JSkel: Towards a Formalization of JavaScript's Semantics

Adam Khayam, Louis Noizet, and Alan Schmitt

Inria

Abstract

We present JSkel, a formalization of the semantics of JavaScript in Skel, the concrete language used to write skeletal semantics. We describe the improvements to Skel we designed and implemented to significantly simplify the formalization. We show the formalization is both close to the specification and executable.

1 Introduction

Mechanizing the semantics of complex languages is useful, not only to make it precise, but also to obtain a framework in which one may prove properties of programs or of the language itself. The success of such a mechanization may be evaluated in two ways: is the actual semantics of the language really captured, and can the formalization be used in practice?

There are two main ways to make sure the semantics is captured: 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. Thus, a suitable tool to mechanize semantics should make it easy to stay close to language definitions, be they 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 [2] is suitable to mechanize semantics. To this end, we describe the ongoing formalization of JavaScript as a skeletal semantics. In this paper, we only address the question of capturing the semantics. We leave the evaluation of the usability of the formalization for future work.

JSkel is the first formalization of a real-world programming language in Skel and it has been a core support in the evolution of Skel, from the introduction of the notion of polymorphism to the addition of monads and first-class functions. The evolution of Skel is motivated by the need of having a formal semantics and a language expressive enough to be able to capture the behavior of JavaScript, ensuring a visual and a behavioral match between the formalization and the specification.

We present existing approaches to mechanize languages in Section 2. We then introduce Skeletal Semantics as well as our concrete syntax in Section 3. Our main contribution is the description of the mechanization of JavaScript in Section 4, where we also present some crucial features of skeletal semantics that significantly improve the readability of the semantics. We conclude and explore future work in Section 5.

2 Context

We first motivate why we chose JavaScript 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 a large 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.

In addition, we have a previous experience in mechanizing JavaScript in Coq, called JSCert [1]. Defining a semantics in a proof assistant is the most direct way of mechanizing it. 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 [4] 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 JavaScript programs, whereas the recursive definition can provide an OCaml interpreter using Coq's extraction mechanisms. The mechanization is fairly 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. In addition, and most importantly, JSCert uses Coq's induction to represent the recursive evaluation of the language. This shallow embedding results in a definition of a semantics as an inductive of about 1000 rules, which is too large to prove properties of the language. The exploration of ways to address these issues was the motivation to create skeletal semantics.

To simplify the mechanization, [6] introduces a core language λ_{JS} that embodies JavaScript essentials, and a desugaring process that transforms a JS program to a λ_{JS} one. This work is based on ECMA 3 and 5. The authors empirically prove that the desugared version of JS is compliant with the implementations of JS itself using JavaScript's test suites. In spite of its efficiency and correctness, λ_{JS} is a feature-based redefinition of ECMA. This formalization does not follow the specification, making it hard to gather evidence it actually captures the language beyond running the tests. We attempted to formalize λ_{JS} in Coq[1], uncovering several issues with the desugaring process that were not witnessed by testing.

An alternative approach to the formalization of JavaScript is the use of an existing framework. $\mathbb{K}$JS [11] 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 to analyse 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 JavaScript has significantly evolved since ES 5.1 (the specification has more than doubled in size), the mechanization has not been updated. Our experience in updating JSExplain [3], a JavaScript interpreter written in OCaml, from ES5.1 to ES6 has shown us it is far from a trivial issue. The power of the $\mathbb{K}$ framework comes at the cost of additional complexity in the description of languages in it, which hinders maintainability and closeness to the specification.

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 to leave some constructions undefined, to let them be implementation dependent or for gradual specification.

The theoretical concept behind skeletal semantics was presented in [2]. The version we show here has been significantly improved, but it is still work in progress, in particular regarding some

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

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

Figure 1: skeleton for the `While` constructor

theoretical aspects such as the definition of abstract interpretations.

The "Skel" language, which has been defined to describe skeletal semantics, serves as support for the necro ecosystem [9], which provides a generator of OCaml interpreters [7], a generator of gallina formalization [8], and a (work in progress) generator of TEX inference rules [10].

The Skel formalization of JavaScript, which constitutes the main topic of this article, has had a lot of impact on the language itself, as it prompted us to make several improvements to Skel, such as adding polymorphism and first-class functions.

Figure 1 shows an example of a skeleton for the evaluation of a *while* block. We see all the main elements of skeletal semantics and we can observe that they are actually the main elements of any semantics.

- Recursion (`eval_stmt` and `eval_expr`) lets us define the semantics depending on the semantics of other terms (frequently subterms, but not always, as we can see in this example with the call `eval_stmt` $(\sigma', \texttt{While}\ (e,t))$).
- There are auxiliary functions and auxiliary types such as booleans, and possibility to match a boolean variable against a given constructors (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. Non-mutually exclusive branches lets us also represent non-deterministic semantics (such as λ-calculus with no evaluation strategy).

3.1 Formalism

We ignore polymorphism in the formal definition below to keep it readable. The use of polymorphism is pretty intuitive though, and its semantics is the usual one. We first give the syntax of skeletal semantics.

$$\begin{array}{rcll}\textsc{Term} & t & ::= & x_i \mid C\ t \mid (t, \dots, t) \mid \lambda x : \tau \cdot S \\ \textsc{Skeleton} & S & ::= & x_i\ t_1 \dots t_n \mid \text{let}\, p = S\, \text{in}\, S \mid (S..S) \mid t \\ \textsc{Pattern} & p & ::= & x_i \mid \_ \mid C\ p \mid (p, \dots, p) \\ \textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau, \dots, \tau)\end{array}$$

A term is either a variable, a constructor applied to a term, a tuple of terms, or an abstraction whose body is a skeleton (where the type of the abstracted variable is explicitly given). Terms can be viewed as expressions in their normal form (that is with no reduction possible). 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 (sometimes, we write ret t to say explicitly that we consider the skeleton and not the term). A pattern is either a variable name, a wildcard, a constructor applied to a pattern, or a tuple 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.

This syntax is very close to Abstract Normal Forms [5]. The main difference is that we add branching as a non-deterministic choice. We also provide constructors and tuple that are not in the basic ANF, and we allow let bindings in let bindings. Of course, the let bindings can always be extracted by using a fresh variable name, which the function `necrotrans extractletin` [9] actually does.

A skeletal semantics is a list of type *declarations*, either *unspecified* or *specified* (by giving its constructors), and a list of term declarations, also either unspecified and containing only a name and type, or specified with an additional term.

We give the following four examples, written in Skel :

```
type int                              term add: int → int → int

type nat = | Zero | Succ nat          term two:nat = Succ (Succ Zero)
```

The possibility to declare non-specified 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 sometimes, we want it to remain unspecified. The partial specifications allows for that.

As shown above, the Skel language is really light (close to λ-calculus), yet very powerful. A good proof of this power is the semantics of JS given in Section 4.

To furthermore improve readability, the Skel language also provide notation for binding operators. That is, assuming a term declaration for `bind`, one can write `let%bind p = x in v` for `bind x (λ v -> let p = v in s)`. It is particularly useful when writing a semantics that heavily uses monadic constructions, such as the one for JavaScript.

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 arrow types for functions.

We give the typing rules for terms and skeletons in Figure 2. They are respectively of the form $\Gamma \vdash_t t : \tau$ and $\Gamma \vdash_S S : \tau$, where Γ is a typing environment (a partial functions that binds variable names to types). The initial typing environment Γ_0 is the function that binds every term name (both specified and non-specified) to the associated type given in the term declaration list.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_t x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash_t t : \tau \qquad \mathit{ctype}(\mathrm{C}) = (\tau, \tau')}{\Gamma \vdash_t \mathrm{C}\, t : \tau'}\ \textsc{Const}$$

$$\frac{\Gamma \vdash_t t_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash_t t_n : \tau_n}{\Gamma \vdash_t (t_1, \dots, t_n) : (\tau_1, \dots, \tau_n)}\ \textsc{Tuple} \qquad \frac{\Gamma + x \leftarrow \tau \vdash_S S : \tau'}{\Gamma \vdash_t (\lambda x : \tau \cdot S) : \tau \to \tau'}\ \textsc{Clos}$$

$$\frac{\Gamma \vdash_t t : \tau}{\Gamma \vdash_S \mathrm{ret}\, t : \tau}\ \textsc{Ret} \qquad \frac{\Gamma \vdash_S S_1 : \tau \qquad \dots \qquad \Gamma \vdash_S S_n : \tau}{\Gamma \vdash_S (S_1 \dots S_n) : \tau}\ \textsc{Branch}$$

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

$$\frac{\forall i\ \Gamma \vdash_t t_i : \tau_i \qquad \Gamma(x) = \tau_1 \to \dots \to \tau_n \to \tau}{\Gamma \vdash_S (x\ t_1 \dots t_n) : \tau}\ \textsc{App}$$

Figure 2: Typing rules of Skeletal Semantics

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 $\mathit{ctype}(C)$ 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. Finally, to type an abstraction, we type the skeleton, and return the arrow from the type of the argument to the type of the skeleton.

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

- $\Gamma + x \leftarrow \tau$ is the partial function Γ' s.t. $\Gamma'(x) = \tau$ and for all $y \neq x$, $\Gamma'(y) = \Gamma(y)$.
- $\Gamma + \_ \leftarrow \tau = \Gamma$.
- $\Gamma + C\ p \leftarrow \tau' = \Gamma + p \leftarrow \tau$ if $\mathit{ctype}(C) = (\tau, \tau')$.
- $\Gamma + (p_1, \dots, p_n) \leftarrow (\tau_1, \dots, \tau_n) = \Gamma + p_1 \leftarrow \tau_1 + \dots + p_n \leftarrow \tau_n$.

Finally, to type n-ary function application, we require the variable in function position to be 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 JSkel formalization below. Polymorphism is always explicitly declared. Thus, when declaring a polymorphic type or term, one has to give its type arguments. For example, here are the declaration of the `list` type and a `map` term.

```
type list<a> =
| Nil
| Cons (a, list<a>)
```

```
term 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
        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 first explain how types are translated into sets of concrete values. Then we define the evaluation relation, which relates every term or skeleton to concrete values of their type.

Formally, for each type τ we define the set of concrete values V_τ in the following way.

- If τ is a non-specified type, V_τ must be given.
- If τ is a specified type, V_τ is the set freely generated by the constructors of type τ, where the argument of each constructor is recursively generated.
- $V_{(\tau_1,\ldots,\tau_n)} = V_{\tau_1} \times \cdots \times V_{\tau_n}$.
- $V_{\tau_1 \to \tau_2} = \mathcal{R}(V_{\tau_1}, V_{\tau_2}) = \mathcal{P}(V_{\tau_1} \times V_{\tau_2})$.

Skel being a strongly typed language, every term and every skeleton has a unique type, with the exception of 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. In a sense, evaluation of a term is 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. The initial environment E_0 is an environment that binds every unspecified term to a value. We denote this evaluation as $\Downarrow_t$ for terms and $\Downarrow_S$ for skeletons.

To evaluate a variable, we look for its value in E. 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. Finally, the evaluation of an abstraction is a subset of the relation R in which v relates to w if and only if the skeleton can be evaluated to w, given that the abstracted variable is bound to v in the environment. It is convenient to only return a subset of the relation as we can then only consider the arguments to which the function is actually applied.

To evaluate a skeleton which is simply a term, we use the evaluation rules for terms. To evaluate a branching, we return the value v of a branch that successfully evaluates to v. To evaluate the application of a variable to terms, we look up the variable in the environment and get a relation. We then evaluate the first term and check that the returned value is in the relation, which provides a result value. If the evaluation is partial and additional terms are present, this value is in turn a relation and we continue. We thus write $R^*(v_1, \ldots, v_n)$ for

$$\frac{E(x) = v}{E, x \Downarrow_t v}\ \textsc{Var} \qquad \frac{\texttt{term x = t} \qquad E_0, t \Downarrow_t v}{E, x \Downarrow_t v}\ \textsc{Term} \qquad \frac{E, t \Downarrow_t v}{E, (\mathrm{C}\, t) \Downarrow_t \mathrm{C}\, v}\ \textsc{Const}$$

$$\frac{E, t_1 \Downarrow_t v_1 \qquad \dots \qquad E, t_n \Downarrow_t v_n}{E, (t_1, \dots, t_n) \Downarrow_t (v_1, \dots, v_n)}\ \textsc{Tuple}$$

$$\frac{R \subseteq \{(v, w) \mid v \in V_\tau \wedge (E + x \leftarrow v), S \Downarrow_S w\}}{E, (\lambda x : \tau \cdot S) \Downarrow_t R}\ \textsc{Clos} \qquad \frac{E, t \Downarrow_t v}{E, \mathrm{ret}\, t \Downarrow_S v}\ \textsc{Ret}$$

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

$$\frac{\forall i\ \ E, t_i \Downarrow_t v_i \qquad E(x) = R \qquad R^*(v_1, \dots, v_n, w)}{E, (x_i\ t_1 \dots t_n) \Downarrow_S w}\ \textsc{App}$$

Figure 3: Concrete Interpretation of Skeletal Semantics

$\exists R_1 \dots \exists R_n . R(v_1, R_1) \wedge R_1(v_2, R_2) \wedge \dots \wedge R_n(v_n, w)$ corresponding to the curried application. 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.

- $E + x \leftarrow v$ is the partial function E' s.t $E'(x) = v$ and for all $y \neq x$, $E'(y) = E(y)$.
- $E + \_ \leftarrow v = E$.
- $E + (C\ p) \leftarrow C\ v = E + p \leftarrow v$.
- $E + (p_1, \dots, p_n) \leftarrow (v_1, \dots, v_n) = E + p_1 \leftarrow v_1 + \dots + p_n \leftarrow v_n$.

To evaluate any term or skeleton, we assume given an initial environment E_0 that maps every non-specified term to a concrete value of the appropriate type.

It is easy to check that if $\Gamma \vdash_t t : \tau$ and $E, t \Downarrow_t v \in V_\tau$, then it entails that $v \in V_\tau$, given that Γ matches E, that is $\forall x$, $\Gamma(x) = \tau \rightarrow E(x) \in V_\tau$.

4 JSkel

In this section, we present the Skel formalization of the `GetValue`[2] method written in Skel. As a first step, we present the JSkel types and helper functions designed for a visually faithful formalization. We use a monadic approach to propagate information implicitly. We claim the result is a simple yet powerful tool.

JSkel is still work in progress. Nevertheless, we are already able to instantiate our interpreter to run basic code, namely a subset of the Expression[3] grammar production, statements such as

[2]https://tc39.es/ecma262/#sec-getvalue
[3]https://tc39.es/ecma262/#prod-Expression

the Expression Statement[4], the Variable Statement[5], and the Empty Statement[6], and Lexical Declarations[7]. We define a specification entry-point that first creates the ES initial environment, then parses and evaluates the script.

4.1 Towards a formalization

4.1.1 ECMAScript

ECMAScript is a large vernacular specification written in an imperative style. It is divided into 28 chapters and 6 appendices. Despite its complexity and verbosity, it provides a complete specification of the behavior of JavaScript. Some choices are left to the implementation, however, so a formalization has to take into account design choices that cannot be considered standard. We explain below how we deal with them.

After an introductive part in chapters 1 to 5, where the notational and algorithmic conventions are defined, the document can be divided into three main functional groups.

Chapter 6 to 9 give a taxonomy of the 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 and iterator operations), the *runtime environment*, 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.

The central part of the specification, chapter 10 to chapter 16, describes the actual ES programming language. Chapters 10 and 11 focus on lexical units. Language constructs are given in chapters 12 to 15, where each construct is given with its syntactic specification and its evaluation. These describe expressions, statements, functions, and scripts. Modules are defined in chapter 16.

Chapters 17 to 27 introduce the default components of the *Global Object*, which can be considered as the standard library of JavaScript. In addition, a memory model is given in Chapter 28.

4.1.2 Challenges of the Formalization

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

First, there is a notion of 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 left unspecified, as well as the functions to access and set it. This is representative of our general approach to have all the "implementation dependent" parts of the ES specification kept unspecified. To remain close to the imperative specification, we design a Skel state monad in Figure 4. This monad lets us implicitly pass the state around.

Second, the specification often breaks the usual control flow by having `return` in the middle of algorithms. We can capture such control flows by using nested branches (see below), but this

[4] https://tc39.es/ecma262/#prod-ExpressionStatement
[5] https://tc39.es/ecma262/#prod-VariableStatement
[6] https://tc39.es/ecma262/#prod-EmptyStatement
[7] https://tc39.es/ecma262/#prod-LexicalDeclaration

```
type state (*implementation dependent*)
type st<α> := state → (α, state)

term st_bind<α,β>: (v: st<α>) → (f: α → st<β>) → st<β> =
    λ s: state → let (v', s') = v s in f v' s'

term st_ret<α>: (v: α) → st<α> = λ s :state → (v, s)
```

Figure 4: State Monad in Skel

Field Name	Value
[[Type]]	One of **normal**, **break**, **continue**, **return**, or **throw**
[[Value]]	Any **ECMAScript language value** or **empty**
[[Target]]	Any **ECMAScript string** or **empty**

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

```
type completionValue<α> =
| Ok α
| Abruption maybeEmpty<value>

type completionTarget :=
    maybeEmpty<string>
```

Figure 5: ES Completion Record and Skel Formalization

significantly reduces the legibility of the mechanization. We thus define 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 4.1.3 an exception monad that captures the behavior of the ES's monadic shorthands `?` and `!`, and in Section 4.1.4 its extension to handle control-flow features.

We claim the combination of the state monad with the control-flow and the exception ones greatly simplifies our code, making JSkel easy to write, maintain and to visually compare to ES. This is shown is Section 4.2.

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

Most ES operations do not directly return values, they instead return *completion records*. A Completion Record describes the runtime propagation of values and control flow. This record is composed of three fields, as depicted in Figure 5. In a nutshell, a completion record indicates what to do (this is a result, a break out of a loop, a return of a function, an exception being thrown), it contains an optional value, and in the case of a break or continue, it contains 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 other constructions. 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>`.

```
type out<α> =
| Success completionRecord<α>
| Anomaly anomaly
```

```
type anomaly =
| AbruptAnomaly completionRecord<()>
| StringAnomaly string
| NotImplemented
```

Figure 6: Out and Anomaly Declarations

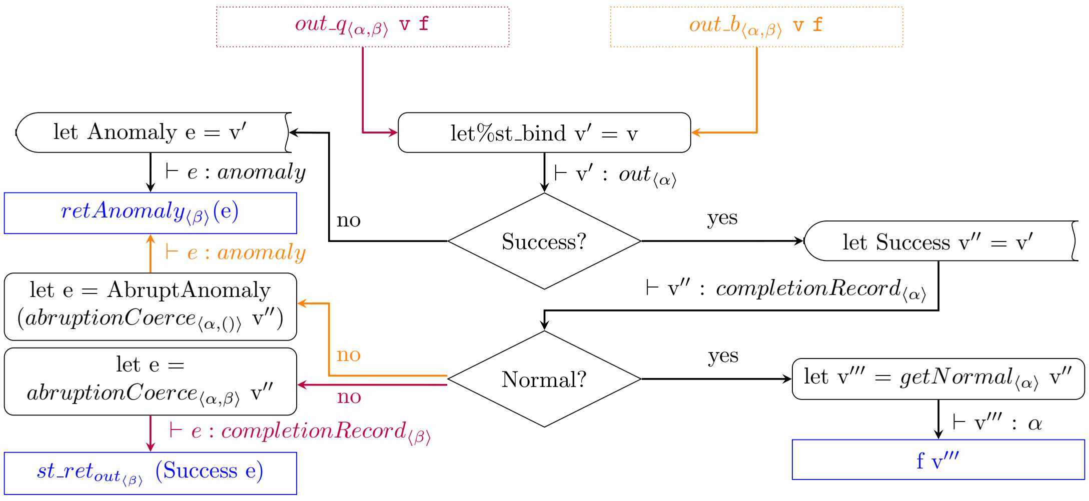
$out\_q_{\langle\alpha,\beta\rangle}$ v f
$out\_b_{\langle\alpha,\beta\rangle}$ v f
let Anomaly e = v′
let%st_bind v′ = v
⊢ e : anomaly
⊢ v′ : $out_{\langle\alpha\rangle}$
$retAnomaly_{\langle\beta\rangle}$(e)
no
Success?
yes
let Success v″ = v′
⊢ e : anomaly
let e = AbruptAnomaly ($abruptionCoerce_{\langle\alpha,()\rangle}$ v″)
⊢ v″ : $completionRecord_{\langle\alpha\rangle}$
no
let e = $abruptionCoerce_{\langle\alpha,\beta\rangle}$ v″
Normal?
yes
let v‴ = $getNormal_{\langle\alpha\rangle}$ v″
no
⊢ e : $completionRecord_{\langle\beta\rangle}$
⊢ v‴ : α
$st\_ret_{out_{\langle\beta\rangle}}$ (Success e)
f v‴

Figure 7: JSkel Model of `?` and `!`. Functions are presented in Figure B1

Figure 5 defines the types corresponding to the contents of this record: `completionType` holds control flow information, `completionValue` is either an `Ok` polymorphic constructor that contains the value of a non-abrupted computation, or an abruption. An ES abrupted 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. An ES completion record for values thus has type `completionRecord<maybeEmpty<value>>`.

We next define a type `out` composed of two constructors, `Success` for *successful computations*, and `Anomaly` for anomalies, which intuitively corresponds to a failure of the specification. A successful computation is one that returns an ES completion record, either `Normal` or `Abrupt`. Incorrect computations are captured by the `Anomaly` constructor. 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 `Abrupt`. The specification authors informally guarantee that these failures never occur. We make them explicit so that we can formally express their absence and thus pave the way for a formal certification of this property.

The `Anomaly` constructor is of type `anomaly`. We define the latter with three type constructors: `AbruptAnomaly` that holds a completion record in case an evaluation returns an unexpected abruption, `StringAnomaly` that contains a textual information about an anomaly—when an assertion of the specification is broken or when an implementation-dependent operation fails, and `NotImplemented` that signals that we have not yet implemented some feature.

ES abruptions are propagated through an abstract method called `ReturnIfAbrupt`. Basi-

```
type controlFlow<α,β> =              term cf_ret<β> : β → controlFlow<(),β>
| ContinueControl α                  term cf_cont<β> : () -> controlFlow<(),β>
| ReturnControl β                    term cf_assign<α,β> : α → controlFlow<α,β>

term cf_bind<α,β,γ> : controlFlow<α,β> → (α → controlFlow<γ,β>) → controlFlow<γ,β>
term cf_res<α,β,γ> : controlFlow<α,β> → (α → β) → β
```

Figure 8: The controlFlow type with binders and setters

cally, this method gets a completion record, and either returns the value of the `[[Value]]` field in case of a normal completion or 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 basically a monadic bind for an exception monad. The other operator used in the specification is `!`: it behaves like `?` 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 $\texttt{out\_q}_{\langle\alpha,\beta\rangle}$(red arrows), and $\texttt{out\_b}_{\langle\alpha,\beta\rangle}$(orange arrows). We define them as the monadic binders of the combination of `st` and `out`.

```
term (out_q|out_b)<α,β> : st<out<α>> → (α → st<out<β>>) → st<out<β>>
term out_ReturnIfAbrupt<α,β> : out<α> → (α → st<out<β>>) → st<out<β>>
```

The state monad is required for getting the result in case it is stored in the data structures in it, such as when manipulating references to values in the heap. We define, in Figure B1, getters (`getNormal, getAbrupt, ...`) for each `completionRecord`'s type, and returns (`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 then the identity, but it changes the type parameter of the completion record.

Given two algorithmic steps,

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

we represent them as

```
let%out_q bar = abstractOperation(foo) in
retNormal<τbar> bar
```

where τ_{bar} is the type of `bar`.

4.1.4 A Control-Flow monad

As said earlier, the ES specification uses imperative control flow, such as returning in the middle of an algorithm. We introduce, in Figure 8, the type `controlFlow<α,β>` for computations that either continue with an argument of type α or that terminate with a result of type β. It is composed of two constructors: `ReturnControl`, to return a result of type β, and `ContinueControl` to continue the execution with a value of type α to be given to the continuation.

We define a monadic binder `cf_bind`, a function `cf_res` for extracting the value from a `controlFlow<α,β>` term, and three return functions: `cf_ret` for `ReturnControl`, `cf_cont` for signaling a unit `ContinueControl`, and `cf_assign` for an α `ContinueControl`. We could

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%cf_res result =
  branch let True = foo in cf_ret<int> 1
  or     let False = foo in cf_cont<int> ()
  end;%cf_bind
  branch let True = bar in cf_ret<int> 2
  or     let False = bar in cf_cont<int> ()
  end;%cf_bind
  branch let True = bar in cf_ret<int> 3
  or     let False = bar in cf_cont<int> ()
  end;%cf_bind
  cf_ret<int> 4
in result
```

Figure 9: Code with and without Control Flow Monad. See Figure B2 for Functions

implement `cf_cont` using `cf_assign`, but we find that making the distinction explicit clarifies the code.

We illustrate in Figure 9 the translation of algorithmic steps in Skel, with and without the control-flow monad. In the examples, we use the form `let (True|False) = ... in ...` to test whether a value is `true` or `false`. 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 common 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 (highlighed on the left-hand side of Figure 9). 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 (*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 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<α,out<β>>> 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<β>> at the end of every function that may raise an exception. To this end, we start each algorithm with `let%cf_out 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 type st<cf<α,out<β>>>.

Despite the closeness to the algorithmic steps, the latter figure, in lines 4, 6 and 8, 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 having to do so, we define two

1. If foo is `true` throw `FooError`
2. If bar is `true` Return `2`
3. If baz is `true` Return `3`
4. Return `4`

```
let%cf_out result =
    branch let True = foo in
           let%throw fe=fooError<int> in fe
    or     let False = foo in cont<int> () end;%bind
    branch let True = bar in ret<int> 2
    or     let False = bar in cont<int> () end;%bind
    branch let True = bar in ret<int> 3
    or     let False = bar in cont<int> () end;%bind
    ret<int> 4
in result
```

Figure 10: Variant of the Figure 9 with Exceptions. See Figure B3 for Functions.

boolean binder terms, `ifpTrue` and `ifpFalse`, for introducing partiality in branchings. The `ifpTrue` binder, in the case v is `True`, applies the bind function `f` to a unit, otherwise, in the case v is `False`, the branch returns a continuation of type α. The `ifpTrue` term definition, and its application to the Figure 10 example, results in the following code.

```
term ifpTrue<α>:
 (v: boolean) →
 (f: () → st<controlFlow<(),out<α>>>
 ) →
 st<controlFlow<(),out<α>>> =

   branch let True = v in f ()
   or     let False = v in cont<α>()
   end
```

```
let%cf_out result =

  branch foo;%ifpTrue
         let%throw fe=fooError<int> in fe end;%bind
  branch bar;%ifpTrue ret<int> 2 end;%bind
  branch bar;%ifpTrue ret<int> 3 end;%bind
  ret<int> 4

in result
```

4.2 The `GetValue(V)` example

In this section, we illustrate our design choices through the mechanization in Skel of the `GetValue` abstract method. This method, presented in Figure 11, is one of the most referenced 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 it 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, Object, or Environment Record. The reference type is shown in Figure 12. It is a record with a field `[[BaseValue]]` of type `environmentRecord` or `value`, and additional fields not relevant here. Our mechanization in Skel leaves this type unspecified. We describe the types and terms used in our formalization of `getValue` in Figure B4.

We now describe step by step how we formalize this method (the code is collected as a single function in Figure B5). The type mixing values and references is the specified type `valref` defined as `Value value | Reference reference`. The argument of `GetValue` thus has type `out<valref>`. 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.

```
let%returnIfAbrupt v = v
```

If V is a normal completion, then the newly bound V will have type `valref`. We could use a different type for `returnIfAbrupt` to be able to write `let%bind v = returnIfAbrupt v`, but we would then need to specify the polymorphic type components of `returnIfAbrupt`. We are working on type inference to enable such a change in the future.

```
1. ReturnIfAbrupt(V).
2. If Type(V) is not Reference, return V.
3. Let base be GetBase(V).
4. If IsUnresolvableReference(V) is true, throw a ReferenceError exception.
5. If IsPropertyReference(V) is true, then
    a. If HasPrimitiveBase(V) is true, then
        i. Assert: In this case, base will never be undefined or null.
        ii. Set base to ! ToObject(base).
    b. Return ? base.[[Get]](GetReferencedName(V), GetThisValue(V)).
6. Else,
    a. Assert: base is an Environment Record.
    b. Return ? base.GetBindingValue(GetReferencedName(V), IsStrictReference(V)).
```

Figure 11: The ECMAScript's $GetValue(V)$

Field Name	Value
[[BaseValue]]	**Value** or **Environment Record**
[[Strict]]	Boolean Flag
[[ThisValue]]	**Value** or **Empty**
[[Name]]	String Value

Figure 12: The Reference type

The next step inspects the type of V. If it is not a reference, hence it is a value, we return it. Otherwise, the `ifpFalse` implicitly propagate a continuation.

```
branch valref_Type(v, T_Ref);%ifpFalse let Value v = v in ret<value> v end;%bind
```

The `GetBase` method in the third instruction takes the `[[BaseValue]]` from the reference V, and assigns it to base. The previous step guarantees that V has type `reference`, so we procede:

```
let Reference v = v in let base = getBase(v) in
```

Now base has type `ref_bv`. The step 4 follows the same pattern as step 2:

```
branch isUnresolvableReference v;%ifpTrue
      let%throw refErr = referenceError<value> in refErr end;%bind
```

An *unresolvable* reference is the one that has `Undefined` as `[[BaseValue]]`. In this case, a `ReferenceError` is thrown. Otherwise, step 5 inspects whether V is a *property reference*. Being a property reference means that a reference have a non-null, defined base value of type `value`.

```
branch let True = isPropertyReference v in
```

Step 5.a. checks whether the reference V has a *primitive* `[[BaseValue]]`. Having a primitive base means that the reference points to a primitive value, i.e., a value of type `Boolean`, `String`, `Symbol`, `BigInt`, or `Number`. After some assertions, this algorithmic step sets base as `ToObject` applied to base. Note the use of the %b monadic operator corresponding to the ! in the call: `ToObject` must return a normal result, otherwise an anomaly is raised. There is no `else` branch, but since base is not primitive in that case, it has to be an `Object`. We extract it using pattern matching. In both case, the resulting `loc_Object` is returned to the binder `let%bind base` using `assign`.

```
7    let%bind base =
8        branch let True = hasPrimitiveBase v in let R_Value base = base in
9               val_Type(base,T_Null);%assF val_Type(base,T_Undefined);%assF
10              let%b base = toObject(base) in assign<loc_Object,value>(base)
11       or     let False = hasPrimitiveBase v in
12              let R_Value (Obj base) = base in assign<loc_Object,value>(base)
13       end in
```

In line 9, we use the monadic binder `assF`⟨*value*⟩ for "assert false". If the result of the `val_type` application is **true**, an anomaly is raised.

Once base is set, the algorithmic step 5.b calls the *object internal method* `[[Get]]`. The JSkel formalization is straightforward with it, using base as the first argument to `o_Get`.

```
14   let name = getReferencedName(v) in let%q thisVal = getThisValue(v) in
15   let%q v' = o_Get(base, name, thisVal) in ret<value> v'
```

Note that we make explicit the order of the calls to `getReferencedName` and `getThisValue`, as Skel requires function application to be to fully computed terms. We could faithfully model different execution orders using branching, but this would make the mechanization confusing. The call to `getReferencedName` is pure, although the specification contains an assertion[8]. Since we know that the assertion is satisfied by typing, we do not include it. The call to `getThisValue`, however, 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 5). We thus use the %q binder in that case to extract the pure value or propagate the anomaly, if any.

If V is not a property reference, then base must be an Environment Record. In this last step, we check that this is the case using an assertion, then we apply the `getBindingValue` operation.

```
16 or     let False = isPropertyReference v in ref_Type(base, T_R_EnvRec);%assT
17        let R_EnvironmentRecord base = base in
18        let name = getReferencedName(v) in let strict = isStrictReference(v) in
19        let%q v' = er_GetBindingValue(base, name, strict) in ret<value> v'
20 end
```

As with most of the algorithms in our mechanization, we need to return a result that is out of the control monad. The whole algorithm is thus surrounded by the following piece of code.

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

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 in keeping the specification and

[8]https://tc39.es/ecma262/#sec-getreferencedname
[9]https://tc39.es/ecma262/#sec-getthisvalue

its mechanization very similar. The current state of our formalization of JavaScript can be found online, at `https://gitlab.inria.fr/skeletons/jskel`. It includes the Skel formalization of the language, boilerplate code to integrate with existing JavaScript parsers (see Appendix A), and an implementation of all the unspecified terms. Using `necro-ml` [7], we have all that is needed to generate an executable semantics.

To this point, our main effort has been in mechanizing the foundations of the language, of which `GetValue` is a good example. We are now ready to formalize more constructs than simple expressions and statements.

We also have been able to start the evaluation of maintainability of the approach. This project began with the formalization of ECMAScript 2020, but after few months, we switched to the newer ECMAScript 2021. This process took only a day of work. This may change once we have a more complete formalization of the specification, but as we can easily produce a list of differences between specifications, the amount of work is proportional to the size of changes and not to the size of the specification. In particular, we only have one mechanization to change, instead of two formal developments and a correctness proof as is the case for JSCert.

A short term goal is to formalize enough of JavaScript to have all the features needed to run the ECMA-262 test suite. A longer term goal is to validate that we can use our mechanized semantics to prove properties of the language. A first property of interest is the guarantee that assertions in the specification are actually satisfied. To achieve this goal, we will use the `necro-coq` tool [8]. As the formalization of JavaScript helped us refine the Skel language, we believe the generation of a Coq semantics will provide many opportunities to improve `necro-coq`.

References

[1] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, 49:87–100, 01 2014.

[2] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44:1–31, 2019.

[3] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. JSExplain: A Double Debugger for JavaScript. In *The Web Conference 2018*, pages 1–9, Lyon, France, Apr 2018.

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

[5] Olivier Danvy. Three steps for the cps transformation. Technical Report CIS-92-2, Kansas State University, 1991.

[6] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. *ECOOP, Lecture Notes in Computer Science*, pages 126–150, 06 2010.

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

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

[9] Louis Noizet. Necro Library, https://gitlab.inria.fr/skeletons/necro.

[10] Louis Noizet. Necro Library, https://gitlab.inria.fr/skeletons/necro-tex.

[11] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. *ACM SIGPLAN Notices*, 50:346–356, 06 2015.

```
type expression =
| Expr_AssExpr assignmentExpression
| Expr_Comma (expression, assignmentExpression)
```

Figure A1: Type definition of expressions in JSkel

A The parsing process

In JSkel, we formalize the JavaScript syntax as it is defined in ECMAScript. For instance, we give in Figure A1 the description of the Expression type[10], which is the same as its ES counterpart.

Unfortunately, no existing parser of JavaScript provides such a faithful representation. We thus had to choose between implementing our own parser or translating the AST provided by on-the-shelf parsers. We chose the latter.

More precisely, we use a parser that conforms to the *SpiderMonkey*'s Parser API[11], which is followed by all parsers we have found. We chose the Flow Parser[12] library because it is written in OCaml, which is the language we can instantiate our interpreter into, and because it is used to manipulate JavaScript in industrial setting.

When instantiating the interpreter, we define a transformation that, given a Flow AST, produces a well-typed ES AST. In the long term, we would like to write a parser faithful to the specification, but the complexity of ES syntax makes it challenging.

In Figure A2, we show the resulting AST of a program produced by the Flow Parser. Its transformation is presented in Figure A3.

An execution of our interpreter on a JS program first performs `InitializeHostDefinedRealm`[13], then parses the source code, and finally calls `ScriptEvaluation`[14].

[10] https://tc39.es/ecma262/#prod-Expression
[11] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
[12] https://github.com/facebook/flow
[13] https://tc39.es/ecma262/#sec-initializehostdefinedrealm
[14] https://tc39.es/ecma262/#sec-runtime-semantics-scriptevaluation

```
(({Loc.source = None; start = {Loc.line = 1; column = 0};
   _end = {Loc.line = 1; column = 16}},
  {Flow_ast.Program.statements =
    [({Loc.source = None; start = {Loc.line = 1; column = 0};
       _end = {Loc.line = 1; column = 10}},
      Flow_ast.Statement.VariableDeclaration
       {Flow_ast.Statement.VariableDeclaration.declarations =
         [({Loc.source = None; start = {Loc.line = 1; column = 4};
            _end = {Loc.line = 1; column = 9}},
           {Flow_ast.Statement.VariableDeclaration.Declarator.id =
             ({Loc.source = None; start = {Loc.line = 1; column = 4};
               _end = {Loc.line = 1; column = 5}},
              Flow_ast.Pattern.Identifier
               {Flow_ast.Pattern.Identifier.name =
                 ({Loc.source = None; start = {Loc.line = 1; column = 4};
                   _end = {Loc.line = 1; column = 5}},
                  {Flow_ast.Identifier.name = "x"; comments = None});
                annot =
                 Flow_ast.Type.Missing
                  {Loc.source = None; start = {Loc.line = 1; column = 5};
                   _end = {Loc.line = 1; column = 5}};
                optional = false});
            init =
             Some
              ({Loc.source = None; start = {Loc.line = 1; column = 8};
                _end = {Loc.line = 1; column = 9}},
               Flow_ast.Expression.Literal
                {Flow_ast.Literal.value = Flow_ast.Literal.Number 1.;
                 raw = "1"; comments = None})})];
        kind = Flow_ast.Statement.VariableDeclaration.Let; comments = None});
     ({Loc.source = None; start = {Loc.line = 1; column = 11};
       _end = {Loc.line = 1; column = 16}},
      Flow_ast.Statement.Expression
       {Flow_ast.Statement.Expression.expression =
         ({Loc.source = None; start = {Loc.line = 1; column = 11};
           _end = {Loc.line = 1; column = 16}},
          Flow_ast.Expression.Binary
           {Flow_ast.Expression.Binary.operator =
             Flow_ast.Expression.Binary.Plus;
            left =
             ({Loc.source = None; start = {Loc.line = 1; column = 11};
               _end = {Loc.line = 1; column = 12}},
              Flow_ast.Expression.Identifier
               ({Loc.source = None; start = {Loc.line = 1; column = 11};
                 _end = {Loc.line = 1; column = 12}},
                {Flow_ast.Identifier.name = "x"; comments = None}));
            right =
             ({Loc.source = None; start = {Loc.line = 1; column = 15};
               _end = {Loc.line = 1; column = 16}},
              Flow_ast.Expression.Literal
               {Flow_ast.Literal.value = Flow_ast.Literal.Number 1.;
                raw = "1"; comments = None});
            comments = None});
        directive = None; comments = None})];
   comments = None; all_comments = []}),
 [])
```

Figure A2: Parsing of “`let x = 1; x + 1`”

```
Script
 (VSome (ScriptBody (STMTList
     (STMTList_Item
         (STMTListIt_Decl
           (LexicalDeclaration
(*LexicalDeclaration :
    LetOrConst BindingList
*)
             (LexDecl
               (Let,
                LexBind (LexBind_ID (BI_identifier (IdentifierName "x"),
                VSome (Init (AssE_CondE (CondE_ShortCircuit (SCE_LOE
                  (LOE_LAE (LAE_BOR (BOE_BXE (BXE_BAE (BAE_Equality
                     (EqExp_Relational (RelExp_Shift (ShExp_Additive
                       (AddE_Multiplicative (MulE_Exponentiation
                         (EE_UnaryExpression (UnaExp (UpdExp
                           (LHSE_NewExpression (NE_MemberExpression
                             (ME_PrimaryExpression (PE_Literal
                               (Lit_NumericLiteral (DecimalLiteral 1.))
                      ))))))))))))))))))))
           )))))
         ),
       STMTListIt_STMT (ExpressionStatement (ExprSTMT (Expr_AssExpr
         (AssE_CondE (CondE_ShortCircuit (SCE_LOE (LOE_LAE (LAE_BOR (BOE_BXE
           (BXE_BAE (BAE_Equality (EqExp_Relational (RelExp_Shift
             (ShExp_Additive
(*AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression
*)
              (AddE_Sum
                (AddE_Multiplicative (MulE_Exponentiation
                  (EE_UnaryExpression (UnaExp
                    (UpdExp (LHSE_NewExpression (NE_MemberExpression
                      (ME_PrimaryExpression (PE_IdentifierReference
                        (IR_identifier (IdentifierName "x")))))))))),
                 MulE_Exponentiation
                   (EE_UnaryExpression (UnaExp (UpdExp (LHSE_NewExpression
                     (NE_MemberExpression (ME_PrimaryExpression (PE_Literal
                       (Lit_NumericLiteral (DecimalLiteral 1.))
                   )))))))
              ))
             ))))))))))))))))))
```

Figure A3: The transformation of Figure A2

B Unspecified Terms

For introductive purposes to JSkel, we have left these terms unspecified, presenting only their definition. All the terms presented in this section are fully implemented in the `skel` language.

```
(*Operations of the completionRecord type*)
term getNormal<α> : completionRecord<α> → α
term getAbrupt<α> : completionRecord<α> → (maybeEmpty<value>, completionType)
term abruptionCoerce<α,β> : completionRecord<α> → completionRecord<β>

(*out type getters and setters*)
term getAnomaly<α> : out<α> → anomaly
term retAnomaly<α> : anomaly -> st<out<α>>
term retAbrupt<α> : (maybeEmpty<value>, completionType) -> st<out<α>>
term retNormal<α> : α -> st<out<α>>
```

Figure B1

```
(*Types*)

type boolean =
| False
| True

(*Binders*)

term cf_bind<α,β,γ> :
     controlFlow<α,β> →
     (α → controlFlow<γ,β>) →
     controlFlow<γ,β>

term cf_res<α,β> :
     controlFlow<α,β> →
     (α → β) →
     β

(*Returns*)

term cf_ret<α> : α → controlFlow<st<out<α>>>
term cf_cont<α> : () → controlFlow<st<out<α>>>
```

Figure B2

```
(*Binders*)

term cf_out<β,γ> :
     st<controlFlow<(),β>> →
     (β → γ) →
     st<γ>

term throw<β> :
     (() → st<out<β>>) →
     (st<controlFlow<(),out<β>>> → st<controlFlow<(),out<β>>>) →
     st<controlFlow<(),out<β>>>

term bind<α,β,γ> :
     st<controlFlow<α,out<β>>> →
     (α → st<controlFlow<γ, out<β>>>) →
     st<controlFlow<γ, out<β>>>

(*Returns*)

term cont<β> : () → st<controlFlow<(), out<β>>>
term ret<β> : β → st<controlFlow<(), out<β>>>

(*Functions*)

term fooError<α> : () → st<out<α>>
```

Figure B3

```
(*Types*)

type valref = | Reference reference | Value value
type type_valref = | T_Ref | T_Val
type ref_bv = | R_Value value | R_EnvironmentRecord loc_EnvironmentRecord
type type_ref_bv = | T_R_EnvRec | T_R_Value
type value = | Null | Undefined | String string | Number number
             | BigInt bigInt | Object loc_Object | Symbol value
             | Boolean boolean
type type_value = | T_Null | T_Undefined | T_String | T_Number
                  | T_BigInt | T_Object | T_Symbol | T_Boolean

(*Binders*)

term returnIfAbrupt<α,β,γ> :
     out<α> → (α →
     st<controlFlow<γ, out<β>>>) →
     st<controlFlow<γ, out<β>>>

term assF<α,β> :
     boolean →
     (boolean → st<controlFlow<α,out<β>>>) →
     st<controlFlow<α,out<β>>>

term assT<α,β> :
     boolean →
     (boolean → st<controlFlow<α,out<β>>>) →
     st<controlFlow<α,out<β>>>

(*Returns*)

term assign<α,β> : α → st<controlFlow<α,out<β>>>

(*Boolean type checkers*)

term valref_Type : (valref, type_valref) → boolean
term val_Type : (value, type_value) → boolean
term ref_bv_Type : (ref_bv, type_ref_bv) → boolean

(*Reference type abstract methods*)

term hasPrimitiveBase : reference → boolean
term getBase : reference → ref_bv
term isUnresolvableReference : reference → boolean
term isPropertyReference : reference → boolean
term getReferencedName : reference → string
term isStrictReference : reference → boolean

(*Type conversion operations*)

term toObject : value -> loc_Object
(*Object internal method*)
term o_Get : (loc_Object, string, value) → st<out<value>>

(*Environment Record's abstract method*)

term er_GetBindingObject : (loc_EnvironmentRecord, string, boolean) → st<out<value>>

(*Errors*)

term referenceError<α> : () → st<out<α>>
```

Figure B4

```
term getValue : (v : out<valref>) -> st<out<value>> =
   let%cf_out result =
     let%returnIfAbrupt v = v in
     branch valref_Type(v, T_Ref);%ifpFalse let Value v = v in ret<value> v end;%bind
     let Reference v = v in let base = getBase(v) in
     branch isUnresolvableReference v;%ifpTrue
            let%throw re = referenceError<value> in re end;%bind
     branch let True = isPropertyReference v in
            let%bind base =
               branch let True = hasPrimitiveBase v in let R_Value base = base in
                      val_Type(base,T_Null);%assF val_Type(base,T_Undefined);%assF
                      let%b base = toObject(base) in assign<loc_Object,value>(base)
               or     let False = hasPrimitiveBase v in
                      let R_Value (Obj base) = base in assign<loc_Object,value>(base)
               end in
            let name = getReferencedName(v) in let%q thisVal = getThisValue(v) in
            let%q v' = o_Get(base,name,thisVal) in ret<value> v'
     or     let False = isPropertyReference v in ref_bv_Type(base,T_R_EnvRec);%assT
            let R_EnvironmentRecord base = base in
            let name = getReferencedName(v) in let strict = isStrictReference(v) in
            let%q v' = er_GetBindingValue(base,name,strict) in ret<value> v'
     end
   in
   result
```

Figure B5