit of this goal, it would be necessary to compare the written ECMA-SL code to the original standard and determine a level of closeness. With a tool meant to convert between the two mediums (that we initially proposed to build in this thesis), this kind of comparison might have been possible at a quantitative level and given us more metrics to gauge the value of the work done. However, given that the goal of the thesis changed after the proposal, the tool never materialized and therefore our evaluation can be considered shallow or insufficient. Although we do not have the tools to evaluate this aspect, we still believe that our line-to-line matching strategy between ECMA-SL statements and standard pseudo-code instructions would have guaranteed a satisfying result.

Chapter 6

Conclusions

In this thesis we have worked in the context of the ECMA-SL project with the goal of extending its more up-to-date reference interpreter (ECMARef6) with support for the built-in libraries of the 6th version of the ECMAScript standard. This was done using the ECMA-SL language which was specifically designed to be similar or identical to the standard's pseudo-code. This similarity allowed us to use a line-by-line strategy, where we matched each pseudo-code instruction of the specification with an ECMA-SL statement in the implementation. This strategy gives us confidence that our implementation can be used as a reference for the ECMAScript standard, and serve as its own executable specification.

As the complexity of the ECMAScript standard increases, it becomes progressively more relevant the existence of a complete reference interpreter that can be used to reason about other implementations and as a testing mechanism. We believe ECMARef6 to be the reference interpreter with the most complete implementation of the built-in libraries of the standard, making its use as a testing oracle possible, since the built-in libraries are a large part of the ECMAScript language and most ECMAScript programs use them during their execution.

In the journey to attaining we had to extend the ECMA-SL language itself and change fundamental design decisions related to the core representation of ECMAScript objects. More concretely, we needed to extend the ECMA-SL language with two types, byte and array, and operators to create and manipulate them. This was necessary as with the previous version of the ECMA-SL language, it was impossible to represent the `Data Block` type introduced with the `ArrayBuffer` library. In the implementation of the `Symbol` library, we had to update the internal model of ECMAScript objects to support the use of `Symbol` values as property keys. The most impactful of these changes is the latter, but since there are abstractions set in place when accessing the named properties of objects, it was possible to avoid changing the entire ECMARef6 codebase. However, this approach may have to be revised if other types of property keys are added in further versions.

Future Work As a continuation of the work done in this thesis, future work could be done to complete the implementation of the ECMARef6 reference interpreter. Core language functionality is not yet implemented, such as execution context switching, which is required for the implementation of the `Generator` and `GeneratorFunction` built-in libraries. Having a complete reference interpreter opens up a great many number of other possibilities for future work. The following are examples of possible projects:

1. A tool capable of generating the HTML document corresponding to the specification using the reference interpreter's code;

2. A tool with the inverse function can be done. It would use the specification to generate a reference implementation. A tool that was actually able to perform this function at a high level would signif-

icantly reduce the implementation time of future reference implementations, such as ECMARef7, ECMARef8, etc;

3. With a complete reference interpreter one could employ automatic test generation techniques to automatically create a conformance test suite, that could potentially complement Test262 as the official test suite of the standard.

Even without an automatic translation tool to generate reference implementations, future work could still be done to keep updating the current reference interpreters to the more recent versions of the standard, even if by hand.

Bibliography

[1] "Node.js." https://nodejs.org/en/. Accessed on 2022-01-14.

[2] "Electron — build cross-platform desktop apps with javascript, html, and css." https://www.electronjs.org/. Accessed on 2022-01-14.

[3] "Discord — your place to talk and hang out." https://discord.com/. Accessed on 2022-01-14.

[4] "Visual studio code - code editing. redefined." https://code.visualstudio.com/. Accessed on 2022-01-14.

[5] "V8 - google's open source high-performance javascript and webassembly engine, written in c++." https://v8.dev. Accessed on 2022-10-30.

[6] "Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet," *Press Release*, Dec 1995.

[7] "Ecma international." https://www.ecma-international.org/. Accessed on 2022-12-02.

[8] "Ecmascript® 2015 language specification, 6th edition / june 2015." https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf. Accessed on 2022-10-31.

[9] "Tc39 - specifying javascript." https://tc39.es/. Accessed on 2022-10-30.

[10] "The tc39 process." https://tc39.es/process-document/. Accessed on 2022-10-30.

[11] "Test262 - official ecmascript conformance test suite." https://github.com/tc39/test262. Accessed on 2022-10-30.

[12] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," vol. 49, pp. 87–100, 01 2014.

[13] A. Guha, C. Saftoiu, and S. Krishnamurthi, "The essence of javascript," pp. 126–150, 06 2010.

[14] P. Gardner, S. Maffeis, and G. Smith, "Towards a program logic for javascript," vol. 47, pp. 31–44, 01 2012.

[15] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi, "A tested semantics for getters, setters, and eval in javascript," *SIGPLAN Not.*, vol. 48, p. 1–16, oct 2012.

[16] D. Park, A. Ştefănescu, and G. Roşu, "Kjs: A complete formal semantics of javascript," *ACM SIGPLAN Notices*, vol. 50, pp. 346–356, 06 2015.

[17] J. Fragoso Santos, P. Maksimović, D. Naudziuniene, T. Wood, and P. Gardner, "Javert: Javascript verification toolchain," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 1–33, 12 2017.

[18] A. Charguéraud, A. Schmitt, and T. Wood, "Jsexplain: A double debugger for javascript," pp. 691–699, 04 2018.

[19] L. Loureiro, "Ecma-sl - a platform for specifying and running the ecmascript standard," Master's thesis, Instituto Superior Técnico, July 2021.

[20] D. Gonçalves, "A reference implementation of ecmascript built-in objects," Master's thesis, Instituto Superior Técnico, October 2021.

[21] F. Quinaz, "Precise information flow control for javascript," Master's thesis, Instituto Superior Técnico, July 2021.

[22] "Ecmascript® language specification, 5.1 edition / june 2011." `https://www.ecma-international.org/wp-content/uploads/ECMA-262_5.1_edition_june_2011.pdf`. Accessed on 2022-10-31.

[23] "Ocaml - general-purpose, multi-paradigm programming language." `https://ocaml.org/`. Accessed on 2020-06-07.

[24] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[25] S. Maffeis, J. Mitchell, and A. Taly, "An operational semantics for javascript," pp. 307–325, 12 2008.

[26] "Coq - interactive formal proof management system." `https://coq.inria.fr`. Accessed on 2022-10-30.

[27] "K - rewrite-based executable semantic framework." `https://kframework.org/`. Accessed on 2022-10-30.

[28] J. Park, J. Park, S. An, and S. Ryu, "JISET: javascript ir-based semantics extraction toolchain," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 647–658, 2020.

[29] P. Gardner, G. Smith, C. Watt, and T. Wood, "A trusted mechanised specification of javascript: One year on," vol. 9206, pp. 3–10, 07 2015.