A Faithful Description of ECMAScript Algorithms

Adam Khayam, Louis Noizet, Alan Schmitt

A Faithful Description of ECMAScript Algorithms

Adam Khayam
adam.khayam@inria.fr
INRIA
Rennes, France

Louis Noizet
louis.noizet@irisa.fr
Université Rennes 1
Rennes, France

Alan Schmitt
alan.schmitt@inria.fr
INRIA
Rennes, France

ABSTRACT

We present an ongoing formalization of algorithms of ECMAScript, the specification describing the semantics of JavaScript, in a tiny functional meta-language. We show that this formalization is concise, readable, maintainable, and textually close to the specification. We extract an OCaml interpreter from our description and run small JavaScript programs whose semantics is based on these algorithms.

CCS CONCEPTS

• **Theory of computation → Operational semantics**; **Functional constructs**.

KEYWORDS

ECMAScript, Skeletal Semantics, Functional programming, Monads

ACM Reference Format:
Adam Khayam, Louis Noizet, and Alan Schmitt. 2022. A Faithful Description of ECMAScript Algorithms. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), September 20–22, 2022, Tbilisi, Georgia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3551357.3551381

1 INTRODUCTION

Mechanizing the semantics of complex languages is a convenient way to make their semantics precise. It also opens the door to manipulations of the semantics, for instance, to derive an interpreter or to study properties of the language. The success of such a mechanization relies on a crucial question: is the actual semantics of the language really captured?

There are two main ways to ensure this is the case: first, the mechanization can be textually close to the specification of the language, if it exists. Second, if the mechanization is executable, it should successfully run tests provided by the language designers. Thus, a suitable tool for mechanizing semantics should make it easy to stay close to language definitions, whether algorithmic (e.g., for JavaScript) or based on inference rules. Furthermore, the tool should also provide an easy way to derive an executable version of the semantics to run code, including tests.

The goal of this paper is to show that the Skeletal Semantics framework [3, 15, 16] is suitable for mechanizing semantics. To this end, we describe the ongoing formalization of ECMAScript algorithms as a Skeletal Semantics. We show that the resulting description is very close to the specification. In addition, we have formalized enough algorithms to be able to run simple JavaScript programs and thus test our approach.

The project is the first formalization of a real-world programming language in Skel, the meta-language used to write Skeletal Semantics. This work has been a strong motivation in the evolution of Skel, from the introduction of the notion of polymorphism to the addition of monads and first-class functions. This evolution is motivated by the need for this meta-language to be expressive enough to be able to capture the behavior of ECMAScript algorithms, while retaining a visual match between the formalization and the specification.

We present existing approaches to mechanizing languages in Section 2. We then introduce Skeletal Semantics as well as its concrete syntax in Section 3. Our main contribution is the description of the mechanization of a core ECMAScript algorithm in Section 4, where we also present some crucial features of Skeletal Semantics which significantly improve the readability of the mechanization. We conclude and explore future work in Section 5.

2 CONTEXT

We first motivate why we chose JavaScript (JS) to evaluate the mechanization of a language using Skeletal Semantics. The three defining features of JavaScript in this regard are the following. First, it is complex, hence a good candidate to see if our solution scales up. Second, it has a precise specification, called ECMAScript (ES), and an extensive suite of test cases; hence we do not have to guess what its semantics is. And third, it has been mechanized in several frameworks, which facilitates the comparison to other approaches.

There has been a previous experience of mechanizing JS in Coq, called JSCert [1]. Defining semantics in a proof assistant is the most direct approach. It has a major drawback, however. If the design choices are not correct, one cannot manipulate the mechanization to modify it, and one must instead redo it using different choices. JSCert is a pretty-big-step [6] Coq definition of ECMAScript 5.1. It consists of an inductive definition, a recursive definition, and a correctness proof showing they match. The goal of the inductive definition is to prove properties of the language and of JS programs, whereas the recursive definition can provide an OCaml interpreter using Coq's extraction mechanisms. The mechanization is reasonably close to the specification for people who can read Coq code, and the test suite can be run using the extracted interpreter. JSCert has two flaws, unfortunately. It is difficult to maintain, as one has to update two formalizations and a proof when ES changes. In addition, and most importantly, JSCert uses Coq's induction to represent the recursive evaluation of the language. This shallow embedding results in a formalization of the semantics that is an inductive definition consisting of about 1000 rules, which is too

PPDP 2022, September 20–22, 2022, Tbilisi, Georgia

ACM ISBN 978-1-4503-9703-2/22/09…$15.00
https://doi.org/10.1145/3551357.3551381

large to prove properties of the language. Some authors of JSCert built upon this work to derive systematic and reusable definition of semantics [3, 4].

Another approach to the formalization of JavaScript is to translate it to a much simpler language, whose semantics is less complex. This is done in the setting of [9], where JavaScript programs are translated to a simple language called λ_{JS}. This work is based on ECMAScript versions 3 and 5. The authors empirically show that the translation of JS programs is correct by running the JS test suite. As this formalization is not textually close to the specification, it is difficult to assess whether it actually captures the language beyond running the tests. In addition, the translation has not been updated since 2015. In particular, it does not include the many changes to the core data structures and algorithms from ECMAScript 6 (or ECMAScript 2015). Attempts to formalize λ_{JS} in Coq[1] uncovered several issues with the desugaring process that were not witnessed by testing. It did not lead to a formal mechanization of the language.

Alternatively, one may use an existing framework designed to describe and manipulate semantics. $\mathbb{K}_{\text{JS}}$ [17] is a complete mechanization of ECMAScript 5.1 in the $\mathbb{K}$ framework. It provides an executable interpreter of JS directly from the semantics with no additional effort. This framework is not suitable for analyzing the language itself as it only provides tools to reason about the execution of programs. In addition, there is no evidence that the mechanization can be easily maintained: although JS has significantly evolved since ES 5.1 (the specification has more than doubled in size), the mechanization has not been updated. Indeed, works such as JSExplain [5], a tool to visualize the step-by-step execution of JS programs, show that updating an ES formalization is far from being a trivial issue. The power of the $\mathbb{K}$ framework comes at the cost of additional complexity in describing languages, which hinders maintainability and closeness to the specification.

After describing our approach, we further detail the comparisons with JSCert and $\mathbb{K}_{\text{JS}}$ in Section 4.7.2.

3 SKELETAL SEMANTICS

Skeletal semantics is a *syntax* to define the semantics of programming languages in a concise yet powerful way, with a light formalism. This provides a way to easily manipulate the semantics, for instance, to convert it into a Coq formalization or an OCaml interpreter. One of the strengths of Skeletal Semantics is the possibility of leaving some constructions undefined to let them be implementation dependent or for gradual specification. Although the theoretical concept behind Skeletal Semantics has already been presented [3], the version we show in this paper has been significantly improved.

The Skel language, defined to describe skeletal semantics, serves as a support for the necro ecosystem [14], which provides, among other things, a generator of OCaml interpreters [2], a generator of Gallina formalizations [13], and a debugger [12].

The Skel formalization of JS, which constitutes the main topic of this article, has led us to propose several improvements to Skel, such as adding polymorphism and first-class functions.

Figure 1 shows an example of a skeleton for evaluating a *while* block of a Vanilla While language. We see all the main elements of

[1] https://github.com/tilk/LambdaCert

$$
\begin{array}{l}
\texttt{eval\_stmt}\ (\sigma, \texttt{While}\ (e,t)) \coloneqq \\
\quad \text{let}\ b = \texttt{eval\_expr}\ (\sigma, e)\ \text{in} \\
\quad \left(\begin{array}{l}
\text{let}\ \texttt{True} = b\ \text{in let}\ \sigma' = \texttt{eval\_stmt}\ (\sigma, t)\ \text{in} \\
\qquad\qquad \texttt{eval\_stmt}\ (\sigma', \texttt{While}\ (e,t)) \\
\text{let}\ \texttt{False} = b\ \text{in}\ \sigma
\end{array}\right)
\end{array}
$$

Figure 1: Skeleton for a `While` Constructor

Skeletal Semantics, and we can observe that they are actually the main elements of any semantics.

- Top-level terms (`eval_stmt` and `eval_expr`) are recursive. They let us define the semantics of expressions depending on the semantics of other expressions (often sub-expressions, but not always, as we can see in this example with the call $\texttt{eval\_stmt}\ (\sigma', \texttt{While}\ (e,t))$).
- There are auxiliary functions and auxiliary types such as booleans and the possibility of matching a value against a given constructor (e.g., `True`).
- The `let-in` construct is used to perform a sequence of operations.
- The branching (represented as a parenthesized system in Figure 1) is a choice between two possible rules. Often, the branches are mutually exclusive, and a pattern matching at the start of the branch determines which branch is taken. Overlapping branches also let us represent non-deterministic semantics (such as λ-calculus with no evaluation strategy).

3.1 Formalism

The syntax of the Skeletal semantics is as follows.

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\ t \mid (t, \ldots, t) \mid \lambda p : \tau \cdot S \mid t.f \\
 & & & \mid (f = t, \ldots, f = t) \mid t \leftarrow (f = t, \ldots, f = t) \\
\textsc{Skeleton} & S & ::= & t_0\ t_1 \ldots t_n \mid \text{let}\ p = S\ \text{in}\ S \\
 & & & \mid \text{branch}\ S\ \text{or} \ldots \text{or}\ S\ \text{end} \mid t \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p, \ldots, p) \mid (f = p, \ldots, f = p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \ldots, \tau) \\
\textsc{Term decl} & r_t & ::= & \textbf{val}\ x : \tau \mid \textbf{val}\ x : \tau = t \\
\textsc{Type decl} & r_\tau & ::= & \textbf{type}\ b \mid \textbf{type}\ b = \text{"|"}\ C\ \tau \ldots \text{"|"}\ C\ \tau \\
 & & & \mid \textbf{type}\ b := \tau \mid \textbf{type}\ b = (f : \tau, \ldots, f : \tau)
\end{array}
$$

A term is either a variable, a constructor applied to a term, a (possibly empty) tuple of terms, a λ-abstraction (where the type of the abstracted variable is explicitly given), the access to a given field of a term, a record of terms, or a term with the reassignment of some fields. Terms can be seen as expressions in their normal form (with no possible reduction). A skeleton is either the application of a variable to several terms, a let binding, where the bound skeleton is matched against a pattern, a branching of several skeletons, or simply a term. A pattern is either a variable name, a wildcard, a constructor applied to a pattern, a tuple, or records of patterns. Finally, a type is either a base type, that is a type declared by the user (either specified or unspecified), an arrow type, or a tuple of

types. We do not formally describe polymorphic types here; they will be introduced below.

This syntax is very close to Administrative Normal Forms [7, 19]. The main difference is that we add branching as a non-deterministic choice. We also provide constructors, tuples, and records, and we allow let bindings in the bound part of let bindings. These embedded let bindings can always be extracted by using a fresh variable name, which can be done automatically using the function `necrotrans extractletin` [14].

A Skeletal Semantics is a list of *terms declarations* (`T`) and *type declarations* (`D`), either *unspecified* or *specified*. Specified type declarations include algebraic data types, type aliases, and record types. Here are some examples of such declarations, written in the actual Skel syntax.

```
(* Unspecified type and term declarations *)
type int
val add: int → int → int

(* Specified type and term declarations *)
type nat = | Zero | Succ nat
val two:nat = Succ (Succ Zero)
```

The possibility to declare unspecified types and terms is a really powerful tool. When defining a semantics, we sometimes do not want to go into details on how every type and every function works, or we cannot specify it as it is implementation dependent. Partial specifications allow us to do that.

As an example, we show in Skel syntax our to write the code corresponding to Figure 1.

```
val eval_stmt (s:state) (t:stmt) =
    branch
        let While(e,tw) = t in
        let b = eval_expr s e in
        branch
            let True = b in
            let s' = eval_stmt s tw in
            eval_stmt s' (While(e,tw))
        or
            let False = b in
            s
        end
    or
        ...
    end
```

The outer branching corresponds to the matching of the input statement `t`: every branch starts with a let-pattern-matching that both ensures the statement has the right shape and extracts its sub-components. If the statement does not have this shape, another branch (not shown, part of the `...`) is taken. Then the expression of the while is evaluated, and the result is checked in another branching, to distinguish `True` and `False` cases. The execution then continues accordingly.

To improve readability, the Skel language also provides notation for binding operators, whose type has shape `a → (b → c) → d`. That is, assuming a term declaration for a function `bind`, and a symbol definition `binder @ = bind`, one can write `let p =@ x in v` as syntactic sugar for `bind x (λp → v)`. It is handy when writing semantics that heavily uses monadic constructions, such as the one for JS.

In itself, the language is only a syntactic construct, but there are multiple ways to derive meaning out of a Skeletal Semantics. This enables the writing of a single Skeletal Semantics to obtain multiple semantics. The usual way to interpret a Skeletal Semantics is the concrete semantics, which corresponds to a natural (big step) semantics. We describe it in Section 3.3, but first, we focus on typing.

3.2 Typing

Skel is strongly typed. As mentioned above, types are threefold: user-defined types (specified or unspecified), product types for tuples and records, and arrow types for functions.

We give the typing rules for terms and skeletons in Figure 2. They are respectively of the form $\Gamma \vdash t : \tau$ and $\Gamma \vdash S : \tau$, where Γ is a typing environment (a partial function that binds variable names to types).

To type a variable, we look for its type in Γ. To type a constructor applied to a term, we type the given term, check that it matches the requirement for the constructor, and return the output type of the constructor. Both are given using $\mathrm{ctype}(C)$, which returns the pair of the declared input type and output type for the constructor C as found in the type declaration list. To type a tuple, we type each component and return the tuple of the types. To type an abstraction, we type the skeleton and return the arrow from the argument's type to the skeleton's.

Similarly, we define $\mathrm{ftype}(f)$, a function that, given a field, returns (τ, ν), where τ is the type of the field f, and ν is the record's type in which f is defined. We also define the function $\mathrm{fields}(\tau)$ that, given a record type, returns the lists of record fields associated with the type declaration for τ. Once this is defined, to type a field-access, we check that the term is of the correct type (the record type for the field) and return the type contained by the field. To type a record, we check that all fields are given and of the correct type. Finally, to type a field-rewrite, we check all types and return the corresponding record type.

To type a branching, we require that every branch has the same type τ, and we return τ (an empty branch, therefore, can have any type). To type a let binding, we first type the bound skeleton and then type the continuation in the extended environment. To this end, we introduce the extension of Γ with a matched type.

$$\Gamma + x \leftarrow \tau \triangleq \Gamma' \text{ where } \Gamma'(x) = \tau \text{ and } \Gamma'(y) = \Gamma(y) \text{ for } y \neq x$$
$$\Gamma + \_ \leftarrow \tau \triangleq \Gamma$$
$$\Gamma + C\ p \leftarrow \tau' \triangleq \Gamma + p \leftarrow \tau \ \mathit{if}\ \mathrm{ctype}(C) \triangleq (\tau, \tau')$$
$$\Gamma + (p_1, \ldots, p_n) \leftarrow (\tau_1, \ldots, \tau_n) \triangleq (\Gamma + p_1 \leftarrow \tau_1) \cdots + p_n \leftarrow \tau_n$$
$$\Gamma + (f_1 = p_1, \ldots, f_n = p_n) \leftarrow \tau \triangleq (\Gamma + p_1 \leftarrow \tau_1) \cdots + p_n \leftarrow \tau_n$$
$$\text{where } \mathrm{fields}(\tau) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$$

Finally, to type the n-ary function application, we require the term in function position to have an arrow type and the types of the applied terms to match the input types of the function.

We now briefly describe how we provide polymorphic types in practice since they are heavily used in the JS formalization below. Polymorphism is always explicitly declared. Thus, when declaring a polymorphic type or term, one has to give its type arguments. For

$$\textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Term}\ \frac{\texttt{val x:}\tau\ \texttt{(= t)} \in \mathsf{T}}{\Gamma \vdash x : \tau} \qquad \textsc{Const}\ \frac{\Gamma \vdash t : \tau \qquad \text{ctype}(\mathrm{C}) = (\tau, \tau')}{\Gamma \vdash \mathrm{C}\, t : \tau'}$$

$$\textsc{Tuple}\ \frac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n)} \qquad \textsc{Clos}\ \frac{\Gamma + p \leftarrow \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \cdot S) : \tau \to \tau'}$$

$$\textsc{FieldGet}\ \frac{\Gamma \vdash t : \nu \qquad \text{ftype}(f_i) = (\tau_i, \nu)}{\Gamma \vdash t.f_i : \tau_i} \qquad \textsc{FieldSet}\ \frac{\Gamma \vdash t : \nu \qquad \forall i, \Gamma \vdash t_i : \tau_i \qquad \forall i, \text{ftype}(f_i) = (\tau_i, \nu)}{\Gamma \vdash t \leftarrow (f_1 = t_1, \ldots, f_m = t_m) : \nu}$$

$$\textsc{Rec}\ \frac{\text{fields}(\tau) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\} \qquad \forall i, \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (f_1 = t_1, \ldots, f_n = t_n) : \tau} \qquad \textsc{Branch}\ \frac{\Gamma \vdash S_1 : \tau \quad \ldots \quad \Gamma \vdash S_n : \tau}{\Gamma \vdash (S_1 \ldots S_n) : \tau}$$

$$\textsc{LetIn}\ \frac{\Gamma \vdash S : \tau \qquad \Gamma + p \leftarrow \tau \vdash S' : \tau'}{\Gamma \vdash \text{let}\, p = S \,\text{in}\, S' : \tau'}$$

$$\textsc{App}\ \frac{\Gamma \vdash t_0 : \tau_1 \to \cdots \to \tau_n \to \tau \qquad \forall i\ \Gamma \vdash t_i : \tau_i}{\Gamma \vdash t_0\, t_1 \ldots t_n : \tau}$$

Figure 2: Typing rules of Skeletal Semantics

example, below, we show the declaration of the `list` type and a `map` term.

```
type list<a> = | Nil | Cons (a, list<a>)
val map<a,b> (f: a → b) (l: list<a>) : list<b> =
    branch let Nil = l in Nil<b>
    or     let Cons (v, q) = l in
           let w = f v in
           let q' = map<a,b> f q in
           Cons<b> (w, q')
    end
```

As we can see, the type arguments must be explicitly given when applying a function or a constructor. On the other hand, they are not given in a pattern because they can always be safely inferred.

3.3 Concrete Interpretation

The concrete interpretation gives a natural, or big-step, semantics to a Skeletal Semantics. To define the concrete interpretation, we need to be given an instantiation for each unspecified type and term. If x is an unspecified term of type $\tau_1 \to \ldots \tau_n \to \tau$, where τ is not an arrow type (written non-arrow(τ)), then the instantiation $[\![x]\!]$ must be a function from $V_{\tau_1} \times \ldots \times V_{\tau_n}$ to $\mathcal{P}(V_\tau)$, where each type τ is associated to a set of values V_τ as follows.

If τ is a non-specified type, V_τ must be given. If τ is specified by a list of constructors, V_τ is the set freely generated by the constructors of type τ, where the argument of each constructor is recursively generated. We have $V_{(\tau_1,\ldots,\tau_n)} = V_{\tau_1} \times \cdots \times V_{\tau_n}$, and $V_\tau = \{(f_1 = v_1, \ldots, f_n = v_n) \mid v_1 \in V_{\tau_1} \wedge \cdots \wedge v_n \in V_{\tau_n}\}$ if $\text{fields}(\tau) = \{f_1 : \tau_1, \ldots, f_n : \tau_n\}$. If τ is an arrow type $\tau_i \to \tau_j$, then V_τ includes closures of the form (p, E, S) where there is a Γ such that, $\forall x, \Gamma \vdash E(x) : \Gamma(x)$ and $\Gamma + p \leftarrow \tau_i \vdash S : \tau_j$. The set $V_{\tau_i \to \tau_j}$ also includes partially applied unspecified terms $\text{unspec}(\mathsf{x}, (v_1, \ldots, v_n))$, where x has type $\tau_1 \to \ldots \to \tau_n \to \tau_i \to \tau_j$ and for each $k \in [1..n]$, we have $v_k \in V_{\tau_k}$.

Skel is a strongly typed language. Every term and every skeleton has a unique type, except for the empty branchings that can have any type. To recover type uniqueness, we require that empty branchings be explicitly annotated with their type. Every term of type τ can be evaluated to zero, one, or several values of V_τ. Evaluation of a term may be non-deterministic. Formally, the evaluation is a relation between terms/skeletons and values. We define in Figure 3 the evaluation of a term and of a skeleton in a given environment E, which is a partial function that binds variables to values. To evaluate a variable, we look for its value in E. A specified term is evaluated to the corresponding term in the term declarations. An unspecified term x is evaluated to a partial application $\text{unspec}(\mathsf{x}, ())$ if it has an arrow type or directly to a value of its instantiation $[\![\mathsf{x}]\!]$ otherwise. To evaluate a constructor applied to a term, we evaluate the term to a value and return the constructor applied to this value. To evaluate a tuple, we evaluate each component and return the tuple of the values. This is similar when evaluating records, record field accesses, and record field updates. To evaluate an abstraction, we make a closure by remembering the evaluation context. To evaluate a branching, we return the value v of a branch that successfully evaluates to v. To evaluate an application, we evaluate all terms and perform the application as explained below. To evaluate a let binding, we evaluate the first skeleton as a value, extend the environment doing pattern matching, and evaluate the second skeleton in this extended environment. Pattern matching is recursively defined as follows.

$$E + x \leftarrow v \triangleq E' \text{ where } E'(x) = v \text{ and } E'(y) = E(y) \text{ for } y \neq x$$
$$E + \_ \leftarrow v \triangleq E$$
$$E + C\ p \leftarrow C\ v \triangleq E + p \leftarrow v$$
$$E + (p_1, \ldots, p_n) \leftarrow (v_1, \ldots, v_n) \triangleq (E + p_1 \leftarrow v_1) \cdots + p_n \leftarrow v_n$$
$$E + (f_1 = p_1, \ldots, f_n = p_n) \leftarrow (f_1 = v_1, \ldots, f_n = v_n) \triangleq (E + p_1 \leftarrow v_1) \cdots + p_n \leftarrow v_n$$

To perform an application, if there are no arguments, we just return the value itself. If there is at least one argument, we check the first value. As it is of functional type, it is either a closure or an unspecified term with missing arguments. For a closure (p, S, E), we extend the stored environment E by matching the first argument with the pattern p and we evaluate the skeleton S in this extended

$$\textsc{Var}\ \frac{E(x) = v}{E, x \Downarrow v} \qquad \textsc{TermSpec}\ \frac{\texttt{val x}:\tau = \texttt{t} \in \mathsf{T} \qquad \emptyset, t \Downarrow v}{E, x \Downarrow v}$$

$$\textsc{TermUnspecArrow}\ \frac{\texttt{val x}:\tau_1 \to \tau_2 \in \mathsf{T}}{E, x \Downarrow \text{unspec}(\texttt{x}, ())} \qquad \textsc{TermUnspecNonArrow}\ \frac{\texttt{val x}:\tau \in \mathsf{T} \qquad \text{non-arrow}(\tau) \qquad v \in [\![x]\!]}{E, x \Downarrow v}$$

$$\textsc{Const}\ \frac{E, t \Downarrow v}{E, (\mathrm{C}\ t) \Downarrow \mathrm{C}\ v} \qquad \textsc{Tuple}\ \frac{\forall i, E, t_i \Downarrow v_i}{E, (t_1, \ldots, t_n) \Downarrow (v_1, \ldots, v_n)} \qquad \textsc{Clos}\ \frac{}{E, (\lambda p : \tau \cdot S) \Downarrow (p, E, S)} \qquad \textsc{FieldGet}\ \frac{E, t \Downarrow (f_1 = v_1, \ldots, f_n = v_n)}{E, t.f_i \Downarrow v_i}$$

$$\textsc{FieldSet}\ \frac{E, t \Downarrow (f_1 = v_1, \ldots, f_n = v_n) \qquad \forall i, E, t_{j_i} \Downarrow w_{j_i} \qquad i \notin \{j_1, \ldots, j_m\} \to w_i = v_i}{E, t \leftarrow (f_{j_1} = t_1, \ldots, f_{j_m} = t_m) \Downarrow (f_1 = w_1, \ldots, f_n = w_n)}$$

$$\textsc{Rec}\ \frac{\forall i, E, t_i \Downarrow v_i}{E, (f_1 = t_1, \ldots, f_n = t_n) \Downarrow (f_1 = v_1, \ldots, f_n = v_n)}$$

$$\textsc{Branch}\ \frac{E, S_i \Downarrow v}{E, (S_1 \ldots S_n) \Downarrow v} \qquad \textsc{LetIn}\ \frac{E, S \Downarrow v \qquad E + p \leftarrow v, S' \Downarrow w}{E, \text{let } p = S \text{ in } S' \Downarrow w}$$

$$\textsc{App}\ \frac{\forall i\ E, t_i \Downarrow v_i \qquad (v_0, v_1, \ldots, v_n) \Downarrow_{\mathsf{app}} w}{E, (t_0\ t_1 \ldots t_n) \Downarrow w} \qquad \textsc{AppNil}\ \frac{}{(v) \Downarrow_{\mathsf{app}} v}$$

$$\textsc{AppClos}\ \frac{E + p \mapsto v_1, S \Downarrow w \qquad (w, v_2, \ldots, v_n) \Downarrow_{\mathsf{app}} x}{((E, p, S), v_1, \ldots, v_n) \Downarrow_{\mathsf{app}} x}$$

$$\textsc{AppUnspecCont}\ \frac{\texttt{val x}:\tau \in \mathsf{T} \qquad \tau = \tau_1 \to \cdots \to \tau_{n+2}}{(\text{unspec}(\texttt{x}, (v_1, \ldots, v_i)), v_{i+1}, \ldots, v_n) \Downarrow_{\mathsf{app}} (\text{unspec}(\texttt{x}, (v_1, \ldots, v_n))}$$

$$\textsc{AppUnspecOver}\ \frac{\texttt{val x}:\tau \in \mathsf{T} \qquad \tau = \tau_1 \to \cdots \to \tau_{n+1} \qquad \text{non-arrow}(\tau_{n+1}) \qquad w \in [\![x]\!](v_1, \ldots, v_n)}{(\text{unspec}(\texttt{x}, (v_1, \ldots, v_i)), v_{i+1}, \ldots, v_n) \Downarrow_{\mathsf{app}} w}$$

Figure 3: Concrete Interpretation of Skeletal Semantics

environment. We then apply the result to the remaining arguments. For an unspecified term, if there are still arguments missing, then we collect all arguments and wait for the others. If all arguments are provided, we apply the instantiation of the unspecified term and take a value from the result.

We can prove that if $\Gamma \vdash t : \tau$ and $E, t \Downarrow v$, then it entails that $v \in V_\tau$, given that Γ matches E, that is $\forall x, \Gamma(x) = \tau \implies E(x) \in V_\tau$. This result is proven in Necro Coq[2]

4 ECMASCRIPT ALGORITHMS IN SKEL

This section illustrates how we formalized ECMAScript algorithms in Skel. In Section 4.1, we provide some context about the ES specification. Then, in Sections 4.2 to 4.4, we introduce methods to extend the tiny Skel language for handling and combining side-effects, ending up in a structured and systematic approach for describing the algorithms. Additional details about this approach may be found in [10]. In Section 4.5, we present the description of the `GetValue`[3] *Internal Method* as an example of our formalization approach. The `GetValue` method is one of the most used in ES, and it is also complex enough to describe our systematic approach to formalizing ES algorithms. In Section 4.6, we detail the current state of the work, and we evaluate it in Section 4.7.

4.1 ECMAScript

ECMAScript is a large vernacular specification written in an imperative style. As of 2022, It is divided into 29 chapters and nine appendices. Despite its complexity and verbosity, it provides a complete specification of the behavior of JS. Some choices are left to the implementation, however, so a formalization must consider these unspecified pieces. We explain below how we deal with them.

After an introductive part in chapters 1 to 5, where the notations and algorithmic conventions are defined, the document can be divided into three main functional groups. Chapters 6, 7, 9, and 10 give a taxonomy of ES *Data Types and Values*, providing each taxon with a definition of its operations and related invariant, followed by the definition of *abstract operations* (type conversion, comparison, object's and iterator's operations), the *runtime environment*, *executions contexts*, and detailed classification of the type *Object* and its internal methods. This block of chapters gives a complete overview of the execution environment in which an ES program should be executed. Formalizing any language construct first requires a formalization of this execution environment. Chapter 8 defines some syntax-directed operations. The central part of the specification, chapter 11 to chapter 16, describes the actual ES programming language. Chapters 11 and 12 focus on lexical units. Language constructs are given in chapters 13 to 15, where each construct is given with its syntactic specification and its evaluation. These describe expressions, statements, and functions. Scripts and modules are defined in chapter 16. Chapter 17 introduces the notion of *early error*, which intuitively corresponds to a parsing error, and of language extensions. Chapters 18 to 28 introduce the default components of the *Global Object* as well as many objects that could be considered as a standard library. Finally, a memory model is given in Chapter 29.

[2]https://gitlab.inria.fr/skeletons/necro-coq/-/blob/master/proofs/typecheck/SubjectReduction.v#L1434
[3]https://tc39.es/ecma262/2021/#sec-getvalue

```
type state (* specified in the instantiation *)
type st<a> := state → (a, state)

val st_bind<a,b> (w: st<a>) (f: a → st<b>) : st<b> =
    λs: state → let (v, s') = w s in f v s'
val st_ret<a>: (v: a) : st<a> = λs: state → (v, s)
```

Figure 4: State Monad in Skel

4.2 Challenges of the Formalization

The first step of the mechanization in the purely functional Skel language is to deal with the imperative nature of the specification, raising two issues.

First, there is a notion of an implementation-dependent state that can be mutated. More precisely, we define by *state* the aggregation of all the imperative data manipulated by the specification. We design it as a record that includes the Execution Context stack, a strictness boolean flag, and a pool of Maps holding *Execution Contexts*, *Environment Records*, *Realms*, *Script Records*, and *Objects*. This record is formalized as a member of an unspecified type, as well as the functions to access and set it. This is representative of our general approach: having all the "implementation dependent" parts of the ES specification kept unspecified. We have fully defined the state and its helper functions in the instantiation in OCaml (see Section 4.7.1). To remain close to the imperative specification, we use a state monad, shown in Figure 4. This monad lets us implicitly pass the state around.

Second, the specification often breaks the usual control flow by having a `return` in the middle of algorithms. We can capture such control flows by using nested branches (see below), but this significantly reduces the legibility of the mechanization. We thus use a *control flow* monad to simplify the mechanization. In addition, the specification itself introduces operators that behave much like an exception monad to deal with `break`, function returns, or exceptions. We thus propose in Section 4.3 an exception monad that captures the behavior of ES monadic short-hands `?` and `!`,[4] and in Section 4.4, its extension to handle control-flow features. We claim that combining the state monad with the control-flow and the exception one dramatically simplifies our code, making the Skel description of ES algorithms easy to write and maintain. The result is close to the specification, as shown in Section 4.5.

4.3 Completion Record and the ECMAScript Error Handling (?!) monad

Most ES operations do not directly return values, they return *completion records*[5] instead. A Completion Record describes the runtime propagation of values and control flow. This record is composed of three fields: a kind (either a result, a break out of a loop, a return of a function, or an exception being thrown), an optional value, and in the case of a break or continue, a target.

In theory, a completion record should only hold ES language values (Null, Undefined, Boolean, Number, BigInt, String, Symbol, and Object) or be empty. In practice, it is used to return many

```
type completionType= |Normal |Break |Continue
                     |Return |Throw

type completionValue<a> =
| Ok a
| Abruption maybeEmpty<value>

type completionTarget := maybeEmpty<string>

type completionRecord<a> = ( _Type_: completionType,
                             _Value_: completionValue<a>,
                             _Target_: completionTarget )
```

Figure 5: Skel Formalization of ES Completion Records

```
type out<a> =
| Success completionRecord<a>
| Anomaly anomaly
type anomaly =
| AbruptAnomaly completionRecord<()>
| StringAnomaly string
| NotImplemented
```

Figure 6: Out and Anomaly Declarations

other constructions.[6] If we consider the `getValue(V)` abstract operation, the completion record given as input can hold either a Value or a Reference in its `_Value_` field. Many such examples litter the spec. Hence, we consider completion records to be polymorphic in what their "value" field holds. We declare such completion records as `type completionRecord<a>`. Figure 5 defines the types corresponding to the contents of this record: the `completionType` holds the kind of the record, and the `completionValue` is either an `Ok` polymorphic constructor that contains the value of a non-abrupt computation or an abruption. An ES abrupt Completion Record is one whose type is not **normal**. For instance, a **throw** record has an exception object as completion value, a **return** record has an optional value, and a **break** record has `empty`. Due to the aforementioned specification issues where non-values may be returned, we store the optional value in a separate constructor to be able to return it independently of the completion type. Finally, the `completionTarget` holds the optional string representing the target, typically used for a `break` to a label. An ES completion record for values has type `completionRecord<maybeEmpty<value>>`.

We next define a type `out` (Figure 6) composed of two constructors, `Success` for *successful computations* and `Anomaly` for anomalies, which intuitively corresponds to a failure of the specification, e.g., when an assertion does not hold. A successful computation is one that returns an ES completion record, either `Normal` or `Abrupt`. The `Anomaly` constructor captures incorrect computations. In the specification, anomalies can be raised by assertion failures or when it is explicitly written that an abstract operation call must not return an `Abruption`. The specification authors informally guarantee that these failures never occur.

The `Anomaly` constructor is of type `anomaly`. It has three type constructors: `AbruptAnomaly` holding a completion record in case

[4] https://tc39.es/ecma262/2021/#sec-returnifabrupt-shorthands
[5] https://tc39.es/ecma262/2021/#sec-completion-record-specification-type

[6] In early 2022, the specification changed to accept any value and not just language values, see https://github.com/tc39/ecma262/pull/2547

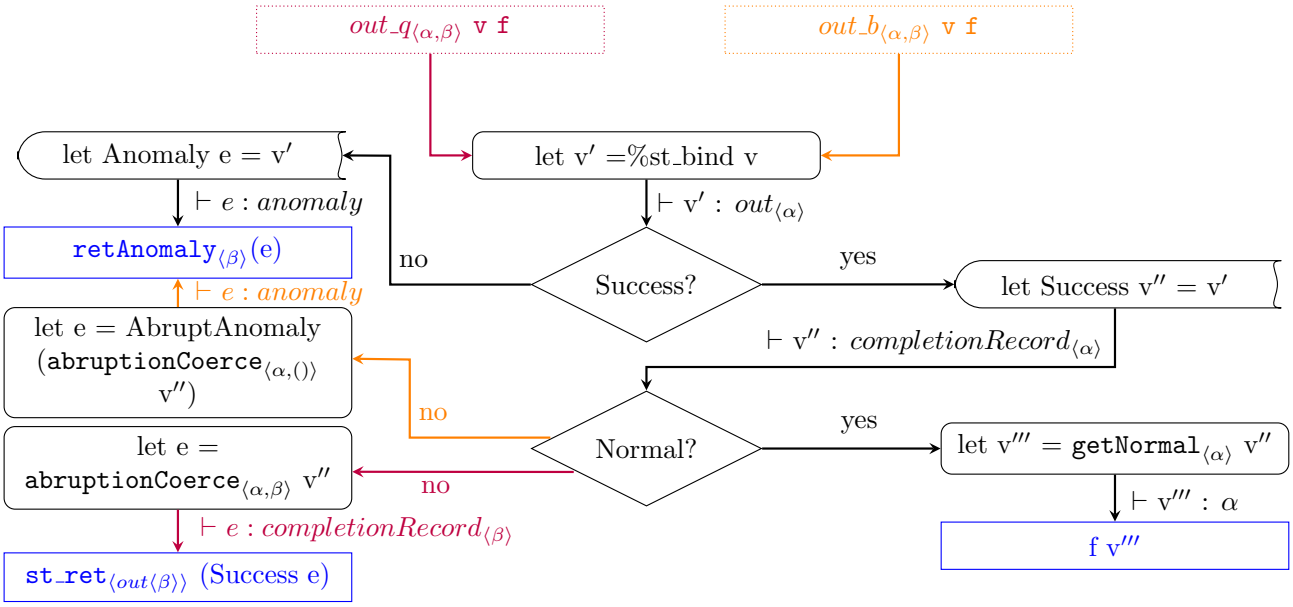

Figure 7: The Model of ? and !

an evaluation returns an unexpected abruption, `StringAnomaly` that contains textual information about an anomaly—when an assertion of the specification is broken or when an implementation-dependent operation fails, and the `NotImplemented` signaling that we have not yet implemented some feature.

ES abruptions are propagated through an abstract method called `ReturnIfAbrupt`. Basically, this method gets a completion record and either returns the value of the `_Value_` field in case of a normal completion, or it propagates the abruption.

In cases of an abstract operation or a recursive evaluation, the specification uses the prefix `?` to indicate that `ReturnIfAbrupt` has to be applied to the resulting completion. This operator is a monadic bind for a variant of the classic exception monad. The other operator used in the specification is `!`: it behaves similarly to `?` on a normal result, but it asserts an abruption cannot occur. We thus model it as transforming an abruption into an anomaly. These behaviors are reflected in the flow-chart presented in Figure 7, presenting respectively `out_q`$_{\langle\alpha,\beta\rangle}$(red arrows) for the `?`, and `out_b`$_{\langle\alpha,\beta\rangle}$(orange arrows) for the `!`. We define them as the monadic binders of the combination of `st` and `out`. We put the state monad outside as we want to return a state in the case of an exception.

```
val out_q<a,b>: st<out<a>> → (a→st<out<b>>) → st<out<b>>
val out_b<a,b>: st<out<a>> → (a→st<out<b>>) → st<out<b>>
val out_ReturnIfAbrupt<a,b>:
      out<a> → (a → st<out<b>>) → st<out<b>>
binder ?out = out_q
binder !out = out_b
```

We also define getters (`getNormal`, `getAbrupt`, …) to extract values from a `completionRecord`, and return functions (`retAnomaly`, `retAbrupt`, `retNormal`, …) for successful and anomaly computations. Note the use of the `abruptionCoerce` operation that is only defined for completion records that are abruptions. It is the identity, but it changes the type parameter of the completion record. What follows shows how faithful is the translation of these types of algorithmic steps in Skel. The annotation τ_{bar} is the type of `bar`.

```
type cf<a> = | ReturnControl a | ContinueControl


val cf_ret<a> (v: a) → cf<a> = ReturnControl<a> v
val cf_cont<a> : cf<a> = ContinueControl<a>

val cf_bind<a> : cf<a> → (() → cf<a>) → cf<a>
binder @cf := cf_bind
val cf_res<a> : cf<a> → (a → a) → a
binder <cf := cf_res
```

Figure 8: The Control-Flow Monad

(1) let bar be ? AbstractOperation(foo)
(2) Return bar

```
(*1*) let bar =?out abstractOperation(foo) in
(*2*) retNormal<τbar> bar
```

4.4 A Control-Flow monad

As said earlier, the ES specification uses imperative control flow, such as returning in the middle of an algorithm. In Figure 8, we introduce the `type cf<a>`. It represents computations that can either continue or that terminate with a result of type `a`. It comprises two constructors: `ReturnControl`, to return a result of type `a`, and `ContinueControl` to continue the execution.

We define a monadic binder `cf_bind`, a function `cf_res` to extract the value from a `cf<a>` term, a function `cf_ret` to break the control-flow and return a value, and a function `cf_cont` to signal the execution should continue.

We illustrate in Figure 9 the translation of algorithmic steps in Skel, with and without the control-flow monad. We recall that we use the forms `let True = ... in ...` to test whether a value is `true`, and `let False = ... in ...` to test whether it is `false`. We also write `foo;@ bar` for `let _ =@ foo in bar`. Despite the slight overhead in notation, the control-flow approach is closer to the algorithmic steps and can be consistently applied to deal with the typical case of a conditional that returns without an else branch. One can achieve a similar behavior without the control flow monad at the cost of nested branching. It is possible to avoid the nesting of branching by hoisting all the branches at top-level. This results in the following code.

```
branch let True = foo in 1
or     let False = foo in let True = bar in 2
or     let False = foo in let False = bar in
       let True = baz in 3
or     let False = foo in let False = bar in
       let False = baz in 4
end
```

The reason previous conditions are repeated is because there is no guarantee that the evaluation of a branching will consider branches in declaration order. In addition, using a collecting semantics (where all branches are considered) would give the wrong result if we did not restate all conditions.

We define `st<cf<out<a>>>` type as the combination of the three monadic types, and accordingly, the definitions of the binders and return functions as `bind` (@), `cf_out` (<), `cont`, and `ret`. Our general approach is to encapsulate all the algorithmic steps in this type, returning an `st<out<a>>` at the end of every function that may raise an exception. To this end, we start each algorithm with `let result =<` to enter the full monad with control, and we exit the control monad at the end of the algorithm with `in result`. Figure 10 gives a variant of Figure 9 with exceptions. Notice that the only thing that changes is the first step and the use of the appropriate monadic binders for the type `st<cf<out<a>>>`.

Despite the closeness to the algorithmic steps, the latter figure, in lines 3, 5, and 7, shows redundancy of code in the `or` branches. Indeed, each time the condition is not satisfied, we have to explicitly state that the evaluation continues. To avoid doing so, we define two boolean binder-like functions, `ifTrue` and `ifFalse`, for introducing partiality in branchings. The `ifTrue` binder, denoted as `@t`, evaluates the rest of the branch when given the `True` value and directly returns a `ContinueControl` when given the `False` value. The following code applies the `ifTrue` binder to the example in Figure 10.

```
let result =<
   branch foo;@t throw<int> fooError end;@
   branch bar;@t ret<int> 2 end;@
   branch baz;@t ret<int> 3 end;@
   ret<int> 4
in result
```

(1) If foo is true Return 1
(2) If bar is true Return 2
(3) If baz is true Return 3
(4) Return 4

```
(*no control-flow monad*)
branch let True = foo in 1
or     let False = foo in
       branch let True = bar in 2
       or     let False = bar in
              branch let True = baz in 3
              or     let False = baz in 4
              end
       end
end
```

```
(*control-flow monad*)
let result =<cf
   branch let True = foo in cf_ret<int> 1
   or     let False = foo in cf_cont<int>
   end;@cf
   branch let True = bar in cf_ret<int> 2
   or     let False = bar in cf_cont<int>
   end;@cf
   branch let True = baz in cf_ret<int> 3
   or     let False = baz in cf_cont<int>
   end;@cf
   cf_ret<int> 4
in result
```

Figure 9: Code with and without Control Flow Monad.

(1) If foo is true throw FooError
(2) If bar is true Return 2
(3) If baz is true Return 3
(4) Return 4

```
1 let result =<
2    branch let True = foo in throw<int> fooError
3    or     let False = foo in cont<int> () end;@
4    branch let True = bar in ret<int> 2
5    or     let False = bar in cont<int> () end;@
6    branch let True = baz in ret<int> 3
7    or     let False = baz in cont<int> () end;@
8    ret<int> 4
9 in result
```

Figure 10: Variant of Figure 9 with Exceptions

4.5 A Real Example in Skel

In this section, we show the effectiveness of the design choices introduced in the previous sections. We proceed by presenting the formalization of the `GetValue` abstract method as an example. This method, presented at the top of Figure 11, is one of the most used methods in the specification, as it is often called after the evaluation of syntactic constructors of the language. It takes V as input, a completion record containing either a value or a reference, and returns a value as output. In a nutshell, if V is a value, `GetValue` returns it, and if it is a reference, `GetValue` acts like a binding resolver to obtain a primitive value from an Object or an Environment Record.

(1) ReturnIfAbrupt(V).
(2) If Type(V) is not Reference, return V.
(3) If IsUnresolvableReference(V) is **true**, throw a **ReferenceError** exception.
(4) If IsPropertyReference(V) is **true**, then
  (a) Let base be V.[[Base]].
  (b) Let baseObj be ! ToObject(base).
  (c) Return ? baseObj.[[Get]](V.[[ReferencedName]], GetThisValue(V)).
(5) Else,
  (a) Let base be V.[[Base]].
  (b) Assert: base is an Environment Record.
  (c) Return ? base.GetBindingValue(V.[[ReferencedName]], V.[[Strict]]).

```
1  (*1*) let v =%returnIfAbrupt v in
2  (*2*) branch valref_Type(v,T_Ref);@f let Val v = v in ret<value> v end;@
3  (*N*) let Ref v = v in
4  (*3*) branch isUnresolvableReference(v);@t throw<value> referenceError end;@
5  (*4*) branch let True = isPropertyReference(v) in
6        (*a*) let R_Val base = v.base in
7        (*b*) let baseObj =! toObject(base) in
8        (*N*) let baseObj =/o baseObj in
9        (*c*) let thisVal =? getThisValue(v) in
10       (*c*) let r =? baseObj._Get_(v.__ReferencedName__, thisVal) in
11       (*c*) ret<value> r
12 (*5*) or    let False = isPropertyReference(v) in
13       (*a*) let base = v.base in
14       (*b*) assert_true<(reference,type_ref)> ref_Type (base, T_R_ER);@
15       (*N*) let R_ER base = base in let base =/er base in
16       (*c*) let r =? base._GetBindingValue_(v.__ReferencedName__, v.__Strict__) in
17       (*c*) ret<value> r
18 end
```

Figure 11: The ECMAScript's `GetValue(V)` and its Skel Formalization

The reference[7] type is a record that contains four fields, including the `[[Base]]` field that holds an environment record or an ES value and three other fields not relevant here. Our mechanization in Skel specifies references as records too.

We now describe step-by-step how we formalize this method (the code is collected as a single function at the bottom of Figure 11). The type mixing values and references is the specified type `valref` defined as `| Val value | Ref reference`. The argument of `GetValue` thus has type `out<valref>`. We highlight the code that does not correspond to the ES algorithmic steps, i.e., code that is an overhead of the Skel formalization. It is necessary for typing reasons (extracting a value from a variant type) or transforming a heap location into its contents.

The first step of the abstract method applies `ReturnIfAbrupt` on V. In case it is an abruption, the method propagates the completion record to the caller. Otherwise, it extracts the `completionValue`. To model this, we use the `%returnIfAbrupt` monadic binder. We could directly write `let v =@ returnIfAbrupt<(valref,value)>(v)`, but this requires specifying the polymorphic type components, which is not necessary for binders as they are inferred. We are working on extending the type inference to make polymorphic annotations unnecessary and thus be closer to the specification. If V is a normal completion, the newly bound V will have type `valref`.

The next ES step inspects the type of V. If it is not a reference, hence a value, we return it. Otherwise, the `@f` implicitly continues the execution. Then, in Skel 's line 3, we can safely extract v from the `Ref` type constructor. Now `v` has type `reference`.

Step 3 follows the same pattern as step 2: An *unresolvable* reference is one that has `Undefined` or `Null` as `[[Base]]`. In this case, a `ReferenceError` is thrown. In Skel, we proceed straightforwardly. If the reference is unresolvable, we throw a `referenceError` of type value. The `throw<a>` function takes an error object constructor as an argument. Then, it returns an abrupt `completionRecord` that contains as a `completionValue` a reference error object.

In case the reference is resolvable, step 4 inspects whether V is a *property reference*. A property reference is a reference that has a non-null, defined base value of type `value`. Step 4.a.assigns the base value stored into the reference to the variable V. In line 6 of the Skel formalization, we access the record field. In the same line, we make an inline pattern matching, expecting the base value to be of type `value` (`R_Val`). Now base is a primitive value, meaning one of type `Boolean`, `String`, `Symbol`, `BigInt`, or `Number`. Then, line 7 describes Step 4.b. The primitive value in base is cast to an object. The method `ToObject`, transforms a primitive value into an object, raising an exception when the value is `Undefined` or `Null`. This operation call is prefixed by `!`, meaning that this method should never raise an exception. Indeed, dealing with primitive objects

[7] https://tc39.es/ecma262/2021/#sec-reference-record-specification-type

prevents from getting an exception[8] as a result of the cast operation. In Skel, the result of `toObject` is an object location. In line 8, we take the object value from the state with the monadic binder `/o`. Finally, Step 4.c. returns the result of the baseObj's `[[Get]]` object internal method applied to a name representing the referenced value's name and to a value resulting from the call to the `GetThisValue` method. In Skel, we split this step into three. First, the method `getThisValue` is called on the reference `v`, assigning its result to the variable `thisVal`. Note that we make explicit the call to `getThisValue`, as Skel requires function application to consider fully computed terms as arguments. The call to `getThisValue` is not pure as the assertion there[9] is not captured by typing (although we have checked that the reference is indeed a property reference at step 4). We thus use the `?` binder, in that case, to extract the pure value or propagate the abruption, if any. Second, once `baseObj` is set, we call its `_Get_` internal method. This method takes `v.__ReferencedName__` and `thisVal` as arguments. The call is prefixed by `?`, meaning, again, that if an abrupt result happens, it is propagated to the caller. Finally, we return the result of the call.

If V is not a property reference, then base must be an Environment Record. The Skel else branch, namely Step 5., follows the structure of Step 4. Notice that in 5.b, there is an assertion that we capture in line 14. It can be read as *"assert that is true that the reference base-value* `base` *is an Environment Record"*. The `ref_Type` is a function that takes a tuple of type `(reference, type_ref)`, returning a boolean. The `type_ref type` constructor **`T_R_ER`** represents references with Environment Records as base values. The `assert_true` is a function that takes as arguments a function `f` and its arguments `a`. It propagates an anomaly when the application `f a` is **`False`** or continues to compute in case of a **`True`** judgment.

As with most of the algorithms in the Skel mechanization, we need to return a result that is out of the control-flow monad type. The following piece of code thus surrounds the whole algorithm.

```
let result =< (* GetValue's algorithmic steps *) in
result
```

4.6 Current Status

We defined a three-step entry-point to the mechanization. First, we create the environment in which the code will be evaluated by initializing the interpretation environment.[10] This operation creates the *Realm*, a record to which all the evaluated ECMAScript code is associated, the *Intrinsic Objects*, the main *Execution Context*, the *Global Environment*, and the *Global Object*, which can be considered as the standard library of JS. Second, we parse the script,[11] and finally, we evaluate it.[12] Executing these three steps required implementing a significant fraction of the specification, mostly from [8, Chapters 6 to 10, 18, 19, 20]. Once the runtime environment of the specification is defined, extending the Skel mechanization is straightforward.

Concerning the language itself, we defined in Skel a large subset of ES Syntax and its evaluation. Unfortunately, no existing parser of JS provides a faithful representation of ES Syntax, which is very verbose. We thus had to choose between implementing our own parser or translating the AST provided by off-the-shelf parsers. We chose the latter. More precisely, we use a parser that conforms to the *SpiderMonkey* Parser API,[13] which is used by all the parsers we have tried. We chose the Flow Parser[14] library because it is written in OCaml, which is the language we can instantiate our interpreter into, and because it is used to manipulate JS in an industrial setting. When instantiating the interpreter, we define transformations that, given a Flow AST, produce a well-typed ES AST.

We implemented most of the *Statement*s and *LexicalDeclaration*s ([8, Chapter 14]). Indeed, these two syntactic productions define what a script is ([8, Section 16.1]). Regarding *Expression*s ([8, Chapter 13]), we formalized most of the chapter. We are still working on the semantics of some *CallExpression*s and of generator and asynchronous function definitions ([8, Sections 13.3, 15.5, 15.6, 15.8, 15.9]), as their evaluation explicitly manipulates continuations of the interpreter code. This issue prevents us from starting to test the interpreter via the ES262 test suite, as function calls are required for testing. Nevertheless, understanding how the function calls are specified led to the discovery of some places where the manipulation of execution contexts was not properly described or handled. [15] This resulted in some clarifications and bug fixes, such as creating a new execution context for generators.[16]

4.7 Interpreter Instantiation and Evaluation

4.7.1 Interpreter Instantiation. To derive an interpreter from the ES skeletal semantics, we use the tool `necroml` that generates an OCaml interpreter from a Skeletal Semantics. The resulting artifact defines four module signatures (`TYPES`, `MONAD`, `UNSPEC`, and `INTERPRETER`) and two functors (`Unspec` and `MakeInterpreter`).

The four module signatures respectively represent the unspecified types, the interpretation monad, the unspecified terms, and the interpreter module describing every type and term defined in the Skel file. The first functor `Unspec` takes as input the instantiation of unspecified types (of signature `TYPES`) as well as an interpretation monad (of signature `MONAD`), and it returns a module that extends the provided unspecified types with a shallow embedding of the specified types in OCaml. This module also includes default implementation for every unspecified term that simply raises a **`NotImplemented`** exception. One may then include this module, override the unspecified terms with their instantiation, then give the resulting specification module as input to the `MakeInterpreter` functor. We now detail this approach for our ES semantics.

Although the ES specification in Skel consists of 15k LOC, only six types are left undefined. We show the instantiation of the `TYPES` module in Figure 12. Notice that we could have left unspecified four data types out of six by writing a library defining the `list` and `intMap` in Skel.

The interpretation monad encapsulates the Skeletal Semantics' behavior itself, providing, for example, the semantics of the branch

[8] In case of an exception, a specification anomaly is then raised.
[9] https://tc39.es/ecma262/2021/#sec-getthisvalue
[10] https://tc39.es/ecma262/2021/#sec-initializehostdefinedrealm
[11] https://tc39.es/ecma262/2021/#sec-parse-script
[12] https://tc39.es/ecma262/2021/#sec-runtime-semantics-scriptevaluation
[13] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
[14] https://github.com/facebook/flow
[15] https://github.com/tc39/ecma262/issues/2400, https://github.com/tc39/ecma262/issues/2409
[16] https://es.discourse.group/t/execution-context-suspension-and-resumption/756, https://github.com/tc39/ecma262/pull/2398

```
module IntMap = Map.Make (Int)

module rec T : sig
  type bigInt = Big_int.t
  and float = Float.t
  and int = Int.t
  and 'a intMap = 'a IntMap.t
  and 'a list = 'a List.t
  and string = String.t
end = T
```

Figure 12: TYPES Module Instantiation

```
exception NotImplemented of string
exception Fail of string

module M =  struct
  exception Branch_fail of string
  type 'a t = 'a
  let ret x = x
  let rec branch l =
    begin match l with
    | [] -> raise (Branch_fail "No branch matches")
    | b1 :: bq ->
       try b1 () with Branch_fail _ -> branch bq
    end
  let fail s = raise (Branch_fail s)
  let bind x f = f x
  let apply f x = f x
  let extract x = x
end
```

Figure 13: MONAD Module Instantiation

construct. Several options exist, such as executing each branch in order until one succeeds or running them all and collecting the results as a list. Our interpreter currently uses the simple identity monad (where branches are run in order). We can easily change this monad if need be. We show the identity monad implementation in Figure 13. In this monad, terms have pure types, and skeletons have monadic types `'a t`. We call this identity monad; the type of skeletons is the same as the types of terms. The most relevant part of the figure is the description of the `branch` operator that tries each branch in turn until it succeeds.

Applying the `Unspec` functor to the modules of Figures 12 and 13 creates a module of signature `UNSPEC` containing every type of the Skeletal Semantics and default implementations for unspecified terms that simply raise **`NotImplemented`** exceptions. Having these defaults is most useful in the case of our formalization, as there are 450 unspecified terms in our skeletal semantics. Having to instantiate them all before running the simplest example is very inconvenient. We can instead gradually instantiate them, testing the code as we go along.

To run the code, we thus instantiate some unspecified functions and give the resulting module to the `MakeInterpreter` functor. We thus have access to all the specified functions we have defined, including the entry point `runJS`. We also need to transform a JavaScript program into its AST to run it. To this end, we transform the Flow Parser AST into the (much more complex) AST used by ES, as explained above.

4.7.2 Evaluation. We propose two ways of evaluating this work. The first one is to compare our work with previous ones (Framework Comparison), and the second is to describe what JS programs we can execute (Program Execution).

Framework Comparison. One of the goals of this work is to show the visual closeness and maintainability of the proposed formalization. We compare our work with $\mathbb{K}_{\text{JS}}$ [17] and JSCert [1] by considering the formalization of the `GetValue` internal algorithm. Note that both these formalizations are based on ECMAScript 5.1, an older version of the standard. The ECMAScript standard has tripled in size since then, and its algorithms are much more complex, including explicit continuation manipulation.

$\mathbb{K}_{\text{JS}}$ is a set of rewriting rules representing the ES semantics. This means that given an internal method, there is a subset of the rewriting rules matching the method's behavior. To simplify, we can say that the left-hand side component of the rule is rewritten into the right-hand side when the conditions allow it. The correctness of $\mathbb{K}_{\text{JS}}$ is attested by a great coverage of the ES262 test suite. Consider the `GetValue` algorithmic steps in ECMAScript 5.1 [17] in Figure 14. One can notice the differences between the method of the 2021 standard and the one of version 5.1. After careful reading, the old rule can be split into four cases defined by the types that the `GetValue`'s parameter `V` can take if it is a reference. In this case, `V` can be either a reference to a variable in the environment, to a property of an object, to the property of the *undefined* value, or to a property of a primitive value. In Figure 15, we present the $\mathbb{K}_{\text{JS}}$ implementation. The four rules concisely cover all the cases. In the case of `V` being a reference to the environment, the GetValue is rewritten in a GetBindingValue. If it is to an object property, the definition is rewritten to an object `Get`. In the latter two cases, instead of using the method `IsUnreasolvableReference` and `HasPrimitiveBase`, the authors define two rules, one in which `V` is Undefined, which corresponds to an unresolvable reference, and the other by matching the input parameter with a Primitive label, representing a subtype of the ES value type. Note that the case where `V` is a value is not covered as it cannot reduce in a rewriting / small-step approach.

We argue that our Skel formalization is better for the following reasons:

(1) We are very close to the textual description of the standard. Indeed, when translating it to Skel, we do not need to understand its behavior fully. This suggests that an automated translation, as done in [18], could be achieved.
(2) The formalization has not been updated since 2015.
(3) The K framework is complex, and external tools cannot easily manipulate it to produce different artifacts. Our Skel definition can be used to generate a Coq description, which is not possible with $\mathbb{K}$.

We show a part of the formalization of `GetValue` in JSCert in Figure 16.[18] The formalization is a Coq shallow embedding of ES in pretty-big-step semantics style [6]. As said before, the two main issues with JSCert are maintainability and usability. Regarding

[17] https://262.ecma-international.org/5.1/#sec-8.7.1
[18] Full code at https://github.com/jscert/jscert/blob/master/coq/JsPrettyRules.v#L5112

(1) If Type(V) is not Reference, return V.
(2) Let *base* be the result of calling GetBase(V).
(3) If IsUnresolvableReference(V), throw a **ReferenceError** exception.
(4) If IsPropertyReference(V), then
   (a) If HasPrimitiveBase(V) is **false**, then let *get* be the [[Get]] internal method of *base*, otherwise let *get* be the special [[Get]] internal method defined below.
   (b) Return the result of calling the *get* internal method using *base* as its **this** value, and passing GetReferencedName(V) for the argument.
(5) Else, *base* must be an environment record.
   (a) Return the result of calling the GetBindingValue (see 10.2.1) concrete method of base passing GetReferencedName(V) and IsStrictReference(V) as arguments.

The following [[Get]] internal method is used by GetValue when V is a property reference with a primitive base value. It is called using *base* as its **this** value and with property P as its argument. The following steps are taken:

(1) Let O be ToObject(*base*).
(2) Let *desc* be the result of calling the [[GetProperty]] internal method of O with property name P.
(3) If *desc* is **undefined**, return **undefined**.
(4) If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
(5) Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]] (see 8.10).
(6) If *getter* is **undefined**, return **undefined**.
(7) Return the result calling the [[Call]] internal method of *getter* providing *base* as the **this** value and providing no arguments.

Figure 14: ECMAScript 5.1 `GetValue(V)`

```
rule GetValue(@Ref(E:Eid,N:Var, Strict:Bool)) =>
     GetBindingValue(E,N,Strict)
rule GetValue(@Ref(O:Oid,P:Var, _)) =>  Get(O,P)
rule GetValue(@Ref(Undefined, P:Var, _ )) =>
     @Throw(@ReferenceError("GetValue",P))
rule GetValue(@Ref(B:Primitive,P:Var, _ )) =>
     GetPrimitive(B,P)
```

Figure 15: GetValue 5.1 Written in the $\mathbb{K}$ Framework

```
(** Get value on a reference (returns value) (8.7.1) *)

| red_spec_ref_get_value_value: forall S C v, (* Step 1 *)
    red_spec S C (spec_get_value v) (ret S v)

| red_spec_ref_get_value_ref_a: (* Steps 2 and 3 *)
    forall S C r (y:specret value),
      ref_is_unresolvable r ->
      red_spec S C (spec_error_spec native_error_ref) y ->
      red_spec S C (spec_get_value r) y

| red_spec_ref_get_value_ref_b_has_primitive_base:
    (* Steps 2 and 4 *)
    forall v S C r o1 (y:specret value),
      ref_is_property r ->
      ref_base r = ref_base_type_value v ->
      ref_has_primitive_base r ->
      red_expr S C (spec_prim_value_get v (ref_name r)) o1 ->
      red_spec S C (spec_get_value_ref_b_1 o1) y ->
      red_spec S C (spec_get_value r) y
```

Figure 16: GetValue 5.1 in JSCert

```
1  let a = [,"one",,,"two",,3];
2  let o = new Object();
3  o.name = "o";
4  o["vec"] = a;
5  (o["vec"])[0] = "modified in line 5";
6  o.vec[0] == a[0]
```

Figure 17: Example of JS Program

the latter, the semantics is too big to be used in the context of Coq induction and inversion (Coq runs out of memory). Using a deep embedding (where the recursive evaluation of the language semantics is not captured using Coq's induction) would help avoid the issue, but doing so requires rewriting the semantics. As outside tools do not easily manipulate Coq descriptions, this motivated us to redo the semantics in a framework where this issue could be circumvented, hence the current work.

To conclude, neither $\mathbb{K}_{\text{JS}}$ nor JSCert are easily maintainable, as their approaches are too embedded in the tools they are using. By having a simple but powerful specification language, we can systematically write code very close to the standard and easy to maintain. We migrated from ECMAScript 2020 to ECMAScript 2021 in less than a week of work by only updating the algorithmic steps that have been modified.

Program Execution. After instantiating the interpreter, we started to test it gradually. We show, in Figure 17, an example of such a test program: a non-trivial manipulation of an object by defining it and modifying its properties via property accessors. In the first line of the program, we define an array by assigning to the variable `a` an array literal with missing items. The right part evaluates into an array object value that does not have a mapping to values in positions 0, 2, 3, and 5. The indices 1, 4, and 6 are mapped respectively

to the string `"one"`, the string `"two"`, and the number `3`. Then, in the second line, we create a new ordinary object and assign it to the variable `o`. We define two properties in the following lines. The property `"name"` is mapped to the string value `"o"`. We write the access to the property in dot notation style, where `o.name` is a way to access the property `"name"`. As the objects are partial mappings from property names to values, when a property is not yet defined, inspecting it would return **`undefined`**. Then, the property `"vec"` maps to a reference to the array object `a`, instantiated in the second line of the program. We modify the value of the property `"vec"` at index 0 by assigning it the string `"modified in line 5"`. Note that we use different ways to access properties for testing purposes. The evaluation of the comparison expression in line 6 results in **`true`**.

We wrote different programs, similar in size, to test non-trivial features of the semantics of expressions and statements we implemented. The programs always execute as expected.

5 CONCLUSION AND FUTURE WORK

We have described how the Skel language can be used to mechanize complex semantics. In particular, we have shown how carefully chosen monads can help to have a formal description close to the specification while still describing the complex behaviors of ES. We decided to formalize JavaScript because of two main reasons. First, we have previous experience dealing with the ES specification [1]. Second, the ES specification is both very precise and very complex. Hence, we do not have to guess the behavior of the language while making sure Skel can scale. In addition to implementing part of ES in Skel, we also provide boilerplate code to integrate with an existing JS parser and implement all the unspecified terms in OCaml. Using `necro-ml` [2], we generate an OCaml executable semantics.

Our main effort stood in mechanizing the foundations of the language, of which `GetValue` is a good example. We can now easily extend our formalization by following the same systematic approach for other abstract algorithms and language constructs.

We have also been able to start the evaluation of the maintainability of the approach. This project began with the formalization of ECMAScript 2020, but after a few months, we switched to the newer ECMAScript 2021. This process took only a few days of work. Naturally, this will change once we complete the formalization of the standard. Nevertheless, we can produce a list of differences between specifications (as a textual diff), making the amount of work proportional to the extent of changes and not the specification's size. In particular, we only have one mechanization to change instead of updating two Coq developments and a correctness proof, as is the case for JSCert. The standard changes quite often, not only by modifying algorithmic steps but also by introducing side-effects into the specification.

In this paper—in the context of JS—and in another work [10], we showed that it is possible to systematically introduce side-effects into a formalization by delegating all the necessary machinery to handle them to some carefully designed monads. This is a well-known result [20], but it was unclear whether it could be applied in the context of Skel. The monadic approach lets us introduce a new behavior throughout the specification by simply changing the monad we use. Thus, a short-term goal is to implement a new monad for manipulating delimited continuations and combining it with the ones presented in this paper. We will then be able to implement generators whose execution can be suspended (using `yield`) and resumed. This feature is necessary to implement function calls as its algorithmic description uses a generator. Once function calls are implemented, we will be able to directly run the ECMA-262 test suite.

In the long run, we plan several developments for this work. The first goal is to validate that we can use our mechanized semantics to prove some properties of the language. One property of interest is the guarantee that assertions in the specification are always satisfied. To this end, we will use the `necro-coq` tool [13]. As the formalization of JS helped us refine the Skel language, we believe the generation of a Coq semantics will provide many opportunities to suggest improvements for `necro-coq`. Another goal is to exploit our systematic approach to produce a Skeletal Semantics directly from the ES document, using a tool similar to the one presented by *J. Park et al.* [18]. Finally, we plan to apply our expertise to formalize other complex languages in Skel. An example could be Python, for which a formal semantics has been described on paper [11].

REFERENCES

[1] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 49 (01 2014), 87–100. https://doi.org/10.1145/2578855.2535876

[2] Martin Bodin, Nathanael Courant, Enzo Crance, and Louis Noizet. 2022. Necro Ocaml Generator. https://gitlab.inria.fr/skeletons/necro-ml

[3] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages* 44 (2019), 1–31. https://doi.org/10.1145/3290357

[4] Martin Bodin, Thomas Jensen, and Alan Schmitt. 2015. Certified Abstract Interpretation with Pretty-Big-Step Semantics. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (Mumbai, India) *(CPP '15)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/2676724.2693174

[5] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *The Web Conference 2018*. International World Wide Web Conferences Steering Committee, Lyon, France, 1–9. https://doi.org/10.1145/3184558.3185969

[6] Arthur Charguéraud. 2013. Pretty-big-step semantics. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*. Springer, Rome, Italy, 41–60.

[7] Olivier Danvy. 1991. *Three Steps for the CPS Transformation.* Technical Report CIS-92-2. Kansas State University.

[8] ECMA. 2021. ECMAScript 2021 Language Specification. https://tc39.es/ecma262/2021/

[9] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. *ECOOP, Lecture Notes in Computer Science* 6183 (06 2010), 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[10] Adam Khayam and Alan Schmitt. 2022. A Practical Approach for Describing Language Semantics. Submitted for publication.

[11] Raphaël Monat. 2021. *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries.* Ph. D. Dissertation. Sorbonne University.

[12] Louis Noizet. 2022. Necro Debugger Generator. https://gitlab.inria.fr/skeletons/necro-debug

[13] Louis Noizet. 2022. Necro Gallina Generator. https://gitlab.inria.fr/skeletons/necro-coq

[14] Louis Noizet. 2022. Necro Library. https://gitlab.inria.fr/skeletons/necro

[15] Louis Noizet and Alan Schmitt. 2022. Semantics in Skel and Necro. Submitted for publication.

[16] Louis Noizet and Alan Schmitt. 2022. *Stating and Handling Semantics with Skel and Necro.* Technical Report. Inria Rennes - Bretagne Atlantique. 20 pages. https://hal.inria.fr/hal-03543701

[17] Daejun Park, Andrei Stefanescu, and Grigore Rosu. 2015. KJS: a Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 346–356. https:

//doi.org/10.1145/2737924.2737991

[18] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 647–658.

[19] Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-Passing Style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*. ACM, 288–298. https://doi.org/10.1145/141471.141563

[20] Philip Wadler. 1992. The Essence of Functional Programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992*, Ravi Sethi (Ed.). ACM Press, 1–14. https://doi.org/10.1145/143165.143169