MiniPoly: Automatic Extraction of Efficient JavaScript Polyfill from Language Specification

Jungwoong Kim*
Korea University
Seoul, Republic of Korea
jungwoongkim@korea.ac.kr

Youngmin Cho*
Korea University
Seoul, Republic of Korea
arcde40@korea.ac.kr

Jihyeok Park✉
Korea University
Seoul, Republic of Korea
jihyeok_park@korea.ac.kr

Abstract

A *polyfill* is a JavaScript implementation of built-in APIs for older execution environments lacking native support. Because developers rely on them to run modern features everywhere, strict conformity to the official language specification (ECMA-262) is crucial. However, manual implementation remains the industry standard; this process is labor-intensive, inherently error-prone, and frequently leads to spec-miscompliance bugs in production.

To address this, we present MiniPoly, the first compilation framework to automatically extract correct and optimized polyfills directly from the ECMA-262 specification. To ensure correct generation, MiniPoly employs a static analysis that resolves the specification's internal control-flow representations (i.e., completion records) and translates them into equivalent JavaScript constructs. Furthermore, it utilizes *rewrite rules* to apply user-defined optimization patterns, replacing algorithmic bottlenecks in the specification with highly efficient implementations. Evaluation shows that MiniPoly successfully generates polyfills for 89 built-in APIs in ECMAScript 2025 (ES2025) using 17 user-defined rewrite rules. It produces a more spec-compliant polyfill than production-standard libraries like `core-js` and `es-shims` with a competitive performance profile. Additionally, we demonstrate that MiniPoly can generate polyfills even for the next draft release of ECMA-262 and early-stage proposals, highlighting its potential to future-proof development.

CCS Concepts

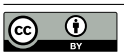
• **Software and its engineering** → **Specification languages**; **Source code generation**; *Scripting languages.*

Keywords

JavaScript, Polyfill, Language Specification, Code Generation

ACM Reference Format:
Jungwoong Kim, Youngmin Cho, and Jihyeok Park. 2026. MiniPoly: Automatic Extraction of Efficient JavaScript Polyfill from Language Specification. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3837549

*Both authors contributed equally to this research.



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837549

1 Introduction

JavaScript is a foundational pillar of the modern web ecosystem and evolves rapidly with new language features introduced annually. However, not all execution environments natively support them, creating a severe compatibility gap. For example, the built-in API `Set.prototype.isSubsetOf` was added to the official language specification in 2025, but is not yet supported by 12.13% of browsers worldwide according to the latest `caniuse` data [10].

To bridge this gap, *polyfills* implement modern built-in APIs in older execution environments and have become indispensable in web development. For instance, `core-js` is the most widely adopted polyfill library, used by 41% of websites worldwide [1] with over 46 million downloads per week [29]. Polyfill correctness is therefore paramount. Even a subtle deviation from ECMA-262 [11], the official specification of JavaScript, can silently break production code. Yet the specification is vast: its 2025 edition (ES2025) alone spans 847 pages. Libraries like `core-js` nonetheless translate it into JavaScript by hand. This manual process is labor-intensive and error-prone, frequently causing spec-compliance bugs. Worse, developers deviate from the specification to optimize performance, introducing still more bugs.

Prior Work. One potential solution is to generate polyfills automatically by mimicking native implementations. For example, Heule et al. [19] synthesize them from runtime memory-access patterns using Markov Chain Monte Carlo search over test cases. However, such behavior-driven approaches do not leverage the specification, the ultimate source of truth, so they may miss untested edge cases and cannot guarantee spec compliance. We therefore generate polyfills directly from the specification, which also yields forward compatibility with future proposals.

Challenges of Spec-to-Code Translation. Although ECMA-262 is written in natural language, its conventions and structures allow systematic translation into an intermediate representation [34]. Still, automatically generating practical polyfills faces two fundamental challenges.

- **Internal Representation of Control Flow.** The specification represents JavaScript-level control flow, such as exceptions, using *completion records*, an internal data type with no JavaScript counterpart. Reifying them as objects is correct but slow, while blindly inserting `try-catch` is unsound when a completion is stored or passed as a value (§2.2). Knowing where native control flow suffices requires static analysis.
- **Algorithmic Inefficiency.** The specification defines *what* a feature must do, not *how* to do it efficiently, so a literal translation often yields algorithmically suboptimal code impractical for production use. Moreover, such inefficiencies

Set.prototype.isSupersetOf (*other*)

1. Let *O* be the **this** value.
2. Perform ? RequireInternalSlot(*O*, [[SetData]]).
3. Let *otherRec* be ? GetSetRecord(*other*).
4. If SetDataSize(*O*.[[SetData]]) < *otherRec*.[[Size]], return **false**.
5. Let *keysIter* be ? GetIteratorFromMethod(*otherRec*.[[SetObject]], *otherRec*.[[Keys]]).
6. Let *next* be NOT-STARTED.
7. Repeat, while *next* is not DONE,
   a. Set *next* to ? IteratorStepValue(*keysIter*).
   b. If *next* is not DONE, then
      i. If SetDataHas(O.[[SetData]], *next*) is **false**, then
         1. Perform ? IteratorClose(*keysIter*, NormalCompletion(UNUSED)).
         2. Return **false**.
8. Return **true**.

(a) The built-in algorithm of `Set.prototype.isSupersetOf`.

```
1  module.exports = function isSupersetOf(other) {
2    var O = aSet(this);                      // Step 1-2
3    var otherRec = getSetRecord(other);      // Step 3
4    if (size(O) < otherRec.size) return false;// Step 4
5    var iterator = otherRec.getIterator();   // Step 5
6    // Step 6-7.b
7    return iterateSimple(iterator, function (e) {
8      // Step 7.b.i
9      if (!has(O, e)) {
10       // Step 7.b.i.1
11       return iteratorClose(iterator,'normal',false);
12     }
13   }) !== false; // Step 7.b.i.2, 8
14 };
```

(b) The polyfill implementation of (a) in `core-js`.

Figure 1: The algorithm of `Set.prototype.isSupersetOf` in ECMA-262 for ES2025 and its polyfill implementation in `core-js`.

cannot be fixed locally: changing the internal data structure of one operation breaks every other operation on the same value, so purely syntactic rewriting is unsound.

Our Approach. To overcome these challenges, we propose MiniPoly, a framework that automatically compiles ECMA-262 specifications into correct and efficient JavaScript polyfills. First, it uses an existing tool called ESMeta [36] to parse the natural-language specification into a structured metalanguage representation. Then, we design a static analysis that resolves completion records, the specification's internal control-flow representation. It systematically translates them into equivalent JavaScript constructs (e.g., `try-catch`) that faithfully mimic the intended semantics. Finally, we introduce *rewrite rules* that replace algorithmically suboptimal specification patterns with highly efficient JavaScript implementations during compilation.

Contributions. Mechanized specifications and rewrite rules are individually established techniques. Our novelty is the first end-to-end pipeline from the mechanized specification to production-grade polyfills, resting on two new formulations.

- We define a spec-to-JS translation using a *completion type analysis* that converts the specification's completion records into explicit JavaScript control flow.
- We design a rewrite-rule system with *symbolic path-based constraints* for semantic-aware algorithmic optimization.
- MiniPoly generates polyfills more spec-compliant than production-ready libraries yet competitively efficient.

2 Background and Motivation

2.1 Background

JavaScript Language Specification. ECMA-262 [11] is the official language specification of JavaScript, defining the language's syntax and semantics. It describes the expected behavior of syntax constructs and built-in APIs in algorithmic steps using natural language and internal data types. For example, Figure 1a shows the built-in algorithm for `Set.prototype.isSupersetOf` in ES2025 that checks whether a set is a superset of another set. In brief, it validates the receiver *O* and the argument *other* and compares their sizes (Steps 1–4), then iterates over *other*, checking each element for membership in *O* and closing the iterator early once an element is missing (Steps 5–8). The specification has evolved from 536 algorithms in 2016 to 960 algorithms in 2025, reflecting the continuous growth of JavaScript's feature set and complexity.

Specification Metalanguage. ECMA-262 defines JavaScript not in JavaScript itself but in a prose *metalanguage*, for two reasons. First, the standard must remain *implementation-agnostic*: it captures the consensus behavior shared by many independent engines (e.g., V8 and JavaScriptCore), so it cannot be tied to any one implementation, and defining JavaScript in terms of only JavaScript would be circular. Second, it must describe runtime mechanics that lie below user-space JavaScript, such as internal slots (or fields) and cross-realm behavior, which JavaScript cannot express. The metalanguage therefore computes over specification-internal data structures (e.g., Lists, Records, and completion records) rather than JavaScript values. Running the specification as JavaScript means translating these structures away; most map naturally to JavaScript, but some do not, the most prominent being the completion record.

Completion Records. A *completion record* reifies JavaScript-level control flow and contains the following three fields:

- [[Type]]: the type of completion, which can be NORMAL, THROW, RETURN, BREAK, or CONTINUE.
- [[Value]]: the value associated with the completion, which can be any JavaScript value.
- [[Target]]: the target of a break or return completion, which can be a label or function name.

We call a completion a *normal completion* if its [[Type]] is NORMAL; otherwise, it is an *abrupt completion*. The operator "? *x*" is a shorthand for ReturnIfAbrupt(*x*), which performs the following steps:

1. If *x* is an abrupt completion, return Completion(*x*).
2. Otherwise, set *x* to *x*.[[Value]].

Thus, "? GetSetRecord(*other*)" in Step 3 of Figure 1a directly returns its return value if it is an abrupt completion. While there are four kinds of abrupt completions, abstract operations reachable from built-in algorithms only return throw completions.

IteratorClose (*iteratorRecord*, *completion*)
It takes *an Iterator Record* and *a Completion Record*.
...
2. Let *innerResult* be Completion(GetMethod(*iterator*, **"return"**)).
3. If *innerResult* is a normal completion, then
   ...
   c. Set *innerResult* to Completion(Call(*return*, *iterator*)).
4. If *completion* is a throw completion, return ? *completion*.
5. If *innerResult* is a throw completion, return ? *innerResult*.
...

Figure 2: Complex JavaScript control flows in IteratorClose, invoked in Step 7.b.i.1 of Figure 1a.

SetDataHas (*setData*, *value*)
1. If SetDataIndex(*setData*, *value*) is NOT-FOUND, return **false**.
2. Return **true**.

SetDataIndex (*setData*, *value*)
...
4. Repeat, while *index* < size, // *Linear search for value in setData*
   a. Let *e* be *setData*[*index*].
   b. If *e* is not empty and *e* is *value*, then
      i. Return *index*.
   c. Set *index* to *index* + 1.
5. Return NOT-FOUND.

Figure 3: SetDataHas and SetDataIndex operations invoked in Step 7.b.i of Figure 1a.

Polyfill. Polyfills are JavaScript implementations of built-in APIs defined in ECMA-262, enabling the use of modern features in older environments that lack native support. Leading libraries are maintained by hand: developers translate the natural-language specification into JavaScript that mimics the intended semantics, a process that is both error-prone and labor-intensive. For instance, `core-js` [29], the most widely adopted library, is essentially maintained by a single developer (7,059 of 7,746 commits, over 91%), an unsustainable way to track a rapidly expanding specification.

Manual maintenance is also fragile because the specification keeps changing. Editions often refactor algorithms without altering behavior, breaking the one-to-one mapping between the specification and hand-written code. This makes implementations hard to keep correct and to debug. The two libraries cope differently. `core-js` diverges into its own abstractions for performance. For example, Figure 1b shows its `Set.prototype.isSupersetOf` built on custom helpers such as `iterateSimple`. Such abstractions are not guaranteed to match the specification. One even conflated `Array` and `TypedArray`, requiring repeated ad-hoc fixes [6]. `es-shims` instead preserves the mapping via `es-abstract`, but must rewrite each affected operation by hand whenever the specification is updated.

2.2 Motivating Example

We use `Set.prototype.isSupersetOf` as a running example to illustrate two challenges in automated extraction.

Challenge 1: Internal Representation of Control Flow. As explained in §2.1, completion records represent JavaScript-level control flow and are not directly convertible to JavaScript. Assume that ? is used directly at a call site, as in Step 3 of Figure 1a:

3. Let *otherRec* be ? GetSetRecord(*other*)

Then, we can drop it and emit the following JavaScript code:

```
var otherRec = GetSetRecord(other);
```

However, algorithms often do not directly handle completion records with the ? operator. Instead, they:

(1) store them in variables and handle them in later steps, or
(2) pass them as parameters to other abstract algorithms.

For example, Figure 2 shows the IteratorClose algorithm invoked in Step 7.b.i.1 of `Set.prototype.isSupersetOf`, which defines how to close an iterator when an abrupt completion occurs during iteration. Unlike direct uses of the ? operator, IteratorClose takes a completion record as the parameter *completion* and stores the completion record returned from GetMethod in *innerResult*; both are used in later steps to determine which completion to propagate. Thus, to faithfully mimic the intended semantics, we need to use `try-catch` to capture exceptions from GetMethod and rethrow them only if the *completion* parameter does not already represent an abrupt completion.

An apparent alternative is to reify completion records as JavaScript objects and thread them as values. However, the polyfill's public boundary must return plain values and throw real exceptions, native operations raise raw exceptions rather than JavaScript objects representing completion records, and every fallible step would need explicit wrapping and unwrapping. To systematically resolve this control-flow mismatch, we instead design a static analysis that infers the completion type of each value, letting us selectively translate completions into JavaScript control flow using `try-catch` only when necessary.

Challenge 2: Algorithmic Inefficiency. The ECMA-262 specification prioritizes clear and unambiguous semantic definitions over computational efficiency. Consequently, a literal translation of the specification's algorithmic steps often results in suboptimal performance that is unsuitable for production use. For example, as shown in Figure 3, `Set.prototype.isSupersetOf` performs membership checks by invoking the SetDataHas abstract operation. This operation further relies on SetDataIndex, which is defined as a linear search over an internal List representation. This results in $O(N)$ time complexity for a single lookup, making a full superset check an $O(N \times M)$ operation. In contrast, production-standard libraries such as `core-js` employ highly optimized hash-based structures to ensure $O(1)$ average-case lookups. Furthermore, manual attempts to optimize these bottlenecks frequently introduce subtle spec-miscompliance bugs, as developers may overlook edge-case side effects defined deep within the specification's logic.

To resolve this without sacrificing correctness, MiniPoly provides a set of declarative *rewrite rules* with which users define custom optimization patterns. Each rule replaces an identified algorithmic bottleneck with an efficient implementation (e.g., an $O(1)$ hash lookup) while abstracting away the underlying specification

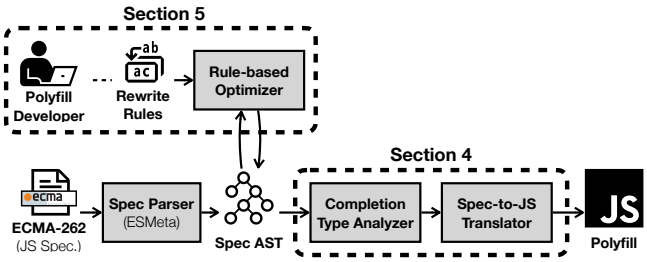


Figure 4: Overall structure of MiniPoly.

logic. The compiler then seamlessly integrates these optimized implementations while preserving the semantics.

3 Overall Structure of MiniPoly

As illustrated in Figure 4, MiniPoly compiles the ECMA-262 specification into a JavaScript polyfill through four components:

Spec Parser. MiniPoly begins by leveraging an existing tool, ESMeta [36], to parse natural-language algorithmic steps in ECMA-262 into a structured Abstract Syntax Tree (AST) representation.

Rule-based Optimizer (§5). To address inherent algorithmic inefficiencies, the Rule-based Optimizer applies developer-provided *rewrite rules* to the parsed AST, each matching a specification pattern and rewriting it into an efficient equivalent. Beyond structural matching, rules add *symbolic path-based constraints* via a *where* clause and support nested *sub-rules* for multi-level transformations, letting users precisely target suboptimal specification patterns.

Completion Type Analyzer (§4.2). To bridge the semantic gap between the specification and native JavaScript, MiniPoly must resolve *completion records*, the specification-internal representation of control flow. The Completion Type Analyzer performs a flow-sensitive static analysis that infers the completion type of each value (i.e., a normal completion, an abrupt completion, or a bare value).

Spec-to-JS Translator (§4.3). Guided by the analysis, the Spec-to-JS Translator performs a *context-aware translation* that lowers the specification directly to JavaScript. It emits completion-handling code (e.g., `try-catch`) only where the analysis reports that a value may be an abrupt completion, and translates the remaining constructs in a syntax-directed manner.

4 Eliminating Internal Control Flows for Correct Spec-to-JS Translation

As motivated in §2.2, completion records cannot be translated directly to JavaScript. MiniPoly resolves them in two stages. First, a *completion type analysis* statically infers the completion type of each variable (i.e., a normal completion, an abrupt completion, or a bare value). Then, a *spec-to-JS translation* lowers each abstract operation to JavaScript, inserting completion-handling code (e.g., `try-catch`) only where a value may actually be an abrupt completion. We formalize both stages over a small imperative core language and state their soundness and correctness. The full semantics and proofs are in the supplementary material.

4.1 Core Language

The core language captures the features of the specification meta-language relevant to completion records. Its grammar is as follows.

$$
\begin{array}{lll}
\textbf{Operations} & f \in \mathrm{Op} & ::= f(\overline{x_i : \tau_i}) : \tau \ \{\ s;\ \mathsf{return}\ x\ \} \\
\textbf{Statements} & s \in \mathrm{Stmt} & ::= \mathsf{skip} \mid x := e \mid x := f(x_1, \ldots, x_n) \\
 & & \mid\ x := \mathsf{normal}(y) \mid x := \mathsf{abrupt}(y) \\
 & & \mid\ s_1;\ s_2 \mid \mathsf{if}\ (e)\ s\ s \mid \ldots \\
\textbf{Expressions} & e \in \mathrm{Expr} & ::= p \mid x \mid e_1 \oplus e_2 \mid \mathsf{normal?}\ x \\
 & & \mid\ \mathsf{abrupt?}\ x \mid \mathsf{get}(x) \mid \ldots \\
\textbf{Values} & v \in \mathrm{Value} & ::= p \mid \mathsf{normal}(p) \mid \mathsf{abrupt}(p) \mid \ldots \\
\textbf{Types} & \tau \in \mathrm{Ty} & ::= \mathtt{bool} \mid \mathtt{num} \mid \mathtt{normal}\langle\tau\rangle \\
 & & \mid\ \mathtt{abrupt}\langle\tau\rangle \mid \tau \vee \tau \mid \ldots \\
\textbf{States} & \sigma \in \mathrm{State} & = \mathrm{Var} \rightharpoonup \mathrm{Value}
\end{array}
$$

An *operation* f takes typed parameters, runs a statement, and returns the value of a variable. The key feature is that completion records appear at the value level: a *value* is a bare *primitive* p (e.g., a boolean or number), a normal completion $\mathsf{normal}(p)$, or an abrupt completion $\mathsf{abrupt}(p)$. Completions are *constructed* by $x := \mathsf{normal}(y)$ and $x := \mathsf{abrupt}(y)$, and *inspected* by $\mathsf{normal?}\ x$, $\mathsf{abrupt?}\ x$ (testing the tag) and $\mathsf{get}(x)$ (reading the payload). For simplicity we assume A-normal form [37] for completion constructs, which therefore take only variable arguments, and sound type annotations. The language's (standard) denotational semantics comprises $O[\![-]\!] : \mathrm{Value}^* \rightharpoonup \mathrm{Value}$ for operations, $\mathcal{S}[\![-]\!] : \mathrm{State} \rightharpoonup \mathrm{State}$ for statements, and $\mathcal{E}[\![-]\!] : \mathrm{State} \rightharpoonup \mathrm{Value}$ for expressions.

4.2 Completion Type Analysis

The completion type analysis is a flow-sensitive abstract interpretation that, at each program point, over-approximates the completion type of every variable.

Abstract Domain. The analysis abstracts a value by a *completion type* from the four-point lattice below: normal (resp. abrupt) means a normal (resp. abrupt) completion or a bare primitive, $\bot$ means definitely a bare primitive, and $\top$ is no information. A *type environment* Γ assigns a completion type to each variable.

$$
\textbf{Type Environment}\ \ \Gamma \in \mathrm{TyEnv} = \mathrm{Var} \to \mathrm{CompTy}
$$

$$
\textbf{Completion Types}\ \ \tau_c \in \mathrm{CompTy} = \left\{\ \begin{array}{ccc} & \top & \\ \mathsf{normal} & & \mathsf{abrupt} \\ & \bot & \end{array}\ \right\}
$$

The order $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$ are the lattice's, lifted point-wise to type environments. A concretization γ maps each completion type to the values it denotes. An environment Γ *soundly abstracts* a state σ, written $\Gamma \models \sigma$, when $\sigma(x) \in \gamma(\Gamma(x))$ for all x.

Abstract Semantics. The abstract semantics mirrors the concrete one, but computes on type environments. It comprises a statement transformer and an expression abstraction:

$$
\widehat{\mathcal{S}[\![-]\!]} : \mathrm{TyEnv} \to \mathrm{TyEnv} \qquad \widehat{\mathcal{E}[\![-]\!]} : \mathrm{TyEnv} \to \mathrm{CompTy}
$$

A completion constructor sets its variable to normal or abrupt, and a call $x := f(\ldots)$ to $\alpha(\tau)$, the completion type of f's declared return type τ (via a fixed map $\alpha : \mathrm{Ty} \to \mathrm{CompTy}$). Every expression yields $\bot$ except a variable, which keeps its environment type. The only rule of interest is the conditional, which *refines* the environment

along each branch by exploiting the known value of the guard before joining the two outcomes:

$$\mathcal{S}[\![\widehat{\text{if } (e)\ s_1\ s_2}]\!](\Gamma) = \widehat{\mathcal{S}[\![s_1]\!]}(\Gamma_1) \sqcup \widehat{\mathcal{S}[\![s_2]\!]}(\Gamma_2)$$

where $\Gamma_1 = \text{refine}(\Gamma, e, \#\text{t})$ and $\Gamma_2 = \text{refine}(\Gamma, e, \#\text{f})$. The function refine sharpens a variable's type when the guard fixes its completion type: if normal? x holds, it meets x's type with normal (with abrupt when it fails), and dually for abrupt? x. It recurses through the boolean connectives and leaves the environment unchanged on guards it cannot exploit.

Soundness. Soundness states that the analysis never misclassifies a completion: whenever Γ abstracts a state σ, the environment computed for a statement abstracts the resulting state.

Theorem 4.1 (Soundness of Completion Type Analysis). *For any statement s, type environment Γ, and state σ such that $\mathcal{S}[\![s]\!](\sigma)$ is defined,*

$$\Gamma \models \sigma \implies \widehat{\mathcal{S}[\![s]\!]}(\Gamma) \models \mathcal{S}[\![s]\!](\sigma).$$

The proof is by induction on s, using monotonicity and meet-exactness of γ and the soundness of refine at conditionals.

4.3 Spec-to-JS Translation

Guided by the analysis, MiniPoly lowers each operation to JavaScript, written $\mathcal{T}[\![-]\!]$, inserting completion-handling code only where the analysis reports that a value may be abrupt.

Representing Completions with Flag Variables. A spec variable x is represented by a JavaScript variable `x` holding its *payload* (the value stripped of its completion tag), together with a boolean *flag* `x_flag` recording the tag. The flag is materialized exactly when the analysis gives $\Gamma(x) \neq \bot$; a value that is definitely bare carries no flag. Abrupt completions are held as ordinary values rather than raised, so control flow is preserved without allocating completion records; an actual exception is thrown only when an abrupt completion reaches a return. The encoding assumes the specification never mixes completion and bare values in one variable; under this assumption a JavaScript store ρ *represents* a spec state σ under an environment Γ, written $\rho \triangleright_\Gamma \sigma$, when their payloads agree and each materialized flag records the correct tag.

Translating Calls and Checks. The interesting case is an operation call $x := f(\ldots)$, chosen by the completion type $\alpha(\tau)$ of f's declared return type τ. When the operation may complete either way ($\alpha(\tau) = \top$), we invoke it inside a `try-catch`. A normal return falls through and sets the flag to `true`, while a thrown value is caught, bound to `x`, and flagged `false`:

$$\mathcal{T}[\![x := f(\overline{x_i})]\!] \;=$$

```
try { var x=f(pass(x̄ᵢ)); var x_flag=true; }
catch (e) { var x=e; var x_flag=false; }
```

where the helper pass expands an argument list, adding a flag for every non-bare argument. The other three cases specialize this by exactness of $\alpha(\tau)$: exactly abrupt always throws (drop the normal branch), exactly normal never throws (drop the `try-catch`), and $\bot$ returns a bare value needing no flag. A check reads the flag: $\mathcal{T}[\![\text{normal? } x]\!]$ = `x_flag` and $\mathcal{T}[\![\text{abrupt? } x]\!]$ = `!x_flag` when $\Gamma(x) \neq \bot$, and both are constantly `false` when x is definitely bare.

Translating Operations and Returns. An operation becomes a JavaScript function whose parameter list is expanded by the same pass helper, so a completion-typed parameter arrives together with its flag. The return is lowered by the completion type of the returned variable: when it may be either kind ($\Gamma(x) = \top$) it becomes `if (x_flag) return x; else throw x;`; when exactly abrupt, `throw x;`; and otherwise (normal or bare) `return x;`.

Correctness. Correctness rests on the representation relation $\rho \triangleright_\Gamma \sigma$: the translation preserves it, so the payloads and flags in the JavaScript store always mirror the spec state.

Theorem 4.2 (Correctness of Spec-to-JS Translation). *Let $f(\overline{x_i : \tau_i}) : \tau$ { s; return x } be an operation, $\overline{v_i}$ arguments, and $\rho_0 \triangleright_{\Gamma_0} [\overline{x_i \mapsto v_i}]$ an entry store. Whenever $O[\![f]\!](\overline{v_i})$ is defined, running $\mathcal{T}[\![f]\!]$ from ρ_0 terminates,* returning p *when $O[\![f]\!](\overline{v_i})$ is normal(p) or a bare p, and* throwing p *when it is abrupt(p).*

The proof is a forward simulation on the height of the source derivation, using soundness (Theorem 4.1) to ensure that every completion-carrying variable has a materialized flag.

4.4 Remaining Manual Effort

While MiniPoly extracts most abstract operations automatically, some require manual handling. We address the following four categories; the number of operations in each is reported in §6.2. The first three are inherent limitations of polyfilling, whereas the fourth reflects a temporary gap in the upstream ESMeta toolchain.

Internal Methods and Slots. Internal methods and slots are not directly accessible from JavaScript, so we manually replace them with equivalent JavaScript constructs. For example, we implement `[[Call]]` using a function call and `[[Construct]]` using the `new` operator. When an internal slot must be accessed directly, we simulate it with a non-enumerable property whose name is mangled to avoid collisions with user-visible properties.

Realm. A realm is an isolated global environment with its own set of intrinsics, and is not directly accessible from JavaScript, so polyfills cannot support cross-realm identity checks. We therefore assume a single realm: cross-realm references are routed to their single-realm counterparts (e.g., `globalThis`), and behaviors that cannot be meaningfully routed are removed. Deciding which references are safe to route or remove depends on each operation's realm semantics, so this step is performed manually.

Host-defined. The specification defines only the required behavior of host-defined operations, without a concrete algorithm, so nothing can be extracted automatically. For example, the abstract operation `HostEnqueuePromiseJob` must schedule a Promise job for later execution, which is host-dependent (e.g., provided as `queueMicrotask` in browser and Node.js hosts, and absent in others). We implement such operations manually in a minimal functional form.

ESMeta Unsupported. These cases arise when ESMeta cannot yet parse certain natural-language steps in the specification. Such statements are emitted as `YetStep` placeholders and must be implemented manually. However, this effort will diminish as ESMeta's coverage improves.

```
pattern: |
  1. For each element {{ elem : var }} of {{ base : ref }}, do
    1. If {{ elem }} is not ~empty~ and SameValue({{ elem }},
          {{ value : var }}) is *true*, {{ body : step }}
where:
  - isSetData(base)
  - { isSetData: "_ (.[[SetData]] | .[[WeakSetData]]) .copy*" }
replace: 1. If SET_DATA_HAS({{ base }}, {{ value }}) is *true*, {{ body }}
subrules:
  - pattern: {{ elem }}
    replace: {{ value }}
```

(a) A rewrite rule for index-based lookup patterns.

```
1. For each element _e_ of _S_.[[SetData]], do
  1. If _e_ is not ~empty~ and SameValue(_e_, _value_)
        is *true*, then
    1. Return _S_.
```

(b) An excerpt of `Set.prototype.add` in the ECMA-262 spec.html.

```
1. If SET_DATA_HAS(_S_.[[SetData]], _value_) is *true*,
     Return _S_.
```

(c) Result of applying the rewrite rule to the excerpt.

Figure 5: A rewrite rule for index-based lookup patterns in `Set` and `WeakSet` and its application to the `Set.prototype.add` built-in.

5 Rewrite Rules for Optimization

A *rewrite rule* defines a transformation that identifies specific fragments in the specification's abstract syntax trees (ASTs) and replaces them with equivalents. It consists of four core components:

- **Pattern.** An AST template with embedded meta-variables that match concrete AST fragments.
- **Where Clause.** Semantic constraints on the captured meta-variables that must be satisfied for the rule to apply.
- **Replacement.** An optional AST template that specifies how to rewrite the matched pattern, with meta-variables substituted by their captured AST fragments.
- **Sub-rules.** Optional nested rules applied recursively to the replacement AST, enabling multi-level transformations.

For example, Figure 5 illustrates a rewrite rule that optimizes index-based membership checks in `Set` and `WeakSet` built-ins. By replacing the original $O(N)$ linear search with a single call to the `SET_DATA_HAS` function, the rule achieves $O(1)$ performance for membership lookups.

Patterns and Replacements. A *pattern* is an AST template whose embedded *meta-variables* match concrete AST fragments. A meta-variable $\{\{x[:\tau]^?\}\}$ represents a typed placeholder, and its matching behavior is determined by its optional type annotation $\tau \in \{\texttt{var}, \texttt{ref}, \texttt{expr}, \texttt{cond}, \texttt{step}, \texttt{step*}\}$:

- $\{\{x:\tau\}\}$ acts as a *definition* that matches any concrete AST fragment of the specified syntactic category and captures it for later pattern matching and substitution.
- $\{\{x\}\}$ performs a *consistency check* against a previously captured AST based on their inferred symbolic paths, ensuring that they refer to the same underlying value.

Captured meta-variables are then substituted into the *replacement* template, filling the meta-variables it shares with the pattern. The *sub-rules* allow for multi-level transformations by applying additional patterns, and they can use the meta-variable captured by the parent rule. The main rule often has only a pattern and no replacement, serving as a context matcher for its sub-rules.

Symbolic Path-based Constraints. Optimizing one operation in isolation is unsafe: replacing the linear `[[SetData]]` list behind `Set.prototype.has` with a hash map breaks `intersection`, which still reads that list by index. A rewrite must therefore identify *every* access to the same underlying value, which is what the symbolic paths below capture. The where clause specifies *additional constraints* on captured meta-variables, beyond structural matching, expressed over the symbolic paths inferred by analysis.

Symbolic Path	$\pi \in \Pi$	$::= \omega \mid \pi.f \mid \pi.\text{copy} \mid \dots$
Symbolic Objects	$\omega \in \Omega$	
Predicates	p	$::= \_ \mid .f \mid .\text{copy} \mid p \cdot p \mid p \mid p \mid p^*$

A *symbolic path* π is either a base symbolic object, field access on a symbolic path, or a copy relation extending another symbolic path. A *symbolic object* ω represents an arbitrary value, but the same object indicates the same underlying value. A *predicate* p is a pattern over symbolic paths that checks whether a captured variable satisfies a property. Its semantics $[\![p]\!] : \mathcal{P}(\Pi) \to \mathcal{P}(\Pi)$ maps a set of symbolic paths to those that satisfy p.

$$
\begin{array}{ll}
[\![\_]\!](X) = X & [\![p^*]\!](X) = \bigcup_{n\geq 0} [\![p]\!]^n(X) \\
[\![.f]\!](X) = \{\pi.f \mid \pi \in X\} & [\![.\text{copy}]\!](X) = \{\pi.\text{copy} \mid \pi \in X\} \\
[\![p_1 \cdot p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!] & [\![p_1 \mid p_2]\!](X) = [\![p_1]\!](X) \cup [\![p_2]\!](X)
\end{array}
$$

We say that a captured variable with symbolic path π *satisfies* a predicate p if π is in the set of symbolic paths that satisfy p:

$$\pi \text{ satisfies } p \iff \pi \in [\![p]\!](\Pi)$$

For example, a predicate `_ (.[[SetData]] | .[[WeakSetData]]) .copy*` defined in Figure 5a describes any symbolic path followed by a field access of `[[SetData]]` or `[[WeakSetData]]`, and then by zero or more copy relationships.

Symbolic Path Analysis. The *symbolic path analysis* is a static analysis that infers the symbolic paths of variables in the specification. We use the following join ($\sqcup$) operator that only preserves the common suffix of two symbolic paths in the analysis:

$$\pi_1 \sqcup \pi_2 = \begin{cases} (\pi_1' \sqcup \pi_2').f & \text{if } \pi_1 = \pi_1'.f \wedge \pi_2 = \pi_2'.f \\ (\pi_1' \sqcup \pi_2').\text{copy} & \text{if } \pi_1 = \pi_1'.\text{copy} \wedge \pi_2 = \pi_2'.\text{copy} \\ \omega & \text{otherwise} \end{cases}$$

where ω is a fresh symbolic object. For example, $\omega_1.\text{a.b.c} \sqcup \omega_2.\text{c.c}$ is $\omega_3.\text{c}$ with a fresh object ω_3 because the common suffix is `.c`. Using this domain, we design a flow-sensitive, inter-procedural analysis based on abstract interpretation [7, 8] to infer the symbolic paths of variables. The analysis results are used in two ways:

Table 1: Correctness and execution time. Pass (%) shows passed Test262 tests and rates; Time is in seconds.

			Native		core-js		es-shims†		MiniPoly		MiniPoly-opt	
Category	**# Built-ins**	**Tests**	**Pass (%)**	**Time**	**Pass (%)**	**Time**	**Pass (%)**	**Time**	**Pass (%)**	**Time**	**Pass (%)**	**Time**
Array	24	4,713	4,713 (100.0)	25.5s	4,617 (98.0)	26.4s	4,355 (94.9)	48.5s	**4,713 (100.0)**	25.4s	**4,713 (100.0)**	25.6s
String	12	638	638 (100.0)	5.9s	616 (96.6)	6.2s	602 (94.4)	6.6s	**638 (100.0)**	6.0s	**638 (100.0)**	6.0s
Set	18	748	748 (100.0)	8.1s	710 (94.9)	8.5s	682 (91.7)	9.4s	**746 (99.7)**	8.2s	**746 (99.7)**	8.4s
Map	12	322	322 (100.0)	4.6s	298 (92.5)	4.8s	256 (79.5)	5.0s	**320 (99.4)**	4.7s	**320 (99.4)**	4.6s
WeakMap	5	198	198 (100.0)	2.2s	168 (84.8)	2.3s	–	–	**196 (99.0)**	2.2s	**196 (99.0)**	2.3s
WeakSet	5	166	166 (100.0)	2.0s	142 (85.5)	2.2s	–	–	**164 (98.8)**	2.1s	**164 (98.8)**	2.1s
Promise	13	1,264	1,264 (100.0)	8.9s	1,154 (91.3)	9.8s	272 (55.5)	19.8s	**1,256 (99.4)**	9.6s	**1,256 (99.4)**	9.6s
Total	**89**	**8,049**	**8,049 (100.0)**	**57.2s**	**7,705 (95.7)**	**60.1s**	**6,167 (90.9)**	**89.3s**	**8,033 (99.8)**	**58.1s**	**8,033 (99.8)**	**58.4s**

† `es-shims` evaluated on supported subset, and (–) indicates unsupported built-ins.

- To check the consistency of multiple captures of the same meta-variable by verifying that their inferred symbolic paths are exactly the same.
- To check that the inferred symbolic paths of meta-variables satisfy the predicate specified in the rule's where clause.

6 Evaluation

We evaluate MiniPoly with the following research questions:

- **RQ1 (Correctness):** Does MiniPoly generate spec-compliant polyfills comparable to industry standards?
- **RQ2 (Automation):** To what extent does MiniPoly automate manual polyfill development?
- **RQ3 (Optimization):** How effectively do the rewrite rules improve runtime performance?
- **RQ4 (Forward Compatibility):** How well does MiniPoly support new features in ES2026 and early-stage proposals?
- **RQ5 (Real-world Usability):** Can the generated polyfills serve as drop-in replacements in real-world applications?

Target Language Specification. All experiments target ES2025, the latest official edition of ECMA-262. For RQ4 (Forward Compatibility), we additionally target the pre-release version of ES2026 and a proposal in Stage 2.7 about the await dictionary feature.

Baselines. We compare MiniPoly against two baselines:

- **`core-js`:** The most widely adopted polyfill library. To address RQ1–RQ3, we compare MiniPoly with `core-js` v3.47.0, which is the final release of 2025. For RQ4, we use v3.48.0, which adds support for ES2026 proposals.
- **`es-shims`:** A collection of per-method polyfill packages maintained under the es-shims GitHub organization [15]. For all experiments we use the latest release of each package as of December 2025.

Execution Environment. All experiments run on the latest stable release of Node.js v24.14.1, which supports ES2025 features natively. We overwrite the target built-in from the global object with the polyfill implementation before evaluation.

Target Polyfills. ES2025 supports 490 built-in objects and methods. Among them, we target 89 methods across seven objects, as summarized in Table 1. We exclude the remaining built-ins from our evaluation for the following reasons:

(1) **Not *Polyfillable* (151):** These cannot be implemented in JavaScript, such as operations over internal slots (e.g., `Proxy`) or floating-point math (e.g., `Math.sin`); polyfill libraries do not support them either.
(2) **Baseline Features (112):** These built-ins were introduced in ECMAScript 3 (ES3, 1999), the minimum specification version assumed by MiniPoly. This baseline assumption is consistent with that of `core-js`.
(3) **Unsupported by ESMeta (119):** These are not yet modeled by ESMeta. They will be automatically supported once ESMeta includes them. `core-js` supports all built-ins in this category.
(4) **First-class continuations (19):** The `Iterator` built-ins are excluded because they require modeling first-class continuations; we leave this for future work. `core-js` supports all built-ins in this category.

6.1 RQ1: Correctness

We evaluate correctness using Test262 [12], the official ECMAScript conformance test suite. Table 1 summarizes the comparison of results. MiniPoly passes **8,033** of 8,049 tests (**99.8%**), where 16 failures stem from inherent limitations of polyfills, not from MiniPoly's approach. Excluding these polyfill-specific limitations, MiniPoly achieves **100%** conformance with Test262. On the other hand, `core-js` passes 7,705 tests (95.7%), and `es-shims` passes 6,167 out of its 6,785 applicable tests (90.9%).

Realm identity (all polyfills, 12 tests). These tests verify whether a built-in object created in one realm is correctly recognized by methods in another. Because polyfills operate strictly within a single realm, they cannot intercept engine-level cross-realm identity checks; consequently, all three polyfills fail identically.

List-as-Array modeling (all polyfills, 4 tests). A specification-compliant List could be implemented (e.g., as a linked list) and would pass all tests, but it is far less efficient than a native Array, allocating a separate object per element. Like major polyfills, we therefore approximate Lists with native Arrays.

Table 2: `core-js` v3.47 failures beyond the 16 failures shared by all polyfills and the 158 tests related to `[[Construct]]`.

#	Built-in	Root Cause
A1	`Array.prototype.map`	Call `Array.prototype` setter
A2	`Array.prototype.filter`	
A3	`Array.prototype.flat`	
A4	`Array.prototype.flatMap`	
A5	`Array.prototype.toReversed`	
A6	`Array.prototype.with`	
B1	`Array.from`	Mishandle `IteratorClose`
B2	`Promise.all`	
B3	`Promise.allSettled`	
B4	`Promise.race`	
B5	`Promise.any`	
B6	`Set.prototype.isDisjointFrom`	
B7	`Set.prototype.isSupersetOf`	
C1	`Array.prototype.flat`	Incorrect length assignment
C2	`Array.prototype.flatMap`	

Failures in `es-shims`. In the releases we evaluate, `es-shims` does not provide the ES2025 Set/Map methods, and its pass rate on the supported subset is 90.9%. Thus, we focus on the root causes of the failures in `core-js` rather than `es-shims`.

Failures in `core-js`. Beyond the 16 failures shared by all polyfills, `core-js` fails 328 additional tests. The largest category, comprising 158 tests, is related to the [[Construct]] internal method. The specification requires built-in methods to be non-constructable, but `core-js` implements them as ordinary `function` expressions, which are constructable. MiniPoly avoids this violation by emitting arrow functions and concise methods, both of which are natively non-constructable.[1] The remaining 170 failures are discussed below and summarized in Table 2.

A: Call `Array.prototype` *setter.* The specification assigns array elements with `CreateDataPropertyOrThrow`, which bypasses the prototype chain, but `core-js` v3.47 used direct index assignment. This invokes setters on `Array.prototype`, causing 46 test failures across the six methods in Table 2. We reported this bug to `core-js` [6], and it was fixed in v3.48:

```
if (IS_MAP) target[index] = result;     // v3.47
createProperty(target, index, result); // v3.48
```

B: Mishandle `IteratorClose`. When iteration terminates early, ECMA-262 requires the iterator to be closed via `IteratorClose`. For Set methods (B6–7), although `core-js` contains an explicit `iteratorClose` implementation, it mistakenly passed `iterator` rather than `iterator.iterator` to the method:

```
iteratorClose(iterator, 'normal', false);          // v3.47
iteratorClose(iterator.iterator, 'normal', false); // v3.49
```

As a result, the close operation was silently skipped. For Promise combinators (B2–4), `core-js` incorrectly handled completion propagation regarding `IteratorClose`. All of these failures were fixed in v3.49.

[1]This guarantee is naturally lost if MiniPoly's output is further transpiled to ES5 (e.g., via Babel [2]), as ES6+ methods are lowered back to ordinary `function` expressions.

Table 3: Implementation size (LoC) per built-in object.

		Baseline		MiniPoly		MiniPoly-opt		
Object	**#**	**`core-js`**	**`es-shims`**	**Auto**	**Man.**	**Auto**	**Man.**	**Rules**
Array	24	2,298	8,978	1,332	317			
String	12	1,281	6,885	418	130			
Set	18	2,313	5,880	864	190	861	283	114 (11)
WeakSet	5	1,882	–	387	123	379	214	
Map	12	1,986	3,062	756	217	747	327	81 (6)
WeakMap	5	2,034	–	446	115	436	231	
Promise	13	2,518	8,031	1,511	203			
Total†	**89**	**14,312**	**32,836**	**5,714**	**1,295**	**5,684**	**1,705**	**195 (17)**

† AOs shared by multiple built-ins are counted repeatedly.

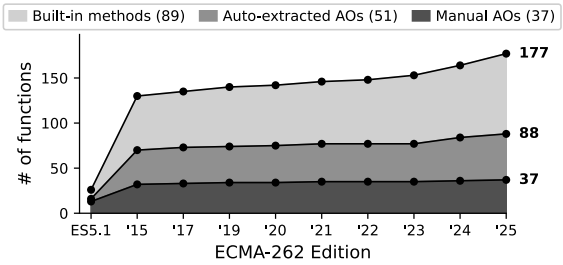


Figure 6: Cumulative functions across ECMA-262 editions.

C: Incorrect length assignment. For `Array.prototype.flat` and `Array.prototype.flatMap` (C1–2), `core-js` incorrectly assigned the numeric return value of the internal `flattenIntoArray` operation to the `length` property of the newly created array. We also reported this bug to `core-js` in the same pull request discussed in *A*, and it was fixed in v3.48:

```
A.length = flattenIntoArray(A, /* ... */); // v3.47
flattenIntoArray(A, /* ... */);            // v3.48
```

6.2 RQ2: Automation

Among the 89 built-in methods across seven target objects, MiniPoly automatically extracts all of them together with 51 of the 88 unique abstract operations (AOs) that they transitively reference, generating 5,714 lines of JavaScript. AOs are internal, method-like algorithms defined in the specification that ECMAScript implementations are required to support. The remaining 37 AOs require 1,295 lines of manual implementation, resulting in an overall *automation ratio* of **81.5%** (5,714/7,009 total LoC). We use lines of code (LoC) as a proxy for manual effort, as it reflects the code a developer must write, keep in sync with the evolving specification, and debug. This total counts shared AOs per built-in; deduplicated, it is only 492 LoC. Of the 37 manually implemented AOs, 19 (144 LoC) are for internal methods and slots, 3 (63 LoC) for realm semantics, and 2 (22 LoC) for host-defined operations; the remaining 13 (263 LoC) arise from temporary ESMeta gaps.

Marginal cost of new built-ins. The automation ratio of 81.5% does not fully capture the practical benefit of MiniPoly. As Figure 6 illustrates, the manual AOs are almost entirely *foundational*

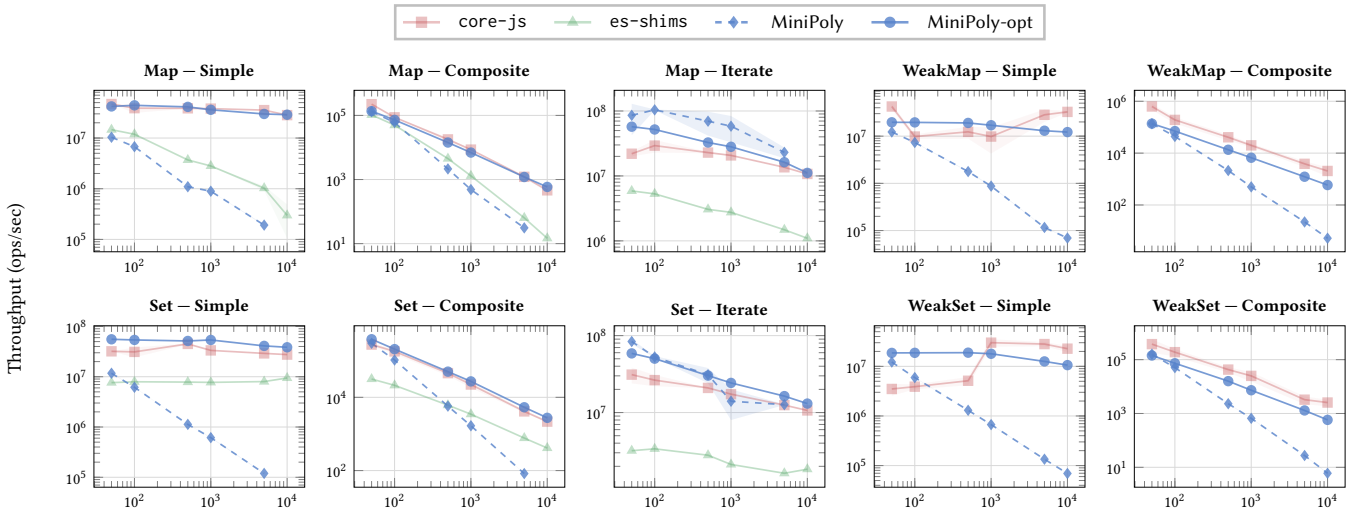


Figure 7: Throughput (ops/sec) across collection types, operation categories, and sizes (N). Dashed lines indicate MiniPoly (unoptimized); solid lines indicate MiniPoly-opt. We average over 5 runs, and the shaded area represents the standard deviation.

work that is reused across all subsequent built-ins. The 89 built-in methods share only 88 unique AOs, and after the initial editions establish the base set, the number of additional AOs required by later editions grows only gradually.

Comparison of manual implementation burden. For `core-js` and `es-shims`, *every* line is written and maintained by developers (Table 3): 14,312 LoC and 32,836 LoC, respectively. With MiniPoly, the developer's actual manual burden is only 1,295 LoC, a **11.1**× and **25.4**× reduction compared to `core-js` and `es-shims`, respectively. The remaining 5,714 lines are entirely auto-generated from the specification. When the specification evolves, only the affected operations are regenerated, so just this small set of manual AOs needs attention (§7). The manual cost for optimization is similarly low: 1,705 LoC of additional JavaScript modeling for the rule-based optimization and 195 LoC of 17 rewrite rules.

6.3 RQ3: Optimization

To evaluate the effectiveness of the DSL, we conduct a case study that implements optimizations for `(Weak)Set` and `(Weak)Map`. To replace linear index-based algorithms with fast-key-based algorithms, we define 11 rules for `(Weak)Set` and 6 rules for `(Weak)Map` in 195 lines of YAML files, together with 1,705 lines of JavaScript code to model the optimized internal data structures and operations. We also rewrite iteration patterns so that index-based accesses become calls to the optimized operations, though this does not change their asymptotic complexity.

Performance Measurement. We compare four implementations, `core-js`, `es-shims`, MiniPoly, and MiniPoly-opt (MiniPoly with rule-based optimization), on custom throughput benchmarks for each target method, varying the collection size N from 50 to 10,000. To reduce measurement noise, we warm up for 500ms so that JIT compilation and garbage collection stabilize.

Algorithmic Categorization. As summarized in Table 4, we classify the target methods into three algorithmic categories:

(1) Simple: The specification uses $O(N)$ linear searches, which our rewrite rules replace with $O(1)$ hash-based lookups.

(2) Iteration: Our rewrite rules replace the specification's index-based list access with a custom linked-list traversal, preserving insertion order while abstracting index-based operations.

(3) Composite: These operations combine iteration and lookups; although they have no dedicated rewrite rules, they benefit transitively from the optimized simple/iteration operations they invoke.

Table 4: Algorithmic categorization of built-in methods.

Category	Map / WeakMap	Set / WeakSet
Simple $O(1)$	`delete`, `get`, `has`, `set`, `size`	`add`, `delete`, `has`, `size`
Iteration $O(N)$	`clear`, `entries`, `forEach`, `keys`, `values`	`clear`, `entries`, `forEach`, `keys`, `values`
Composite $O(N \times M)$	`constructor`	`constructor`, `difference`, `intersection`, `isDisjointFrom`, `isSubsetOf`, `isSupersetOf`, `symmetricDifference`, `union`

Internal Data Structure. The rule-based optimization templates model keyed collections using a hash map indexed by a *fast-key* function for $O(1)$ element access, combined with a doubly linked list to preserve insertion order as required by the specification. This is the same technique `core-js` uses internally. The rewrite system is agnostic to the backing data structure: users may supply any that satisfies the interface expected by the rules.

Results. Figure 7 shows the throughput across all objects, collection types, and sizes.

(1) Simple. The unoptimized MiniPoly exhibits $O(N)$ degradation: throughput drops by two to three orders of magnitude as N grows from 50 to 10,000, confirming that its linear-search semantics are faithfully preserved but impractical. With rule-based optimization, MiniPoly-opt maintains near-constant throughput, matching or exceeding `core-js` and `es-shims` across all four types.

Table 5: Forward compatibility on recent and in-progress proposals. Pass (%): passed Test262 tests and pass rate. Throughput (ops/sec): mean of 5 runs at $N = 5{,}000$, shown JIT-enabled / JIT-disabled for `core-js` and MiniPoly-opt. (–): unsupported.

Proposal	Method	Tests	Correctness (Pass %)				Throughput (ops/sec) — N = 5,000			
			core-js	es-shims	MiniPoly	MiniPoly-opt	core-js	es-shims	MiniPoly	MiniPoly-opt
Upsert (ES2026)	Map.prototype									
	.getOrInsert	28	26 (92.9)	24 (85.7)	**28 (100.0)**	**28 (100.0)**	13.7M / 2.0M	61.2K	107K	8.2M / 2.5M
	.getOrInsertComputed	37	35 (94.6)	33 (89.2)	**37 (100.0)**	**37 (100.0)**	11.7M / 1.8M	40.8K	77.7K	6.5M / 1.9M
	WeakMap.prototype									
	.getOrInsert	34	26 (76.5)	—	**34 (100.0)**	**34 (100.0)**	18.4M / 3.1M	—	104K	8.2M / 2.3M
	.getOrInsertComputed	43	32 (74.4)	—	**43 (100.0)**	**43 (100.0)**	5.7M / 1.2M	—	73.1K	5.0M / 1.8M
Await Dictionary (Stage 2.7)	Promise									
	.allKeyed	12	—	—	**12 (100.0)**	**12 (100.0)**	—	—	—	—
	.allSettledKeyed	12	—	—	**12 (100.0)**	**12 (100.0)**	—	—	—	—
Total		**166**	**119 (83.8)**	**57 (87.7)**	**166 (100.0)**	**166 (100.0)**	11.3M / 1.9M	50.0K	89.3K	6.8M / 2.1M

(2) Iteration. Both MiniPoly and MiniPoly-opt perform comparably to `core-js`, as iteration is inherently $O(N)$; the linked-list traversal even removes the specification's repeated index-bounds checking.

(3) Composite. The unoptimized MiniPoly exhibits quadratic degradation: the `Map` and `Set` constructors invoke element insertion N times, each costing $O(N)$, yielding $O(N^2)$ overall. MiniPoly-opt removes this bottleneck through the transitively applied simple-operation optimizations, restoring $O(N)$ and matching `core-js` across all sizes. The same holds for the ES2025 composite Set methods, which combine iteration and membership checks.

6.4 RQ4: Forward Compatibility

Specification-driven generation supports new proposals with no new engineering. We evaluate this on the next edition's draft (ES2026) and an early-stage proposal (Table 5).

Draft of the Next Edition. ES2026 introduces several new built-ins, including `getOrInsert(Computed)` on `(Weak)Map.prototype` from the *Upsert* proposal [18]. MiniPoly generates these four methods with zero manual effort, and the rule-based optimization templates apply directly, yielding MiniPoly-opt with no extra DSL work. Polyfills generated by MiniPoly and MiniPoly-opt for these methods pass all 142 tests in Test262 (**100%**). In the JIT-enabled setting, MiniPoly-opt is 1.66× slower than `core-js`, whereas in the JIT-disabled setting it is 1.1× faster. Code-level inspection confirms that both share the same asymptotic complexity, so we hypothesize that `core-js`'s JIT-enabled advantage stems from its manual, JIT-friendly engineering. Once such patterns are identified, they could be encoded as rewrite rules to close the gap.

Early-stage proposal (Await Dictionary). To test applicability beyond imminent editions, we target the *Await Dictionary* proposal (Stage 2.7) [5], which adds `Promise.{allKeyed, allSettledKeyed}`. MiniPoly generates both, passing all 24 Test262 tests (**100%**), whereas neither `core-js` nor `es-shims` implements them. It thus delivers spec-compliant polyfills ahead of manual implementations, even while a proposal is still in draft.

6.5 RQ5: Real-world Usability

To assess whether MiniPoly's polyfills are usable as drop-in replacements in real-world code, we run the official test suites of seven widely used npm libraries spanning three domains: general-purpose utilities (`lodash`, `underscore`, `ramda`), asynchronous control flow (`async`, `axios`), and network/server frameworks (`express`, `ws`). For each, we replace every targeted built-in with its polyfill and run the suite. We exclude `es-shims` here: with partial coverage, unimplemented methods silently fall back to the native built-ins, so results would reflect the host engine rather than the polyfill under test. Table 6 reports per-library pass rates and execution times for Native, `core-js`, MiniPoly, and MiniPoly-opt.

Across all seven libraries (**10,795** tests), MiniPoly and MiniPoly-opt match `core-js` exactly, each passing **10,772** tests (**99.8%**). The 23 failures are concentrated in `lodash` and are shared by every polyfill: they stem from library-side assumptions about the native built-ins (e.g., `_.isNative` cannot succeed on a polyfill, which is by definition not a native function) rather than from any deficiency in the polyfill implementations. Execution time is also on par with `core-js` (within 1% in aggregate), confirming that MiniPoly's polyfills add no measurable overhead on real workloads. The lone outlier is `underscore`, whose suite stresses tight collection-iteration loops: unoptimized MiniPoly runs about **1.2**× slower than `core-js` on this workload (8.2s vs. 6.7s), but MiniPoly-opt's DSL-based rewrites close the gap entirely (6.7s). Overall, MiniPoly and MiniPoly-opt serve as drop-in replacements for `core-js` across these real-world applications, matching it in both correctness and performance.

7 Discussion and Limitations

Spec-faithfulness by construction. Unlike hand-written polyfills, MiniPoly preserves the specification's control flow and step ordering by construction. This eliminates subtle manual-translation bugs, such as the `core-js` issues in §6.1. Re-running the pipeline also propagates specification updates to all affected polyfills. Regeneration is modular: only the operations affected by a change are re-extracted, so developers need not rebuild from scratch. Optimizations encoded as rewrite rules also survive specification updates.

Table 6: Real-world usability across seven widely used JavaScript libraries. Pass (%): passing tests (and pass rate) from each library's official suite after replacing all targeted built-ins with their polyfills. Time: wall-clock test-suite execution (seconds). *Native*: unmodified built-ins, as a baseline.

			Native		core-js		MiniPoly		MiniPoly-opt	
Library	**Version**	**Tests**	**Pass (%)**	**Time**	**Pass (%)**	**Time**	**Pass (%)**	**Time**	**Pass (%)**	**Time**
lodash	4.17.21	6,794	6,794 (100.0)	9.7s	6,771 (99.7)	9.7s	6,771 (99.7)	9.7s	6,771 (99.7)	9.6s
underscore	1.13.7	208	208 (100.0)	6.6s	208 (100.0)	6.7s	208 (100.0)	8.2s	208 (100.0)	6.7s
ramda	0.30.1	1,192	1,192 (100.0)	0.9s	1,192 (100.0)	1.1s	1,192 (100.0)	1.0s	1,192 (100.0)	1.0s
axios	1.7.9	195	195 (100.0)	74.0s	195 (100.0)	74.6s	195 (100.0)	74.0s	195 (100.0)	74.1s
async	3.2.5	690	690 (100.0)	16.4s	690 (100.0)	16.5s	690 (100.0)	16.5s	690 (100.0)	16.4s
express	4.21.2	1,288	1,288 (100.0)	1.3s	1,288 (100.0)	1.4s	1,288 (100.0)	1.4s	1,288 (100.0)	1.4s
ws	8.18.0	428	428 (100.0)	2.2s	428 (100.0)	2.4s	428 (100.0)	2.3s	428 (100.0)	2.3s
Total		**10,795**	**10,795 (100.0)**	**111.0s**	**10,772 (99.8)**	**112.4s**	**10,772 (99.8)**	**113.1s**	**10,772 (99.8)**	**111.6s**

This faithfulness is formally guaranteed for the core language of §4 and validated empirically via Test262 for the extracted built-ins.

Dependency on ESMeta. MiniPoly's coverage is currently bound by ESMeta's parser. Unmechanized objects (e.g., `Math`, `Date`) and certain specification steps (§6.2) require manual handling. However, as ESMeta's coverage expands, these components can be fully automated without modifying MiniPoly's architecture.

External validity. We evaluate MiniPoly on Test262 in a modern JavaScript engine rather than legacy browsers. This isolates semantic correctness from transpilation artifacts and engine-specific quirks. Since production polyfills are authored in modern JavaScript and down-leveled by standard tools (e.g., Babel [2], Webpack [41]), legacy deployment is orthogonal to MiniPoly's contribution.

Future direction of rewrite rules. A current limitation of MiniPoly-opt is that its rewrite rules are manually designed and lack automated validation of their semantic correctness and performance benefits. As observed in RQ4, the resulting polyfills can therefore be slower than `core-js` in JIT-enabled environments, where carefully engineered libraries benefit from engine-specific optimizations. Future work could combine LLM- or synthesis-based rule generation with semantic and performance validation, generalizing per-method optimizations from existing implementations (e.g., `core-js`) into specification-wide transformations.

Scope of JavaScript-Expressible Semantics. MiniPoly extracts only the specification steps that have an equivalent JavaScript implementation, which is precisely what makes a built-in polyfillable. Constructs below the JavaScript level, such as internal slots, realms, and host-defined operations, are not polyfillable at all. They must instead be provided by the underlying engine or host environment. We therefore exclude the built-ins depending on such constructs and manually implement the remaining abstract operations (§6, §6.2). This restriction is inherent to the notion of a polyfill, which must itself be implemented in JavaScript.

8 Related Work

Mechanized language specifications. Mechanizing specifications is well-studied for JavaScript [4, 16, 17, 30], C [13, 23], and WebAssembly [40]. For JavaScript, JISET [34] and its successor ESMeta [36] pioneered extracting a mechanized IR from the ECMAScript specification for downstream analysis [22, 31–33, 35]. Complementary efforts such as `engine262` [14] and ECMA-SL [26] provide executable reference implementations, but they reify JavaScript values in their own representation, so their built-ins cannot be invoked on native values. Generating usable polyfills from such specifications thus remains unexplored. MiniPoly addresses this gap, building directly on ESMeta's metalanguage AST to become the first framework to produce production-grade JavaScript polyfills from the mechanized specification.

Specification-driven synthesis. Prior work synthesizes implementations from formal specifications via stepwise refinement [9], observed behavior [19], or verified lifting [3, 20]. Separately, semantic rewrite rules are widely used for compiler optimizations [24, 25, 39]. MiniPoly adapts this rewrite-rule paradigm to a novel domain: optimizing *specification-derived* code.

Synthesis and verification of rewrite rules. Complementary research automates the creation and validation of rewrite rules via equality saturation [28], program synthesis [27], or stochastic search [38]. Currently, MiniPoly's 17 rewrite rules are hand-written. Automatically synthesizing these rules or formally verifying their semantic preservation at the ESMeta IR level remain promising directions for future work.

9 Conclusion

We presented MiniPoly, a compilation framework that automatically extracts correct and optimized JavaScript polyfills directly from the ECMA-262 specification. It achieves 99.8% Test262 conformance, outperforming production-standard libraries like `core-js`, and seamlessly supports early-stage proposals to establish a robust foundation for verifiable web standards.

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597).

Data Availability

The software artifacts and datasets supporting the results of this study are available in a Zenodo repository [21] to facilitate replication and reuse. The repository includes the source code, experimental scripts, and raw output files used in our evaluation.

References

[1] Abdul Haddi Amjad and Nishu Goel. 2024. Web Almanac 2024: JavaScript. https://almanac.httparchive.org/en/2024/javascript#library-usage. Accessed: 2026-03-04.

[2] Babel contributors. 2026. Babel – The compiler for next generation JavaScript. https://babeljs.io. Accessed: 2026-03-27.

[3] Sahil Bhatia, Jie Qiu, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. 2024. Verified Code Transpilation with LLMs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems (NeurIPS)*. Article 1310, 31 pages. doi:10.52202/079017-1310

[4] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudžiūnienė, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 87–100. doi:10.1145/2535838.2535876

[5] Ashley Claymore, Jordan Harband, and Chris de Almeida. 2024. TC39 Proposal: Await Dictionary. https://github.com/tc39/proposal-await-dictionary. Accessed: 2026-03-27.

[6] core-js contributors. 2026. Fix: Use `createProperty` in array iteration methods. https://github.com/zloirock/core-js/pull/1498. Accessed: 2026-03-04.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/512950.512973

[8] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation (JLC)* 2, 4 (1992), 511–547. doi:10.1093/logcom/2.4.511

[9] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 689–700. doi:10.1145/2676726.2677006

[10] Alexis Deveria. 2026. Online report on browser support for `Set.prototype.isSubsetOf`. https://caniuse.com/mdn-javascript_builtins_set_issubsetof. Accessed: 2026-03-25.

[11] ECMA International. 2025. *ECMA-262, 16th edition, ECMAScript ®2025 Language Specification.* https://262.ecma-international.org/16.0

[12] ECMA International. 2026. *Test262: A conformance test suite for the latest draft of ECMAScript.* https://github.com/tc39/test262

[13] Chucky Ellison and Grigore Roşu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/2103656.2103719

[14] engine262 contributors. 2026. engine262 – An implementation of ECMA-262 in JavaScript. https://github.com/engine262/engine262. Accessed: 2026-07-31.

[15] es-shims contributors. 2026. es-shims: ECMAScript Shims. https://github.com/es-shims. Accessed: 2026-03-26.

[16] José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript Verification Toolchain. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 50 (2018), 33 pages. doi:10.1145/3158138

[17] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *European Conference on Object-Oriented Programming (ECOOP)*. 126–150. doi:10.1007/978-3-642-14107-2_7

[18] Jonas Haukenes and Daniel Minor. 2025. TC39 Proposal: Upsert. https://github.com/tc39/proposal-upsert. Accessed: 2026-03-26.

[19] Stefan Heule, Manu Sridharan, and Satish Chandra. 2015. Mimic: Computing Models for Opaque Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 710–720. doi:10.1145/2786805.2786875

[20] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 711–726. doi:10.1145/2908080.2908117

[21] Jungwoong Kim, Youngmin Cho, and Jihyeok Park. 2026. *MiniPoly: Automatic Extraction of Efficient JavaScript Polyfill from Language Specification.* doi:10.5281/zenodo.21722035

[22] Kanguk Lee, Seunghwan Kim, Jihyeok Park, and Sukyoung Ryu. 2026. Selective Feature-Sensitive Coverage for Conformance Testing of Programming Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (April 2026). Just Accepted. doi:10.1145/3808231

[23] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115. doi:10.1145/1538788.1538814

[24] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 65–79. doi:10.1145/3453483.3454030

[25] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 22–32. doi:10.1145/2737924.2737965

[26] Luís Miguel Alves Loureiro. 2021. *ECMA-SL: A Platform for Specifying and Running the ECMAScript Standard.* Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[27] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proceedings of the ACM on Programming Languages* 8, OOPSLA, Article 120 (2024), 29 pages. doi:10.1145/3649837

[28] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 119 (2021), 28 pages. doi:10.1145/3485496

[29] npm, Inc. 2026. core-js – npm. https://www.npmjs.com/package/core-js. Accessed: 2026-03-04.

[30] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 346–356. doi:10.1145/2737924.2737991

[31] Jihyeok Park, Seungmin An, and Sukyoung Ryu. 2022. Automatically Deriving JavaScript Static Analyzers from Specifications Using Meta-Level Static Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. doi:10.1145/3540250.3549097

[32] Jihyeok Park, Seungmin An, Wonho Shin, Yusung Sim, and Sukyoung Ryu. 2021. JSTAR: JavaScript Specification Type Analyzer using Refinement. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1109/ASE51524.2021.9678781

[33] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification. In *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 13–24. doi:10.1109/ICSE43902.2021.00015

[34] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 647–658. doi:10.1145/3324884.3416632

[35] Jihyeok Park, Dongjun Youn, Kanguk Lee, and Sukyoung Ryu. 2023. Feature-Sensitive Coverage for Conformance Testing of Programming Language Implementations. In *Proceedings of the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3591240

[36] Sukyoung Ryu and Jihyeok Park. 2024. JavaScript Language Design and Implementation in Tandem. *Commun. ACM* 67, 5 (2024), 86–95. doi:10.1145/3624723

[37] Amr Sabry and Matthias Felleisen. 1993. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation (LASC)* 6 (1993), 289–360. doi:10.1007/BF01019462

[38] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 305–316. doi:10.1145/2451116.2451150

[39] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. 2022. Complexity-Guided Container Replacement Synthesis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA, Article 68 (2022). doi:10.1145/3527312

[40] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*. 53–65. doi:10.1145/3167082

[41] webpack contributors. 2026. webpack – A bundler for JavaScript and friends. https://webpack.js.org. Accessed: 2026-03-27.

Received 2026-03-27; accepted 2026-06-18