



BY SUKYOUNG RYU AND JIHYEOK PARK


JavaScript Language Design and Implementation in Tandem

Key Insights

- JavaScript is the first programming language for which each change to its prose language specification is both “type checked” and “tested” to identify bugs and inconsistencies.
- The primary enabler is the automatic extraction of a “mechanized specification” from a language specification written in prose, which allows the generation of a reference implementation of the language from the specification.
- In addition to reference implementations, mechanized specifications can be used to detect conformance bugs between language specifications and existing JavaScript engines in major Web browsers, and generate more special-purpose JavaScript implementations, such as static analyzers, in a correct-by-construction manner.
- A promising approach to programming language development is to first design the language in a mechanized specification and then generate both human-friendly specifications written in a variety of natural languages and correct-by-construction implementations from the mechanized specification.

Programming languages have been specified using a wide variety of approaches. Most programming language specifications are written in unstructured prose, but some are written rigorously to help developers build correct language implementations. For example, Standard ML (SML) was first designed with a formal specification that defined the language syntax and semantics in mathematical notation, followed by a reference implementation of the specification. JavaScript is well known for its language specification, which is written in highly structured prose at the level of pseudocode algorithms. Finally, the specification of WebAssembly provides the syntax and semantics of the language in both highly structured prose and mathematical notation.

Unfortunately, rigorous language specifications do not prevent bugs in language implementations. SML maintains a list of reported bugs, and different implementations have different sets of bugs. JavaScript has many implementation bugs in the JavaScript engines of various Web browsers.

 Rigorous language specifications do not prevent bugs in language implementations.

More importantly, it is difficult to get a rigorous language specification right. Despite its complete formal semantics, the WebAssembly 1.0 specification had bugs detected by mechanized proofs of Conrad Watt.³⁵ The developers of the Verse programming language² described the language semantics in rewriting rules and opened a call for participation to the PL community for confluence proofs.

In this article, we present how to automatically extract a mechanized specification from a prose specification and how useful it can be in practice. Using the example of JavaScript, we show how mechanized specifications can be used to (1) detect conformance bugs between language specifications and existing JavaScript engines in major Web browsers, and (2) generate more special-purpose JavaScript implementations, such as static analyzers, in a correct-by-construction manner. We propose a new approach to programming language development as a promising direction for the future: first design the language in a mechanized specification and then generate both human-friendly specifications written in diverse natural languages and correct-by-construction implementations and tools from the mechanized specification.

We propose a new approach to programming language development: design the language in a mechanized specification, then generate both human-friendly specifications written in diverse natural languages and correct-by-construction implementations and tools from the mechanized specification.

History of JavaScript

JavaScript is the most actively used programming language on GitHub.¹⁷ All Web browsers include a JavaScript engine. It was initially designed and implemented by Brendan Eich in May 1995 as a simple dynamic language that allowed code snippets to be interpreted by Web browsers. In early 1996, companies including Netscape and Microsoft were frequently releasing browser technology, but language standardization was slow and often contentious. To ensure interoperability between different browsers, TC39, the Ecma Technical Committee responsible for standardizing JavaScript, had meetings to create the JavaScript language specification.

Unlike programming languages that “grow up” via a single implementation, JavaScript began with multiple implementations, which guided its specification:³⁶

Richard Gabriel, who attended some of the working group meetings, recalled in a personal communication a not uncommon interaction during these meetings. Guy Steele would ask a question about some edge-case feature behavior. Sometimes Brendan Eich would say “I don’t know,” and sometimes Eich and Shon Katzenberger would be unsure or disagree; in such cases, they would each turn to their respective implementation and try a test case. If they got the same answer, that became the specified behavior. If there were a difference, they would discuss the issue until they reached an agreement.

The history of JavaScript is described in great detail in Wirfs-Brock and Eich.³⁶ The first edition of its language specification ECMA-262, abbreviated ES1, was released in 1997, edited by Guy L. Steele, Jr, in 95 pages. JavaScript developers continued to demand more advanced language features, so ES2 and ES3 were released in 1998 and 1999, respectively. However, attempts to define a fourth edition were eventually abandoned due to the radical changes in a single update that included a variety of new language features, and ES5 was finally released in 2009. Starting with the sixth edition, TC39 adopted the practice of using the year of publication as an abbreviation. Thus, both “ES6” and “ES2015” are informal abbreviations for “ECMA-262, 6th edition.” TC39 also decided to release ECMA-262 annually, starting with ES2015 to ensure rapid adoption of new language features. The latest ECMA-262¹⁰ is a much larger specification at 827 pages.

Now, ECMA-262 is maintained as an open source project¹¹ and follows the TC39 process¹⁴ for handling proposals for new language features. JavaScript contributors propose new features along with specification changes and tests, which are maintained in a separate repository⁶ over six stages. Since 2015, TC39 has successfully published an updated edition of the ECMAScript specification every June, following the TC39 process.

As with the language specification, various companies, including Microsoft and Google, have released their own open source test suites for JavaScript. In 2010, TC39 decided to maintain Test262,¹⁵ an open source JavaScript implementation conformance test suite. After working through many policy and licensing issues, Test262 is now an integral part of TC39's development process. Every new ECMAScript feature must be accompanied by its tests before it is incorporated into the ECMAScript standard. At the time of writing, Test262 consists of 48,854 tests.

Correctness and Conformance of the Specification and Implementations

Along with its reputation as the most widely used language, JavaScript is also well-known for its unintuitive semantics due to its highly dynamic nature and extensive use of implicit type conversion. As a result, there are many sophisticated JavaScript examples. Consider the following JavaScript code:

```
function f(x) { return x == !x; }
```

Even for this simple function, it is not easy to understand exactly what its behavior is: the function `f` simply compares the given argument `x` with its negation, so it looks like it returns `false`. However, when an empty array is given as an argument, it returns `true` due to a number of implicit conversions for the negation and equality operators. More specifically, when `f([])` evaluates `[] == ![]`, the negation of the empty array `![]` evaluates to `false` because any object represents `true`. The operands `[]` and `false` of the equality operator are then both converted to values of the same type according to the implicit conversion rules defined in ECMA-262. In this example, they both get converted into the same `Number` type value, `0`, so the final result becomes `true`.

Such counterintuitive semantics often leads to various bugs and security vulnerabilities in implementations. Experienced JavaScript developers often introduce bugs that are difficult to catch due to the extremely dynamic nature of JavaScript. Mainstream JavaScript engines like V8, JavaScriptCore, SpiderMonkey, and Chakra had various bugs that were more harmful than bugs in JavaScript programs.³⁴ They also had security vulnerabilities that could lead to remote attacks. For example, a high-severity bug in V8, tracked as CVE-2021-21224, was widely exploited in April 2021.³³ Besides, it is more challenging to correctly develop special-purpose JavaScript implementations that require a deeper understanding of the specification for specialized language semantics. For example, most existing JavaScript static analyzers^{19,20,23} require a sound abstraction of the language semantics to guarantee the soundness of their analysis. However, because they need to consider not only concrete semantics but also how to abstract them soundly, they have been plagued by soundness bugs²⁵ for unusual edge cases in language semantics.

ECMA-262 also had a number of bugs. Consider the following **Math.round** built-in library function (specified in Section 20.3.2.28 of an ECMA-262 internal version):⁷

20.3.2.28 Math.round (*x*)

1. Let *n* be ? `ToNumber` (*x*).
2. If *n* is an integral Number, return *n*.
3. If *x* < 0.5 and *x* > 0, return + 0.
4. If *x* < 0 and *x* ≥ - 0.5, return - 0.

5. Return the integral Number closest to n , preferring the Number closer to $+\infty$ in the case of a tie.

It first converts the given parameter x to its numeric value n using `ToNumber`. The remaining steps should be performed using n , but the specification writer of this section mistakenly used x instead of n in steps 3 and 4. This bug was introduced in ECMA-262 on September 11, 2020 and was later fixed by another contributor.

In addition, keeping a rapidly evolving language specification up to date and managing the many different language implementations that conform to the specification is challenging even with a large test suite. The three editors of ECMA-262 had to manually review new proposals and changes to the specification. In addition to `Test262`, various browsers maintain their own test suites, but they may still behave differently. Therefore, Ficarra,¹⁶ an editor of ECMA-262, said, “one of my primary goals has been to make the specification easier to consume for automated analysis tools.”

Academic Research into the CI Systems

We helped Ficarra achieve his goal in November 2022: each ECMA-262 pull request (PR) runs a type checker against the prose specification, and all new or changed tests in `Test262` PRs are run using an interpreter extracted directly from the text of ECMA-262. For example, if one sends a PR of the `Math.round` function, the type checker will detect a bug and reject the PR. First, note that the parameter x can accept any JavaScript value: string, boolean, number, object, and so on. Applying `ToNumber` to x in step 1 converts x to a number or an exception. Exception cases are filtered out using the question mark operator, so n always points to a number. Because x is compared to several numbers with inequality operators on lines 3 and 4, the type checker reports them as type mismatch bugs because non-numeric values are not valid arguments for inequality operators. Whenever a language feature is added to ECMA-262, it must be accompanied by its corresponding tests in `Test262`, which now leverage interpreters extracted from ECMA-262, always checking for conformance to ECMA-262. These automated tools, heavily used in the continuous integration (CI) system of ECMA-262 and `Test262`, are based on a series of academic papers.

How have ideas from academia been integrated into real-world industry? How did researchers convince the TC39 committee to use their ideas?

The KAIST Programming Language Research Group (PLRG) has been researching JavaScript since 2011. Initially, we mainly formalized the semantics of the JavaScript language with various features, but now our research focuses on program analysis and bug finding in JavaScript applications. Our research problems are often motivated by real-world customers in companies such as Samsung Electronics and IBM. This work had been challenging, interesting, rewarding, and fun until TC39 decided in 2015 to release ECMA-262 annually. As the JavaScript language has evolved more rapidly, developing and maintaining JavaScript analysis tools has become increasingly difficult.

As the JavaScript language has evolved, developing and maintaining JavaScript analysis tools has become increasingly difficult.

In March 2019, Ph.D. candidate Jihyeok Park cautiously shared an outlandish idea. ECMA-262 had been released annually since 2015, but existing JavaScript analyzers, including our own, were still based on ES5, which was released in 2009. It is impossible to manually keep up with the changes in an 800-page specification every year. Then he realized something: the English phrases in the specification had common patterns. It might be possible to “parse” the English sentences and “compile” them into abstract algorithms in an intermediate language. We considered this a clever engineering hack, which could help us generate more tests for features of ECMA-262 that `Test262` does not cover.

In essence, it was the primary enabler; the direct extraction of “mechanized specifications” from prose-written language specifications has opened the door to the automatic generation of language-manipulating tools. To bridge the gap between ECMA-262 and its implementations, ESMeta²¹ extracts mechanized specifications to automatically generate a variety of language-based tools from a given version of ECMA-262. It is based on several papers. JISET²⁸ extracts a mechanized specification from ECMA-262. A mechanized specification consists of two parts: a JavaScript parser constructed from the syntax written in a variant of the extended BNF (EBNF) notation, and functions in an intermediate representation (IR) compiled from abstract algorithms written in English for the language semantics. JEST²⁷ synthesizes conformance test programs and checks discrepancies between JavaScript engines and the specification. Using this tool, we detected 44 bugs in four engines (V8, GraalJS, QuickJS, and Moddable XS) and 27 bugs in ES2020. JSTAR²⁶ analyzed the types of English sentences in ECMA-262 and detected 93 type-related specification bugs, which were confirmed by TC39. JS-AVER²⁵ automatically generates a JavaScript static analyzer from ECMA-262, which outperforms the state-of-the-art JavaScript static analyzers that were manually developed. The next section offers a description of the technical details behind them.

The direct extraction of “mechanized specifications” from prose-written language specifications has opened the door to the automatic generation of language-manipulating tools.

Because the papers presented various new techniques using mechanized specifications, we used their bug-finding capabilities to evaluate the effectiveness of the techniques. Thus, we submitted many bug reports to mainstream JavaScript engine developers and the TC39 committee for confirmation. They kindly confirmed the bugs and expressed a lot of curiosity. Then, the ECMA-262 editors invited us to a TC39 meeting.

The presentation was very well received. All the excitement from the TC39 committee and the detailed discussion can be found in the meeting note.¹³ After mutually exciting meetings with the TC39 committee, we decided to integrate JSTAR and JISET into the CI systems of ECMA-262 and Test262, respectively. Since these tools were prototype implementations to see their feasibility in academic publications, we reimplemented all the tools and rebranded them as ESMeta to make them practically available to all PRs in the ECMA-262 and Test262 repositories.

After the first meeting with ECMA-262 editors on Nov. 24, 2021, we gave a presentation at the TC39 meeting on Jan. 27, 2022. ESMeta was then integrated into ECMA-262’s CI system on Nov. 3, 2022¹² and Test262’s CI system on Nov. 25, 2022.⁹ This was about a year after the first meeting with the TC39 committee. This is how the initial outlandish idea and subsequent academic papers were integrated into real-world programming language development.

Technical Details

Researchers have proposed various approaches to help developers build correct JavaScript applications.^{1,32}

One approach is to formalize the JavaScript language semantics described in ECMA-262. Because ECMA-262 defines semantics in prose, it is sometimes ambiguous and contains bugs and infeasible behavior. Researchers have proposed formal specifications for JavaScript semantics to provide a solid foundation for JavaScript research. Maffei et al.²² proposed a small-step operational semantics for ES3; Guha et al.¹⁸ used a desugaring process to develop λ_{JS} , a core calculus of ES3; and Park et al.²⁴ defined ES5 using the K framework.³⁰

Another approach is to analyze JavaScript programs to reason about their behavior or detect bugs and security vulnerabilities. WALA¹⁹ was initially developed for Java pointer analysis and has been extended to support more languages, including Android Java and JavaScript. TAJs²³ is a dataflow analysis for JavaScript that uses a model of ES3 and a partial model of ES5. It provides partial support for the latest ECMAScript language features with Babel,³ which compiles the latest features down to lower versions. SAFE²⁰ is a general analysis framework for JavaScript web applications. These are all

open source projects for static analysis of JavaScript. In contrast, Jalangi³¹ is a general framework for JavaScript dynamic analyzers such as memory profilers and dynamic JIT-unfriendly code snippet detectors.

While most of the research on JavaScript is for ES3 and ES5, ECMA-262 has been released every year since 2015. Thus, manually updating the semantic formalizations and analysis implementations is tedious, labor-intensive, and error-prone.

To bridge the gap between the rapidly evolving ECMA-262 and its implementations, ESMeta generates various tools directly from ECMA-262. Figure 1 illustrates the overall structure of ESMeta. The first step is to extract a mechanized specification from an input ECMA-262 via JISET. Once a mechanized specification is available, it can be used to check the validity of ECMA-262. We can use JEST to synthesize new kinds of conformance tests and JSTAR to analyze the types of English phrases in the specification. Finally, we can use JSAVER to derive a static analyzer for a given version of ECMA-262. We will describe them in order.

Extraction of Mechanized Specifications

ECMA-262 defines the language syntax using a variant of EBNF and the semantics using abstract algorithms in a clear and structured manner. For example, the following production shows the syntax of *ArrayLiteral* in ES2022:

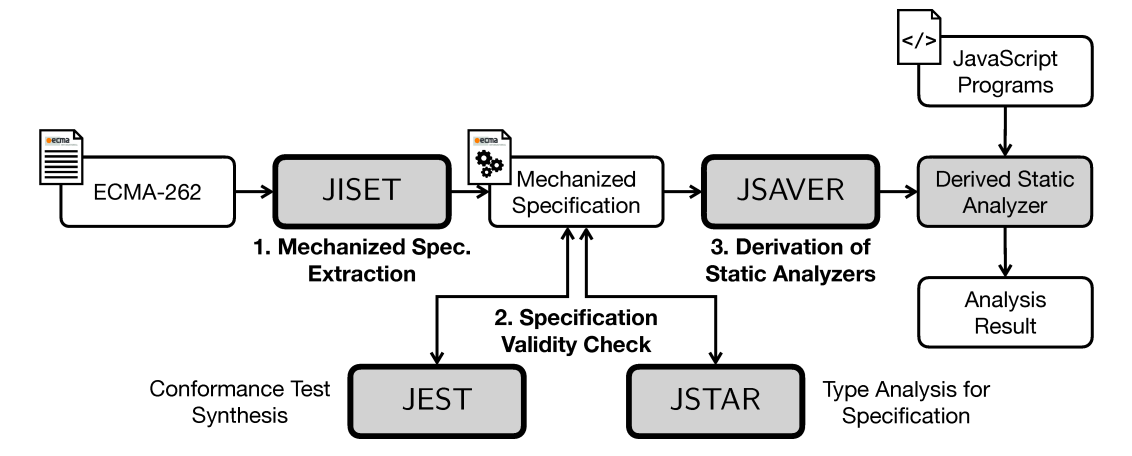
```
ArrayLiteral[Yield, Await] :
  [ Elisionopt ]
  [ ElementList[?Yield, ?Await] ]
  [ ElementList[?Yield, ?Await] , Elisionopt ]
```

It takes two boolean parameters *Yield* and *Await*, and has three alternatives. The following abstract algorithm defines the semantics of the third alternative:

```
ArrayLiteral : [ ElementList , Elisionopt ]
```

1. Let *array* be ! *ArrayCreate*(0).
2. Let *nextIndex* be ? *ArrayAccumulation* of *ElementList* with arguments *array* and 0.
3. If *Elision* is present then:
 - Perform ? *ArrayAccumulation* of *Elision* with arguments *array* and *nextIndex*.
4. Return *array*.

Figure 1. Overall structure of ESMeta.



It has four steps. In the HTML files describing ECMA-262, each nonterminal, such as *ElementList*, or local variable, such as *array*, has a `<nt>` or `<var>` tag, respectively. From the above production, the *lookahead parsing* technique²⁸ generates a parser in Scala code as follows:

```
val ArrayLiteral: List[Boolean] => LParser[T] = memo {
  case List(Yield, Await) =>
    "[" ~ opt(Elision) ~ "]"      ^^ ArrayLiteral0 |
    "[" ~ ElementList(Yield,Await)
      ~ "]"                      ^^ ArrayLiteral1 |
    "[" ~ ElementList(Yield,Await)
      ~ "," ~ opt(Elision) ~ "]" ^^ ArrayLiteral2
}
```

Each parser has the `List[Boolean] => LParser[T]` type because each production is parametric with boolean values. Similarly, the *algorithm compiler*²⁸ translates the above abstract algorithm to the following function in a domain-specific intermediate representation, IR_{ES}:

```
syntax def ArrayLiteral[2].Evaluation(
  this, ElementList, Elision
) {
  let array = [! (ArrayCreate 0)]
  let nextIndex =
    [? (ElementList.ArrayAccumulation array 0)]
  if (! (= Elision absent))
    [? (Elision.ArrayAccumulation array nextIndex)]
  return array
}
```

We evaluated the correctness of the semantics extracted from ES2019⁵ by running Test262. It took about three hours to evaluate 18,064 applicable tests; 1,709 tests failed due to nine specification bugs in ES2019. Four of these bugs were newly reported and confirmed by TC39.

Synthesis of Conformance Tests

In addition to the annual updates to ECMA-262, the various JavaScript engines continue to provide various extensions to the specification to meet rapidly changing user needs. Unfortunately, these updates, both in the specification and in implementations, make synchronization difficult, leading to unexpected behavior.

Inspired by the ECMA-262 bugs detected by the extracted semantics, we devised an *N + 1-version differential testing*.²⁷ Traditional differential testing runs *N* implementations of a specification simultaneously for each input and detects problems when the outputs do not match. *N + 1-version differential testing* also tests the specification using a mechanized specification extracted from the specification.

It consists of four steps:

1. Automatically synthesize programs according to the syntax and semantics from a given language specification.
2. generate conformance tests by injecting assertions into the synthesized programs to check their final program states.

3. Run the conformance tests against multiple implementations to detect bugs in the specification and implementations
4. Use statistical information to localize bugs in the specification.

We evaluated the effectiveness of the synthesized tests with ES2020 and four JavaScript engines that fully support modern JavaScript features in ES2020: V8, GraalJS, QuickJS, and Moddable XS. For evaluation, we injected seven kinds of assertions: exception, abort, variable value, object value, object property, property key, and internal method and slot. For example, to check whether a final program state has the correct value for each object property, we implemented a helper `$verifyProperty`, which checks the attributes of each property for each object. Thus, the following code checks the attributes of the property of `x.p`:

```
var x = { p: 42 };
$verifyProperty(x, "p", {
  value: 42.0,      writable: true,
  enumerable: true, configurable: true
});
```

The bug detection and localization phase then uses the results of running given conformance tests on multiple JavaScript engines. If a small number of engines fail in each test, it reports a potential bug in the engines that failed the test. If a large number of engines fail, it reports a potential bug in the specification. It uses spectrum-based fault localization (SBFL),³⁷ a localization technique that leverages the coverage of test cases and pass/fail results, to localize potential bugs. We detected 44 bugs in the engines and 27 bugs in ES2020. One of the ES2020 bugs was a newly detected bug confirmed by TC39, caused by not handling abrupt completions in property definitions of object literals.

Type Analysis of Specifications

Manually reviewing every specification update is inherently labor-intensive and error-prone, making ECMAScript vulnerable to specification bugs. Because the average number of updated steps of abstract algorithms between consecutive releases from ECMAScript 2016 to 2019 was 9,645.5,²⁸ manually checking for every update is a daunting task. Thus, TC39 pushed to add various manual annotations to the abstract algorithms to reduce specification bugs. First, it introduced two kinds of annotations: *assertions*, which indicate assumptions at specific points in abstract algorithms, and the *prefixes* `?` and `!`, which indicate whether the execution of an abstract algorithm completes abruptly. For example, “Assert: Type(*O*) is Object” denotes that the variable *O* always has an Object value at the point of the assertion, and “? **GetV**(*V*, *P*)” denotes that the execution of **GetV**(*V*, *P*) can complete abruptly. These annotations help readers understand specifications clearly. Second, the committee decided to support type annotations for variables, parameters, and return values of abstract algorithms. However, manual annotations of any kind are laborious, prone to mistakes, and do not provide an automatic mechanism for detecting specification bugs.

Manual annotations of any kind are laborious, prone to mistakes, and do not provide an automatic mechanism for detecting specification bugs.

To alleviate this problem, we developed JSTAR,²⁶ which takes a mechanized JavaScript specification from JISET and performs type analysis of compiled functions using the specification types defined in ECMA-262. ECMA-262 contains not only JavaScript language types, but also specification types such as abstract syntax trees (ASTs), internal list-like structures, and internal

records including environments, completions, and property descriptors. For records and AST types, we also defined their fields. We defined their type hierarchies based on *subtype relations*. The subtype relation between types is shown in Figure 2; a directed edge from τ' to τ denotes a subtype relation (that is, $\tau' \prec \tau$), and the relation is reflexive and transitive. The subtype relation depends on the nominal types defined in ECMAScript. We extract the subtype relation for AST types from the JavaScript syntax. For example, consider the following syntax:

FormalParameter[?Yield, ?Await]: ***BindingElement***[?Yield, ?Await]

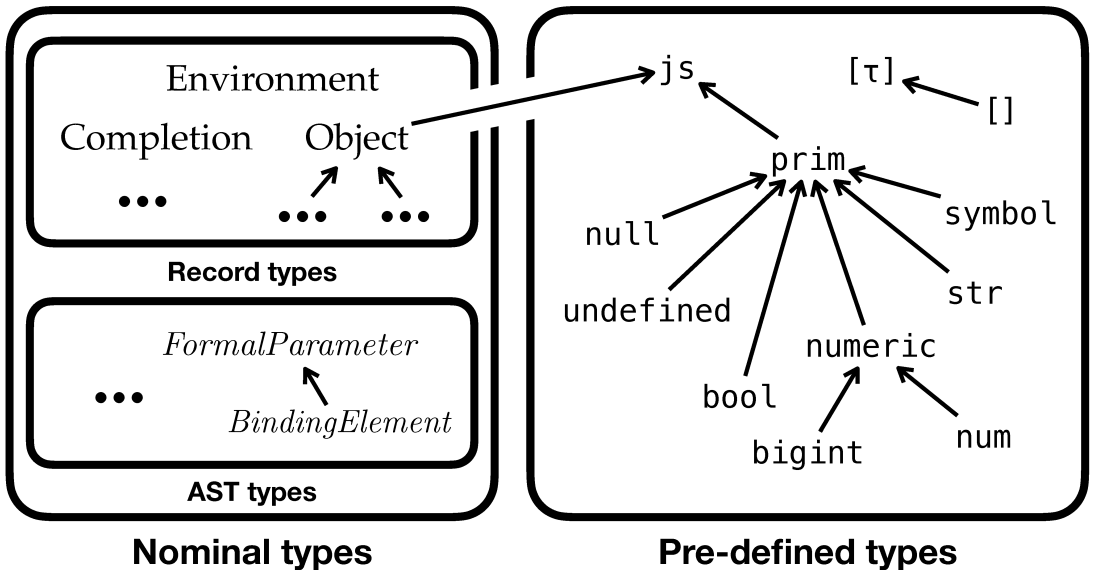
Because the nonterminal *BindingElement* is the unique alternative of the production *FormalParameter*, we automatically extract the subtype relation: *BindingElement* \prec : *FormalParameter*. Using the subtype relation, the expression $e : \tau$ checks whether the evaluation result of e has type τ' satisfying $\tau' \prec \tau$. These subtype relations help enhance the precision of type analysis by keeping track of the precise types of variables and expressions.

Using such type information, JSTAR performs type analysis and detects specification bugs using a bug detector consisting of four checkers: 1) reference checker, 2) arity checker, 3) assertion checker, and 4) operand checker. JSTAR also uses condition-based refinement for type analysis, which improves the precision of type analysis by using conditions on assertions and branches to eliminate infeasible parts. We evaluated JSTAR with all 864 versions in the official ECMAScript repository from 2018 to 2021. The evaluation showed that the refinement technique can reduce the number of false-positive bugs due to spurious types inferred by imprecise type analysis. JSTAR detected 14 type-related bugs in ES2021,⁸ which were confirmed by TC39.

Derivation of Static Analyzers

Finally, we developed JSAYER,²⁵ which automatically generates a JavaScript static analyzer from ECMA-262. First, JSAYER extracts definitional interpreters²⁹ from ECMA-262. A definitional interpreter provides a way to represent the language semantics of a defined language using its interpreter written in a defining language. We extract a JavaScript definitional interpreter from JISET. In the extracted definitional interpreter, the defined language is JavaScript, and the defining language is IR_{ES}. We then present meta-level static analysis, which uses the extracted interpreter to indirectly analyze JavaScript programs. Meta-level static analysis is an interpreter-based approach for static analysis of a defined language L_1 using the static analyzer of a *defining-language* L_2 , as depicted in Figure 3. Since an L_1 interpreter is an L_2 program, we can indirectly analyze an L_1 program by taking the L_1 program as input and using the static analyzer of L_2 to analyze the interpreter. Thus,

Figure 2. Subtype relation \prec :



we developed a static analyzer of IR_{ES} for a meta-level static analysis of JavaScript and showed that it can indirectly analyze JavaScript programs effectively. We also presented ways to indirectly configure abstract domains and analysis sensitivities for JavaScript in the static analysis of IR_{ES} . First, we provide a method to configure abstract domains for JavaScript values and structures. Second, we present AST sensitivities to express analysis sensitivities for JavaScript, such as flow-sensitivity and k -callsite-sensitivity.

Figure 4 shows the analysis results of existing static analyzers (TAJS and SAFE) without and with Babel, and JSA_{ES2021} , the JavaScript static analyzer derived from ES2021 via JS-AVER, for the applicable tests. In each chart, the x-axis represents the point in time when the tests were generated and the y-axis represents the number of tests generated before that point in time. The mark sound (green, filled) denotes a sound analysis, unsound (red, striped) denotes an unsound analysis, and error (white, blank) denotes an unexpected error. Figures 4(a) and 4(b) show that TAJS and SAFE analyzed most tests generated before 2015 in a sound way. However, the number of tests that cannot be soundly analyzed has been steadily increasing since 2015. As shown in Figures 4(d) and 4(e), Babel transpiles ES2015+ features to ES5.1 to mitigate this issue and increase the number of programs that TAJS and SAFE analyze soundly. However, TAJS and SAFE still failed to soundly analyze more than half of the Test262 test programs, while JSA_{ES2021} succeeded in soundly analyzing all applicable test programs without the need for Babel. The figures show that JS-AVER can reduce the burden of defining the abstract semantics of ES2015+ features for static analysis.

Figure 3. Interpreter-based static analysis approach.

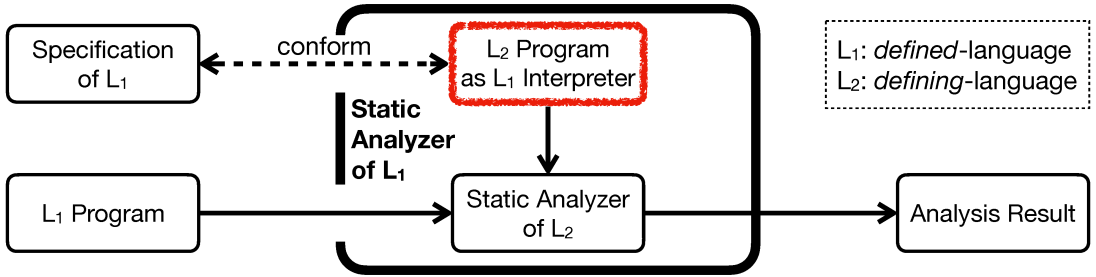
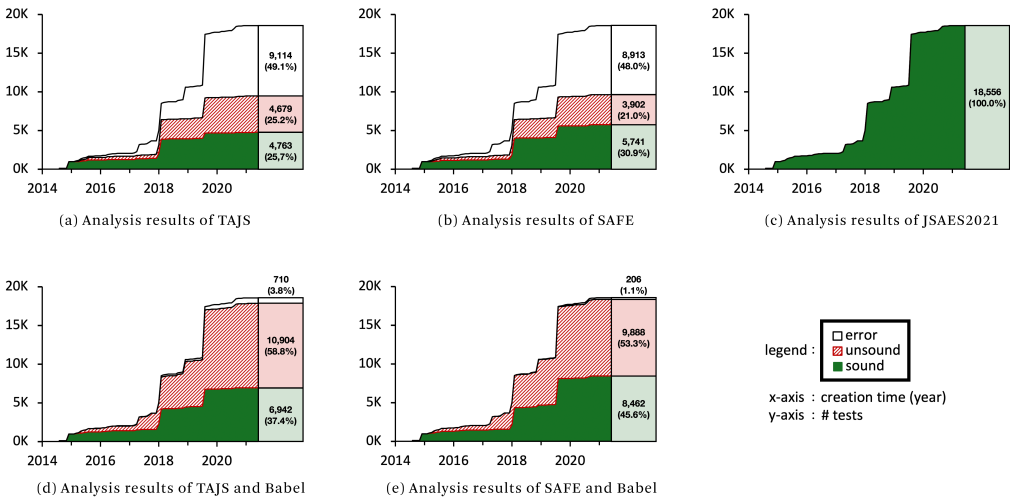


Figure 4. Analysis results of TAJS and SAFE without and with Babel and JSA_{ES2021} for applicable tests



A Promising New Approach to Programming Language Development

Designing and implementing real-world programming languages is challenging. The ability to reason about program behavior often comes from a formal specification of the language's semantics, but the time-consuming effort of formalizing the semantics often falls behind actual implementation. For example, Rust is actively developed by a large and diverse community of contributors and is used in real-world software such as the Linux kernel and Mozilla Firefox. However, it has not resolved soundness bugs reported years ago⁴ because its strong, static type system does not yet cover various language features and APIs. Applying the ESMeta approach to Rust can help efficiently generate machine-checkable proof sketches, especially with mechanized semantics extracted from mechanized specifications.

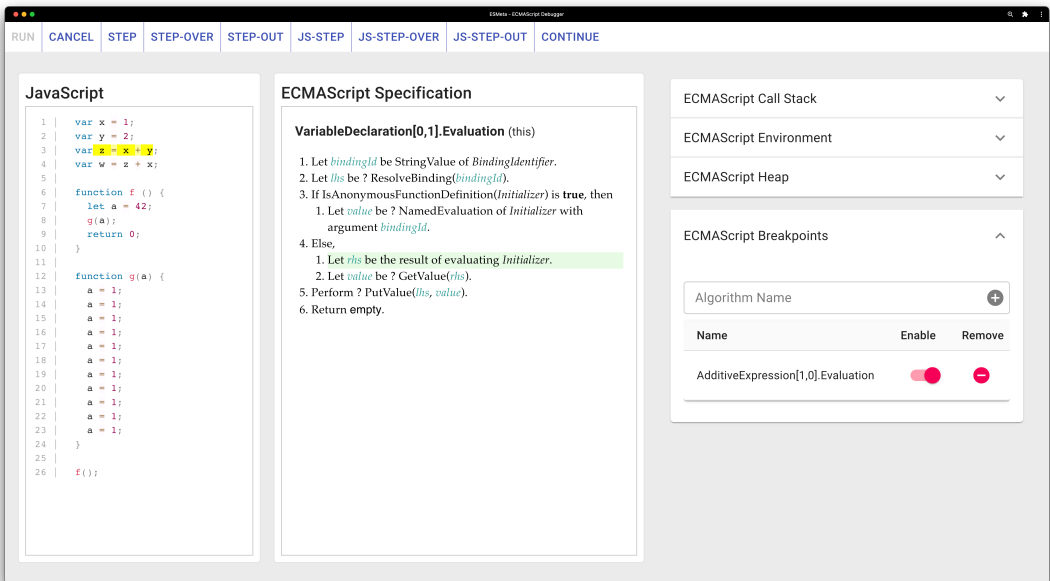
Along with formalizing the semantics of the language, it would be helpful to perform extensive testing of the semantics using implementations extracted from the mechanized specification. Watt et al.³⁵ presented two mechanizations of WebAssembly 1.0 and found bugs in it, but mechanization of WebAssembly 2.0 will still be quite time-consuming because the entire mechanization process is done manually. Applying the ESMeta approach to WebAssembly can reduce the burden of such manual mechanization.

A promising new approach to programming language development is to design languages with mechanized specifications from the beginning. For developers, mechanized specifications can be easier to understand than specifications in natural language because they are unambiguous and always executable. For non-developers, mechanized specifications can be translated into diverse, human-friendly natural languages. Furthermore, implementations and tools that are extracted directly from mechanized specifications are correct by construction.

A promising new approach to programming language development is to design languages with mechanized specifications from the beginning.

Designing a new programming language by writing a mechanized specification that correctly describes the language's intended behavior can seem daunting, but it is possible because mechanized specifications allow us to create a variety of tools. For example, one can run the

Figure 5. ECMAScript double debugger.



specification interactively. Figure 5 shows another ESMeta tool, the ECMAScript *Double Debugger*.²¹ This tool extends the interpreter extracted from ECMA-262 to help users understand how JavaScript programs are executed according to ECMA-262. It supports step-by-step execution of ECMA-262 abstract algorithms, line-by-line execution of JavaScript code, breakpoints by abstract algorithm name in ECMA-262, and visualization of ECMA-262 internal states. Language designers can use the debugger to run example code to debug their mechanized specifications. For instance, Verse introduced new features such as logical variables, equality constraints between variables, and choice that allows multiple alternatives. Describing the intended behavior precisely is cumbersome, but a double debugger can ease the burden on language designers.

Conclusion

JavaScript is the first programming language for which each change to its prose language specification is both “type checked” and also “tested” to identify bugs and inconsistencies. In this article, we presented our story of applying various ideas from academic papers to the continuous design and implementation process of the real-world programming language in the wild. As one of the reviewers of the JISET paper suggested, we believe that:

This is the right order to design and document languages: first the semantics, then the implementation and documentation, ideally generated from the semantics.

Acknowledgments

We would like to thank all members of the KAIST Programming Language Research Group (PLRG) for their collaboration, especially Jaemin Hong for his insightful feedback. This research was supported by National Research Foundation of Korea (NRF) (2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460 and 2023-2020-0-01819), and Samsung Electronics Co., Ltd (G01210570).

References

1. Andreasen, E. et al. A survey of dynamic analysis and test generation for JavaScript. *Comput. Surveys* 50, 5 (2017), 66:1–66:36.
2. Augustsson, L., Breitner, J. et al. The Verse calculus: A core calculus for deterministic functional logic programming. In *Proceedings of ACM Program. Lang.* 7, ICFP, Article 203 (Aug. 2023), 31; 10.1145/3607845
3. Babel Team. *Babel is a Javascript compiler*. Babel Community, 2022; <https://babeljs.io/>
4. Ben-Yehuda, A. *Coherence Can Be Bypassed by an Indirect Impl for a Trait Object*(2019); <https://bit.ly/4bqvoAB>.
5. Ecma International. *ECMA-262, 10th Edition, ECMAScript®2019 Language Specification* (June 2019); <https://bit.ly/488c9J9>
6. Ecma International. *Github Repository for ECMAScript Proposals* (2019); <https://bit.ly/48az6M1>
7. Ecma International. *Github Repository for an Internal Version of ECMA-262* (2020); <https://bit.ly/3Uzf8Y1>.
8. Ecma International. *ECMA-262, 12th Edition, ECMAScript®2021 Language Specification*; <https://bit.ly/3OzoHCo>
9. Ecma International. *CI: Integrate ESMeta #3730* (2022); <https://bit.ly/3HQcLJ1>.
10. Ecma International. *ECMA-262, 14th Edition, ECMAScript®2023 Language Specification*; <https://bit.ly/3ODWNVX>
11. Ecma International. *ECMAScript Repository* (2022); <https://bit.ly/49sM88x>
12. Ecma International. *Meta: Integrate ESMeta Type Checker into CI #2926* (2022); <https://bit.ly/3Us6smm>.
13. Ecma International. *Tc39: 26 January 2022 Meeting Notes*; <https://bit.ly/489mGnu>.
14. Ecma International. *The Tc39 Process* (2022); <https://bit.ly/42BnEYK>
15. Ecma International. *Test262: ECMAScript Test Suite* (2022); <https://bit.ly/3w9BQfj>
16. Ficarra, M. *Personal Communication* (2021).
17. GitHub. *The Top Programming Languages* (2022); <https://bit.ly/3utUMF1>.
18. Guha, A., Saftoiu, C., and Krishnamurthi, S. The essence of JavaScript. In *Proceedings of the European Conf. on Object-Oriented Programming*. Springer Berlin Heidelberg (2010), 126–150.
19. IBM Research. *T.J. Watson Libraries for Analysis (WALA)*, 2006; <http://wala.sf.net>.

20. KAIST PLRG. *SAFE: Javascript Analysis Framework*, 2012; <http://safe.kaist.ac.kr>.
21. KAIST PLRG. *ESMeta* (2022); <https://bit.ly/48YjA77>.
22. Maffeis, S., Mitchell, J.C., and Taly, A. An operational semantics for JavaScript. In *Proceedings of the Asian Symp. on Programming Languages and Systems*. Springer Berlin Heidelberg (2008), 307–325.
23. Møller, A. et al. *TAJS: Type Analyzer for JavaScript* (2012); <https://bit.ly/3HRyMHb>.
24. Park, D., Stefanescu, A., and Rou, G. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation*. Association for Computing Machinery (2015), 346–356.
25. Park, J., An, S., and Ryu, S. Automatically deriving JavaScript static analyzers from specifications using meta-level static analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. Association for Computing Machinery (2022), 1022–1034.
26. Park, J. et al. JSTAR: JavaScript specification type analyzer using refinement. In *Proceedings of the 36th IEEE/ACM Intern. Conf. on Automated Software Engineering*. Association for Computing Machinery (2021), 606–616.
27. Park, J. et al. JEST: N+1-version differential testing of both JavaScript engines and specification. In *Proceedings of IEEE/ACM 43rd Intern. Conf. on Software Engineering*. IEEE, Association for Computing Machinery (2021), 13–24.
28. Park, J., Park, J., An, S., and Ryu, S. JISET: JavaScript IR-based semantics extraction toolchain. In *Proceedings of the 35th IEEE/ACM Intern. Conf. on Automated Software Engineering*. IEEE, Association for Computing Machinery, (2020), 647–658.
29. Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conf. 2*. Association for Computing Machinery (1972), 717–740.
30. Rou, G. and Herbrun, T.F. K overview and SIMPLE case study. In *Proceedings of the 2nd Intern. Workshop on the K Framework and Its Applications 304*. Elsevier (2014), 3–56.
31. Sen, K., Kalasapur, S., Brutch, T., and Gibbs, S. Jalangi: A tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In *Proceedings of the 30th ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. Association for Computing Machinery (2013), 615–618.
32. Sun, K. and Ryu, S. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Survey* 50, 4 (Aug. 2017), 34.
33. Tung, L. *Bugs in Chrome's Javascript Engine Can Lead to Powerful Exploits* (2021); <https://zd.net/3HS2lbJ>
34. Wang, Z. et al. An empirical study on bugs in JavaScript engines. *Information and Software Technology* 155 (2023), 107105.
35. Watt, C. et al. Two mechanisations of WebAssembly 1.0. In *Formal Methods*. M. Huisman, C. Păsăreanu, and N. Zhan (eds). Springer Intern. Publishing, Cham, (2021), 61–79.
36. Wirfs-Brock, A. and Eich, B. JavaScript: The first 20 years. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 77 (June 2020), 189.
37. Wong, W. et al. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

Sukyoung Ryu is a professor at KAIST, Daejeon, Republic of Korea.

Jihyeok Park is an assistant professor at Korea University, Seoul, Republic of Korea.
