IRIDIUM: A Framework for Statically Optimizing JavaScript Programs

MEETESH KALPESH MEHTA, Indian Institute of Technology Bombay, India
ANIRUDH GARG, Indian Institute of Technology Bombay, India
ANEEKET YADAV, Indian Institute of Technology Delhi, India
MANAS THAKUR, Indian Institute of Technology Bombay, India

Static analysis of JavaScript remains notoriously difficult due to the language's dynamically typed nature, unconventional scoping rules, and pervasive side effects. Unlike mature infrastructures such as LLVM for C/C++ or Soot for Java, comparable frameworks for JavaScript are fragmented and limited in scope. In this paper, we introduce IRIDIUM, a first-of-its-kind framework to statically optimize JavaScript programs. IRIDIUM systematically lowers JavaScript into a structured intermediate representation (called IRI) that models bindings, environments, and control flow explicitly. The resultant expressiveness enables more predictable analyses and transformations, ranging from dataflow tracking to optimization passes to executable code generation for existing runtimes, that are otherwise hindered by the language's complexity. By bridging the gap between JavaScript's surface syntax and the requirements of static analysis, IRIDIUM, thus, lays the foundation for a new generation of tools that can reason effectively about modern JavaScript applications.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Compilers**; **Just-in-time compilers**; *Dynamic analysis.*

Additional Key Words and Phrases: Dynamic languages, Static analysis, JIT compilation.

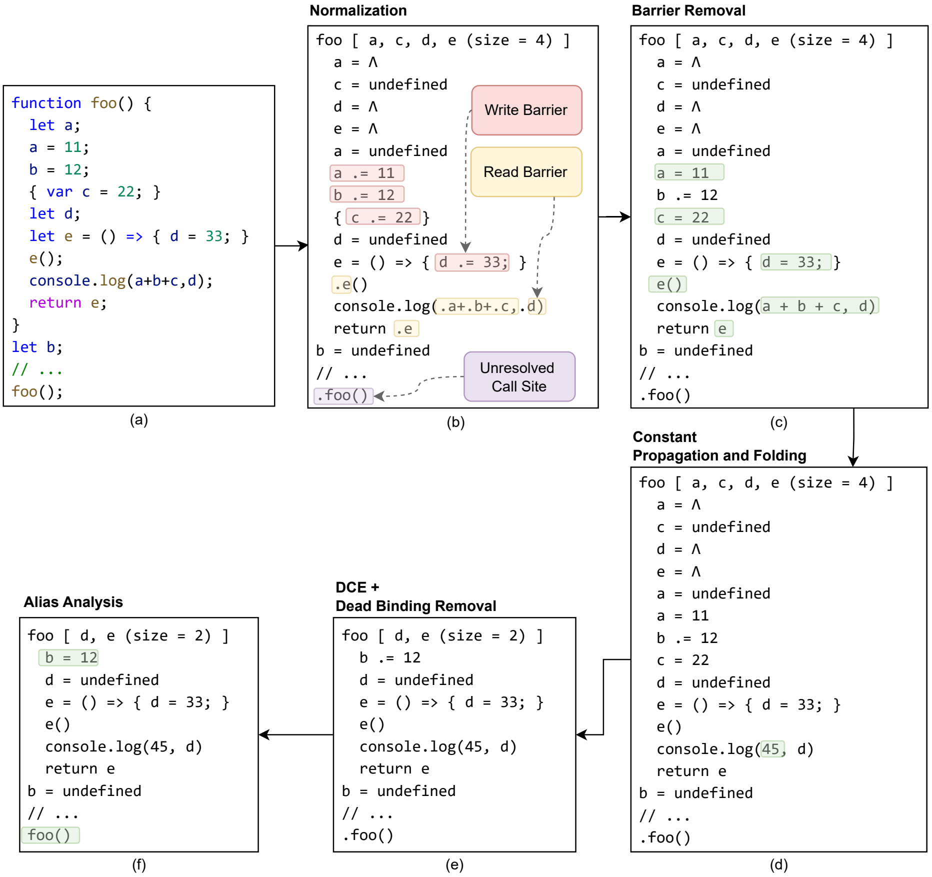
ACM Reference Format:
Meetesh Kalpesh Mehta, Anirudh Garg, Aneeket Yadav, and Manas Thakur. 2026. IRIDIUM: A Framework for Statically Optimizing JavaScript Programs. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 165 (April 2026), 41 pages. https://doi.org/10.1145/3798273

1 INTRODUCTION

JavaScript, once a simple scripting language for adding interactivity to web pages, has evolved into a cornerstone of modern software development. Its reach extends far beyond the browser, powering server-side applications with Node.js, mobile apps via frameworks like React Native, and even desktop software. This ubiquity has driven an insatiable demand for higher performance. However, JavaScript's dynamic and flexible nature, a key to its popularity, presents profound challenges for compilers striving to generate efficient machine code. Features like dynamic typing, complex scoping rules, closures, and a mutable global object create a semantic minefield for optimization. Consequently, ahead-of-time static analysis of JavaScript programs is considered hard, and most prior efforts to statically analyze JavaScript programs have limited themselves to non-optimizing applications that do not require soundness or completeness [17, 34].

Authors' Contact Information: Meetesh Kalpesh Mehta, Indian Institute of Technology Bombay, Mumbai, India, meeteshmehta@cse.iitb.ac.in; Anirudh Garg, Indian Institute of Technology Bombay, Mumbai, India, 22b1005@iitb.ac.in; Aneeket Yadav, Indian Institute of Technology Delhi, New Delhi, India, cs1221116@cse.iitd.ac.in; Manas Thakur, Indian Institute of Technology Bombay, Mumbai, India, manas@cse.iitb.ac.in.

ACM 2475-1421/2026/4-ART165
https://doi.org/10.1145/3798273



(a)

```
function foo() {
  let a;
  a = 11;
  b = 12;
  { var c = 22; }
  let d;
  let e = () => { d = 33; }
  e();
  console.log(a+b+c,d);
  return e;
}
let b;
// ...
foo();
```

Normalization

(b)

```
foo [ a, c, d, e (size = 4) ]
  a = Λ
  c = undefined
  d = Λ
  e = Λ
  a = undefined
  a .= 11
  b .= 12
  { c .= 22 }
  d = undefined
  e = () => { d .= 33; }
  .e()
  console.log(.a+.b+.c,.d)
  return .e
b = undefined
// ...
.foo()
```

Barrier Removal

(c)

```
foo [ a, c, d, e (size = 4) ]
  a = Λ
  c = undefined
  d = Λ
  e = Λ
  a = undefined
  a = 11
  b .= 12
  c = 22
  d = undefined
  e = () => { d = 33; }
  e()
  console.log(a + b + c, d)
  return e
b = undefined
// ...
.foo()
```

Constant Propagation and Folding

(d)

```
foo [ a, c, d, e (size = 4) ]
  a = Λ
  c = undefined
  d = Λ
  e = Λ
  a = undefined
  a = 11
  b .= 12
  c = 22
  d = undefined
  e = () => { d = 33; }
  e()
  console.log(45, d)
  return e
b = undefined
// ...
.foo()
```

DCE + Dead Binding Removal

(e)

```
foo [ d, e (size = 2) ]
  b .= 12
  d = undefined
  e = () => { d = 33; }
  e()
  console.log(45, d)
  return e
b = undefined
// ...
.foo()
```

Alias Analysis

(f)

```
foo [ d, e (size = 2) ]
  b = 12
  d = undefined
  e = () => { d = 33; }
  e()
  console.log(45, d)
  return e
b = undefined
// ...
foo()
```

Fig. 1. A motivating example showing various transformations that are enabled by the IRIDIUM framework.

To illustrate a few challenges in statically optimizing JavaScript programs, consider the example in Figure 1. The initial code (Figure 1a), though concise, is riddled with semantic ambiguities. For instance, the variable b is assigned a value without a local declaration, making it an implicit write to a global (or lexical) variable that could be aliased anywhere in the program. The variable c, declared with var, is function-scoped despite its position within a block – a historical quirk that violates modern block-scoping principles. Most critically, the arrow function e forms a closure, capturing the variable d from its parent scope, which often necessitates the runtime to allocate d on the heap and insert expensive run-time checks, or "barriers", to manage its lifetime and access. Language features like these make it hard to reason about the program behavior, and make it very difficult to soundly transform JavaScript programs. As a result, most real-world JavaScript optimization happens in managed runtimes (such as Google's V8 [12], Apple's JavaScriptCore [6], and Mozilla's SpiderMonkey [27]), based on expensive profile-guided speculations made during JIT compilation. There is no tool that allows one to explore how much of the language can be optimized ahead-of-time in presence of standard execution pipelines, and much of classic static-analysis research remains untried on JavaScript programs.

In this paper, we present a framework for statically optimizing JavaScript programs called IRIDIUM that solves many of the problems discussed above and for the first time allows one to explore such opportunities for JavaScript in a sound manner. IRIDIUM converts source JavaScript programs, through a series of lowering passes, to a simple and analyzable intermediate representation called IRI, and provides various elements to write static analyses and transformations. It also includes a code generation pass for a popular JavaScript runtime (without being tied to it), which allows it to act as a transparent middle layer in traditional JavaScript execution pipelines, while ensuring straightforward correctness of transformed programs.

IRIDIUM's pipeline begins with a *normalization pass* that translates ambiguous JavaScript code into a precise intermediate representation (IR); see Figure 1b. This IR gets rid of implicit scopes created by various JavaScript constructs, and makes the program's underlying structure explicit: `foo [ a, c, d, e ]` defines the logical stack frame, Λ marks variables as uninitialized (we refer to this value as `NUBD` or No-Use Before-Def), and memory operations are qualified as safe (=) or potentially unsafe (`.=`, `.x`), clearly delineating which accesses require run-time barriers.

With the program's semantics now explicit, we can apply a series of targeted optimizations. For the example under consideration, we can perform a *barrier removal* pass (Figure 1c) that analyzes variable lifetimes using an underlying temporal dead zone (TDZ) analysis. The TDZ analysis finds that the accesses to `a`, `c`, and `d` (within the closure) do not occur within their temporal dead zones and are not subject to TDZ failures at runtime. This allows us to remove the costly barriers, converting unsafe operations (`.=`) into simple, direct assignments (=). This may further lead to a cascade of classic compiler optimizations. For instance, in presence of a write barrier associated with `somevar.=10`, the variable `somevar` gets read before the write happens, making it *live* and consequently rendering a transformation pass such as dead-code elimination useless – while barrier removal may allow the corresponding write to be removed if found dead.

After barrier removal, a *constant propagation* pass (Figure 1d) would observe that the variables `a` and `c` are assigned constant values (`11` and `22`), which are used in the expression `a+b+c`. While `b` remains a symbolic global, the known parts of the expression can be pre-calculated to `33`. In this example, however, we can safely propagate `b` as well, because the write barrier to `b` would have already thrown an error if the corresponding write would have failed. This allows us to further fold the expression down to `45`. Following constant propagation, a *dead code and dead binding elimination* pass (Figure 1e) can prune unnecessary parts of the program. As the variables `a` and `c` previously only used in the `console.log` expression were constant-folded, their declaration and assignment can be identified as dead code and removed entirely. This shrinks the function's logical stack frame, reducing its memory footprint. Finally, an *alias analysis* pass (Figure 1f) can reason about the scope of the function `foo`. If it can prove that the call to `foo` only happens after it escapes its TDZ (and `foo` does not escape the caller scope), it can remove the final barrier on the access to `b`.

Observe that the sequence of transformations described above systematically refines the ambiguous source code into a highly optimized form, removing semantic overhead and paving the way for efficient code generation. We have implemented several such analysis and optimization passes in IRIDIUM, with a neat separation between classic and JavaScript-specific optimizations.

One may wonder how static analysis and optimization may compare with a traditional JIT pipeline for JavaScript programs (such as V8 and SpiderMonkey). Though many of the optimizations one could perform statically could also be performed by a JIT compiler (albeit with high runtime overhead), some limitations of a JITted system are worth pointing out. A traditional JIT compiler analyzes and optimizes code lazily, focusing only on "hot" functions that are executed frequently. In our example, the function `foo` returns the closure `e`. If `e` is later invoked in a hot loop, the JIT may lack the global context of its creation. Consequently, a JIT typically optimizes based on profile-guided *speculation*, which requires run-time checks to prevail as *guards*. As an instance, consider the bytecode generated for e by Mozilla's SpiderMonkey [27], which contains run-time checks to validate the lexical environment as shown below:

```
00000:   7  Int8 33                     # 33
00002:   7  GetAliasedVar "d" (hops = 0, slot = 4) # 33 d
00007:   7  CheckAliasedLexical "d" (hops = 0, slot = 4) # 33 d
00012:   7  Pop                         # 33
00013:   7  SetAliasedVar "d" (hops = 0, slot = 4) # 33
00018:   7  Pop                         #
00019:   7  RetRval                     #
```

The CheckAliasedLexical instruction, which verifies that the binding for d has not been reconfigured, represents overhead that a JIT may never be able to elide without a more powerful ahead-of-time analysis, like the one we can perform with a tool like IRIDIUM. Thus, in this paper, we invert the premise and put forth the following as a motivation for IRIDIUM in presence of JITted systems: If an optimization can be performed statically, it is better to perform it ahead-of-time — this could not only reduce the burden of a JIT, but also help it perform more aggressive optimizations. Furthermore, there are systems such as Amazon's LLRT [4] that cannot afford a JIT, and operating systems such as iOS that would not allow a third-party JIT because of the security concerns it may pose. Static analysis, enabled by IRIDIUM, comes to rescue in all these situations.

As stated previously, prior static analysis efforts for JavaScript have limited themselves to applications (such as suggesting call targets for IDEs) that do not pose challenges in case only a subset of the language was considered or if the static analysis was unsound. However, with IRIDIUM, our goals include enabling optimizations, ensuring correctness, and generating executable code. Toward this end, we have extended IRIDIUM with a code-generation pass for the QuickJS VM [7], which is used popularly in embedded and cloud-based systems [4]. We take optimized IRI and generate QuickJS bytecode for a plethora of standard JavaScript benchmark suites, and run it through the QuickJS runtime, not only validating correctness but also improving performance (on average, 2.5×, over a wide range of benchmark programs). This makes IRIDIUM the first-of-its-kind runtime-agnostic static optimizer for JavaScript, that can transparently fit in as a middle layer in traditional JavaScript pipelines by writing a code-generator pass for any existing back-end.

Contributions:

- A layered intermediate representation (IR) that exposes hidden JavaScript semantics (for the official ECMAScript 2025 language specification [1]) and makes static analysis feasible.
- A static-analysis framework for JavaScript programs, consisting of utilities to express transfer functions and express dataflow analyses.
- A set of standard static analysis and transformation passes that help optimize JavaScript code and demonstrate the utility of the presented IR and analysis framework.
- An instruction-selection pass that generates code for an existing JavaScript backend, with a detailed evaluation that shows obtained and potential performance improvements.

The rest of the paper is organized as follows. Section 2 gives an overview of the various stages of our proposed IRIDIUM framework. Section 3 describes how we simplify JavaScript code to a three-address-like form (called 3JS). Next, Sections 4 and 5 describe various stages of the core IRIDIUM pipeline that at the end generates executable bytecode for the QuickJS runtime. Section 6 samples various analyses and optimizations made possible by and implemented in IRIDIUM. Section 7 evaluates IRIDIUM against the baseline QuickJS, and studies various aspects of the improvements obtained. Finally, Section 8 sheds light on close related works and Section 9 concludes the paper.

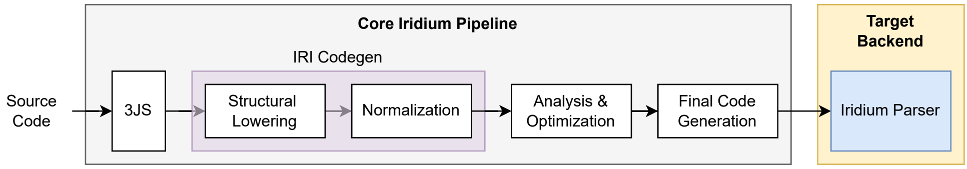


Fig. 2. Overview of the IRIDIUM pipeline.

2 THE IRIDIUM FRAMEWORK

Figure 2 illustrates the IRIDIUM pipeline. We first convert JavaScript source to a three-address-like subset called 3JS, which allows us to perform differential testing over a simpler language. We then translate 3JS into IRIDIUM's primary intermediate representation called IRI, targeted toward analysis and optimization. The translation to IRI involves Structural Lowering and Normalization, which is followed by Analysis & Optimization, and Final Code Generation; we describe these stages next. The language specifications of 3JS and IRI are presented in Appendices A and B, respectively.

Stage 1: Structural Lowering. This stage reduces the parsed source into a structured Three-Address-Code (TAC) form. All implicit JavaScript semantics—such as hoisting, scoping, and control-flow behaviour—are made explicit. This ensures that subsequent analyses can operate over a precise, semantically grounded representation rather than a syntactic AST.

Stage 2: Normalization. The TAC is then normalized to conform to the intermediate representation IRI. Each statement and expression are rewritten into a canonical form, enforcing uniformity across the representation. The result of this and the previous stage is a consistent and analyzable IR where all language constructs are reduced to a small, well-defined core.

Stage 3: Analysis & Optimization. This stage runs a collection of static analyses, whose results are used to eliminate redundant computations, fold constants, and merge effectful operations, while strictly preserving JavaScript semantics. This stage is orchestrated by a *pass manager*, which schedules and executes analyses and transformations in a certain dependency order.

Stage 4: Final Code Generation. Lastly, the Final CodeGen pass lowers optimized IRI into a compact, serializable form ready for backend consumption. This output is passed to the Target Backend, where an Iridium Parser reconstructs executable bytecode and executes the program.

2.1 Design Choices

We made a few design choices while designing IRI, based on prevalence of JavaScript features and our goal of being able to generate executable code. These choices allow us to not only design a layered IR (a "core" extended with JavaScript specifics), but also affect the design of various analysis and optimization passes as discussed throughout the paper.

To begin with, we believe that the effectiveness of an intermediate representation is often derived not from its flexibility, but from its restrictions. For instance, in the language R where lazy binding is prevalent, an effective IR must model its complexity using first-class environments [9]. On the other hand, JavaScript only allows lazy binding through a *legacy* and rarely used `with` statement whose use is discouraged in future ECMAScript code[1]. This derives our ***first design choice:*** *IRI will model logical stack frames explicitly.*

Next, observe that a program's execution inside a JavaScript closure is usually comprised of three fundamental operations: (i) local and remote environment accesses; (ii) synchronous and asynchronous operations that can either pause execution or create first class continuations; and (iii) calls to functions often with hidden contexts. This leads to our ***second design choice:*** *IRI will*

[1] https://tc39.es/ecma262/multipage/ecmascript-language-statements-and-declarations.html

```
// Source JavaScript code:
let {
  user: {
    profile: {
      contact: {  emails: [ primaryEmail, , backupEmail = "@example" ] }
    } } } = fetchConfig();
// Transformed 3JS code:
let js3$1 = fetchConfig();
let { user: objPat$2 } = js3$1;
// ... further simplifications ...
let condRes$9 = assnPat$8 === undefined;
let tempID$10 = assnPat$8 = condRes$9 ? "@example" : assnPat$8;
let backupEmail = assnPat$8;
```

Fig. 3. Example showing conversion of JavaScript code to 3JS format.

model local and remote environment accesses explicitly, provide native support for asynchronisity, and desugar all call instructions into simpler constructs.

Despite having support for a far extensible class prototype based inheritance model, large programs use classes just as they might be used in Java. We believe Java provides the best balance of object-oriented features while being a perfect candidate for static analysis based optimizations. Hence, in IRIDIUM, we imagine object operations such as field dereferences and functions calls belonging to a Java-like language, which means the core IRI supports classes and basic OOP operations natively (this inspires static analysis focus towards proving parts of JavaScript code to conform to this simpler semantics, making the existing rich body of research work applicable to IRIDIUM). Thus, our ***third design choice:*** *Core IRI will model Java-like object-oriented operations natively, while complex JavaScript semantics will be an extension to this language.*

Finally, reckon that most dynamic languages like JavaScript often use a stack-based virtual machine for execution. Hence, in order for IRIDIUM-processed code to be usable, we have our ***fourth design choice:*** *Core IRI will natively support stack operations.*

In the next sections, we describe the important stages employed by the IRIDIUM pipeline. The syntax of IRI is added to the supplementary material for reference (Appendix B), and the complete language specification is available online [25].

3 LOWERING SOURCE CODE INTO TAC

An IR facilitates analysis by lowering source code into a more manageable form, a popular one being three-address code (TAC). TAC can also break down syntactic sugar which is not required by the analysis stage of the framework. The IRIDIUM pipeline begins by lowering high-level JavaScript into an intermediate form called 3JS. This representation is designed as a TAC-like subset of JavaScript, acting as a bridge between the source code and IRIDIUM's core IR.

A defining feature of 3JS is that while reducing complex source nodes to simpler constructs, it remains a strict subset of JavaScript. As an example, consider the JavaScript source code and its corresponding 3JS version in Figure 3. Observe that in the 3JS version, the nested and default-value assignments are broken down into a linear sequence of simple operations, clarifying the program's execution flow. Also, while tools such as Babel [21] can be used to perform similar tasks, their transformations can introduce unnecessary bloat and are not always semantics-preserving, rendering them unsuitable for our use case.

```
var t = "outer"
function f1(a = t, t = "args" ){
  var t = "inner"
  return a;
}
console.log(f1()); //ReferenceError: Cannot access t before initialization
```

(a)

```
var t = "outer";
function f1() {
  var a = arguments.length<=0 || arguments[0]===undefined ?   t    :
      arguments[0];
  var t = arguments.length<=1 || arguments[1]===undefined ? "args" :
      arguments[1];
  return (function () { var t = "inner"; return a; })();
}
console.log(f1()); //Output: undefined
```

(b)

```
@t = "outer"
@f1 [ARG(a) ARG(t) a t, _]
  @a = Λ @t = Λ
  @a = ARG(a) === undefined ? @t : ARG(a) //Reference error on reading @t
  @t = ARG(t) === undefined ? "args" : ARG(t)
  @t = "inner"
  return a
console.log(@f1())
```

(c)

Fig. 4. (a) Source JavaScript code; (b) invalid transformation by Babel; (c) valid transformation by IRIDIUM.

As an example, consider the code snippet in Figure 4(a). The code is expected to throw a reference error to t as it is accessed before initialization[2]. Upon transforming the code using Babel, we obtain the code in Figure 4(b). Not only is the transformed code incorrect, it introduces the arguments object to the scope. This makes analysis much harder than it was before, because in JavaScript, the arguments object can conditionally alias to the actual passed arguments. As a result, the above transformation would make even simple optimizations like copy propagation impossible without a flow-sensitive alias analysis. On the contrary, IRI models evaluation order explicitly, avoiding the need to introduce the arguments object and wrapping the function in a new closure scope.

Having described why we cannot use existing tools like Babel, we now illustrate a few interesting scenarios that we encounter while generating 3JS. Consider the code snippet in Figure 5(a), where an anonymous function is passed as an argument to the function foo. During evaluation, JavaScript implicitly assigns a name to a function expression if it is assigned to a variable or appears as the value of a computed/static field of an object. This name is fixed at creation and cannot be changed later. When attempting to reduce code into a TAC-like form, one may be tempted to simply extract the anonymous function into a temporary variable, but it introduces semantic differences (see Figure 5(b)). This is surprising because it violates the general expectation that

[2] Arguments in JavaScript behave like var bindings. The general expectation is that a var binding is always initialized (to undefined) before entering a scope so it can never realistically throw a reference error.

```
function foo(a) {
 console.log(a.name)
}
foo(() => {}) // ""
```

(a)

```
function foo(a) {
 console.log(a.name) }
let t1 = () => {}
foo(t1) // "t1"
```

(b)

```
function foo(a) {
 console.log(a.name) }
let t2 = [() => {}][0]
foo(t2) // ""
```

(c)

Fig. 5. (a) Source JavaScript code; (b) invalid transformation; (c) valid transformation.

functions are first-class citizens whose behavior should not change when moved into a temporary variable. Consequently, an IR that must preserve semantics cannot universally apply this seemingly simple transformation. Figure 5(c) shows a valid transformation (as employed by 3JS) where the anonymous function is placed inside an array and immediately read from.

As a second example, evaluation scopes in JavaScript can become complex in scenarios that introduce special evaluation environments implicitly, such as during the evaluation of default arguments in functions or while processing class declarations. Consider the following simplified testcase from Test262 [8], the official ECMAScript conformance suite:

```
var fun1; var fun2; var C = "outer";
C = class C extends (fun1=function (){return C;}, fun2=function (){C=null
    ;}){};
fun2(); // Error: Assignment to constant variable
```

Here, a class `C` is defined with its parent class determined by evaluating the sequence expression `(e1, e2, ...)`. Closures are assigned to `fun1` and `fun2` during this evaluation. When `fun2()` is invoked on line 3, it throws an `assignment-to-constant-variable` error, even though no constants are explicitly declared in the source. This occurs because JavaScript implicitly creates a special evaluation scope where the binding for `C` is a constant, ultimately holding the class value. Therefore, any closures capturing `C` must reference this implicit environment as their evaluation scope.

A potential workaround in simpler cases is to wrap the expression in an anonymous function:

```
(EXPRESSION) => (() => { let RES = ...LOWER_EXPRESSION; return RES; })()
```

However, this transformation fails for expressions involving asynchronous or generator constructs:

```
(a + yield) => (() => { let t1 = a; let t2 = yield; return t2; })()
// Syntax error
```

Essentially, when the wrapped expression references asynchronous primitives such as `yield` (which pauses the current function and returns control to the caller), the transformation is unsound and produces a syntax error. This example illustrates that evaluation scopes in JavaScript can be subtle and error-prone, and naive attempts to reduce them into simpler code often fail. To solve this issue, in IRI, we introduce explicit well-defined evaluation scopes where bindings are resolved to their exact logical stack frames; this is discussed in Section 4.

4 STRUCTURAL LOWERING

The allure of JavaScript lies not in its complexity, but in its familiarity. For the most part, JavaScript *looks, works, and feels* like C++ or Java. Our guiding principle in designing IRIDIUM was to strip away JavaScript's syntactic and semantic complexity, systematically reducing it to a simpler, more analyzable intermediate language. IRIDIUM aims to provide a representation of JavaScript where bindings and scopes are explicit, control flow is well-defined, and there are no hidden semantic traps such as hoisting or implicit coercion. As shown in Figure 2, we generate this representation in two stages: structural lowering (described next) and normalization (Section 5).

The structural lowering stage is responsible for producing the first structured, graph-based representation of a JavaScript program in IRIDIUM. Its primary role is to translate the Three-Address-Like (3JS) input into a formal Three-Address Code (TAC) representation embedded within a Control Flow Graph (CFG). The lowering process performs three fundamental tasks:

1. Resolve implicit scope bindings and make them explicit. Hidden or implicitly created bindings in JavaScript are introduced explicitly into the IRI through special nodes. The `this` pointer and `import.meta` object are added to top-level scopes, while `this` and `arguments` are introduced within function scopes. Class-level closures receive bindings for prototype-related entities such as the home object and the `super()` constructor reference.

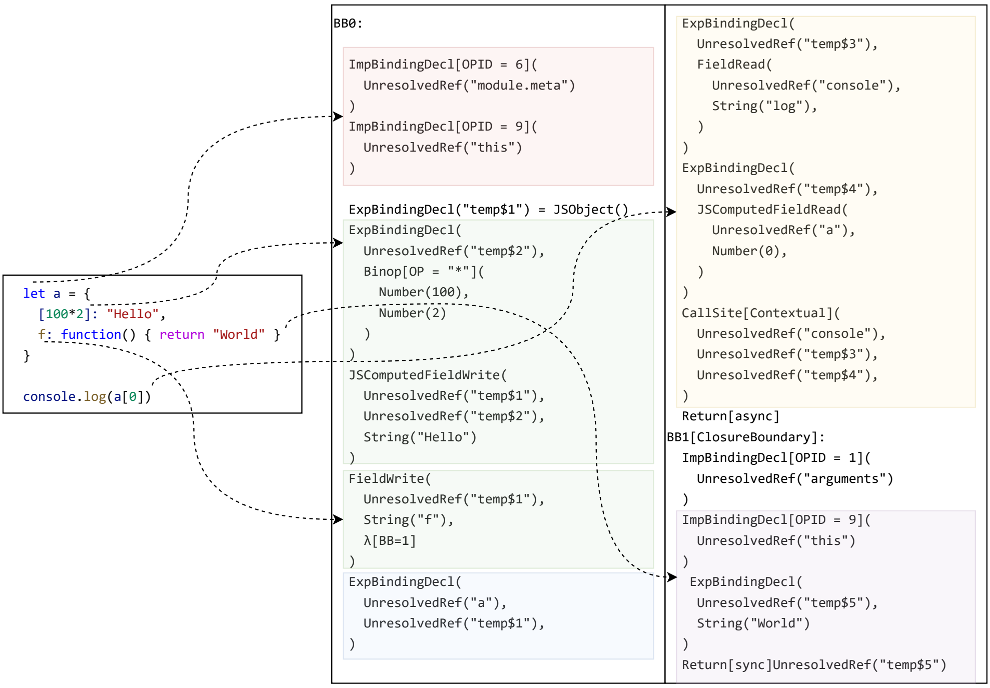
2. Make control flow explicit. High-level control constructs are decomposed into primitive control flow operations. Complex loop forms, such as `for-of`, `for-in`, and `while`, are lowered into explicit iteration logic including iterator initialization, next-value retrieval, and termination tests. Each basic block represents a linearized sequence of TAC instructions with explicit control-flow edges.

3. Desugar high-level constructs. High-level syntactic sugar is systematically decomposed into primitive operations. For example, class declarations are desugared into closure-based constructs, exposing constructor creation, method initialization, and prototype linkage. Similarly, default parameter initializers are lowered into assignments at the beginning of the function body.

The result of structural lowering is a well-structured CFG where complex JavaScript semantics are made explicit but not yet fully resolved. Non-local control transfers—such as labeled `break` and `continue` statements—remain annotated but unresolved. Control transfers that exit function or `try` block boundaries are similarly marked, allowing later passes to insert the appropriate finalizer or context-pop logic. Environment operations, such as variable allocation and lookup, are represented as *transitional* nodes that will later be resolved to specific lexical stack frames during scope resolution. Overall, this stage transforms the high-level syntactic structure of JavaScript into a normalized, semantically explicit graph-based form, establishing the foundation for subsequent analysis, normalization, and optimization in IRIDIUM.

Example 1 (Implicit bindings). Figure 6 shows the transformation result of source JavaScript code into IRI. The source code is module mode JavaScript code, so the top-level scope introduces the `module.meta` and global `this` implicit bindings. Creation of an object is broken down into simpler constructs. First, an empty object is created via `JSObject()` and stored in a temporary variable `temp$1`. The computed property key `100*2` is evaluated using a `Binop` instruction, and its result is stored in `temp$2`. This result is then used in a `JSComputedFieldWrite` to set the property on the new object. The property `f` is added with a `FieldWrite` instruction, where the value is a lambda λ[BB=1] referencing the function's first basic block. The `console.log(a[0])` expression is deconstructed into a `FieldRead` to get the `log` method and a `CallSite` instruction to invoke it with the correct arguments. The function is extracted into a new basic-block context labeled as `ClosureBoundary` (`BB1`); here, the `arguments` and `this` implicit bindings are added for the function's scope. The function is not asynchronous, so the return is marked as `synchronous`. The call site is marked as *ContextualCall* which indicates that the first two arguments form the calling context. In this case the `console` object may be bound to the `this` object when the function is called. The top-level module code finally concludes with an `async` return.

Example 2 (Resolving control flow). High-level JavaScript constructs often expand into complex low-level control flows. Consider the standard `for-of` loop. Figure 7 shows the transformation result of iterated loops into IRIDIUM. While syntactically simple, this loop conceals several underlying operations, including implicit object conversions, the creation and management of an

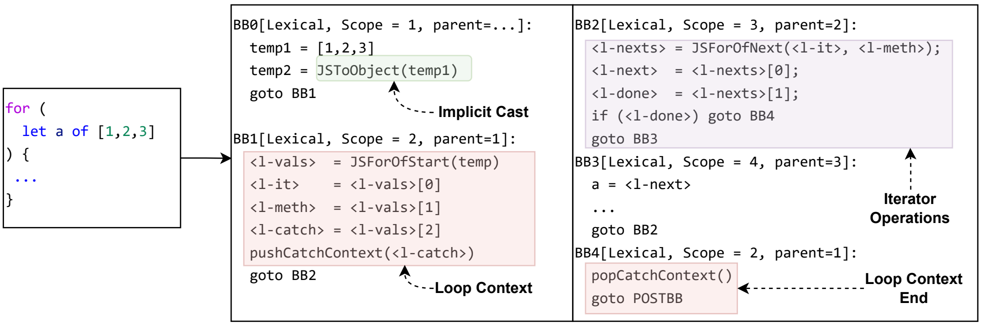


```
let a = {
  [100*2]: "Hello",
  f: function() { return "World" }
}

console.log(a[0])
```

```
BB0:

ImpBindingDecl[OPID = 6](
  UnresolvedRef("module.meta")
)
ImpBindingDecl[OPID = 9](
  UnresolvedRef("this")
)

ExpBindingDecl("temp$1") = JSObject()
ExpBindingDecl(
  UnresolvedRef("temp$2"),
  Binop[OP = "*"](
    Number(100),
    Number(2)
  )
)
JSComputedFieldWrite(
  UnresolvedRef("temp$1"),
  UnresolvedRef("temp$2"),
  String("Hello")
)
FieldWrite(
  UnresolvedRef("temp$1"),
  String("f"),
  λ[BB=1]
)
ExpBindingDecl(
  UnresolvedRef("a"),
  UnresolvedRef("temp$1"),
)
```

```
ExpBindingDecl(
  UnresolvedRef("temp$3"),
  FieldRead(
    UnresolvedRef("console"),
    String("log"),
  )
)
ExpBindingDecl(
  UnresolvedRef("temp$4"),
  JSComputedFieldRead(
    UnresolvedRef("a"),
    Number(0),
  )
)
CallSite[Contextual](
  UnresolvedRef("console"),
  UnresolvedRef("temp$3"),
  UnresolvedRef("temp$4"),
)
Return[async]
BB1[ClosureBoundary]:
ImpBindingDecl[OPID = 1](
  UnresolvedRef("arguments")
)
ImpBindingDecl[OPID = 9](
  UnresolvedRef("this")
)
 ExpBindingDecl(
  UnresolvedRef("temp$5"),
  String("World")
)
Return[sync]UnresolvedRef("temp$5")
```

Fig. 6. The transformation of a JavaScript snippet into IRI, detailing the breakdown of object creation, function extraction, and method calls into low-level instructions.



```
for (
  let a of [1,2,3]
) {
  ...
}
```

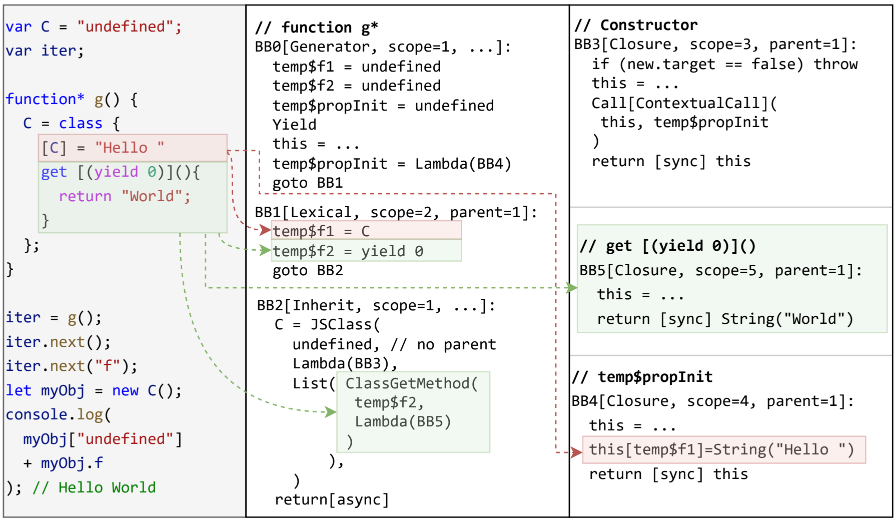
```
BB0[Lexical, Scope = 1, parent=...]:
  temp1 = [1,2,3]
  temp2 = JSToObject(temp1)
  goto BB1

BB1[Lexical, Scope = 2, parent=1]:
  <l-vals>  = JSForOfStart(temp)
  <l-it>    = <l-vals>[0]
  <l-meth>  = <l-vals>[1]
  <l-catch> = <l-vals>[2]
  pushCatchContext(<l-catch>)
  goto BB2
```

```
BB2[Lexical, Scope = 3, parent=2]:
  <l-nexts> = JSForOfNext(<l-it>, <l-meth>);
  <l-next>  = <l-nexts>[0];
  <l-done>  = <l-nexts>[1];
  if (<l-done>) goto BB4
  goto BB3
BB3[Lexical, Scope = 4, parent=3]:
  a = <l-next>
  ...
  goto BB2
BB4[Lexical, Scope = 2, parent=1]:
  popCatchContext()
  goto POSTBB
```

Fig. 7. Transformation of a for-of loop into IRI.

iterator, the setup of new evaluation scopes, and a try-catch block to ensure the iterator is properly closed. IRI deconstructs the loop into its fundamental control-flow graph, making these hidden mechanics explicit. In Figure 7, BB0 and BB1 handle the setup, including creating the iterator and a catch context, while the core iteration occurs in BB2, which checks if the loop is done, and in BB3 where the loop variable a is bound to the next value. Finally, BB4 performs the cleanup.

Example 3 (Desugaring high-level constructs). Figure 8 illustrates how a complex JavaScript class declaration is de-sugared into IRIDIUM. This process breaks down the high-level class syntax into simpler, more explicit computational steps involving closures, temporary variables, and basic



```
var C = "undefined";
var iter;

function* g() {
  C = class {
    [C] = "Hello "
    get [(yield 0)](){
      return "World";
    }
  };
}

iter = g();
iter.next();
iter.next("f");
let myObj = new C();
console.log(
  myObj["undefined"]
  + myObj.f
); // Hello World
```

```
// function g*
BB0[Generator, scope=1, ...]:
  temp$f1 = undefined
  temp$f2 = undefined
  temp$propInit = undefined
  Yield
  this = ...
  temp$propInit = Lambda(BB4)
  goto BB1

BB1[Lexical, scope=2, parent=1]:
  temp$f1 = C
  temp$f2 = yield 0
  goto BB2

BB2[Inherit, scope=1, ...]:
  C = JSClass(
    undefined, // no parent
    Lambda(BB3),
    List( ClassGetMethod(
            temp$f2,
            Lambda(BB5)
          )
        ),
    )
  return[async]
```

```
// Constructor
BB3[Closure, scope=3, parent=1]:
  if (new.target == false) throw
  this = ...
  Call[ContextualCall](
   this, temp$propInit
  )
  return [sync] this

// get [(yield 0)]()
BB5[Closure, scope=5, parent=1]:
  this = ...
  return [sync] String("World")

// temp$propInit
BB4[Closure, scope=4, parent=1]:
  this = ...
  this[temp$f1]=String("Hello ")
  return [sync] this
```

Fig. 8. Transformation of a JavaScript class into IRI.

control-flow blocks. The constructor logic is desugared into BB3, the instance-property initialization logic into BB4, and the body of the class method is in BB5. In addition, to manage the state required for dynamic class definitions, IRIDIUM introduces three key temporary variables in the generator's entry block (see BB0). temp$f1 stores the computed value of the instance property name (C), temp$f2 stores the computed value of the method name (yield 0), and temp$propInit holds a reference to the instance property initializer closure (BB4).

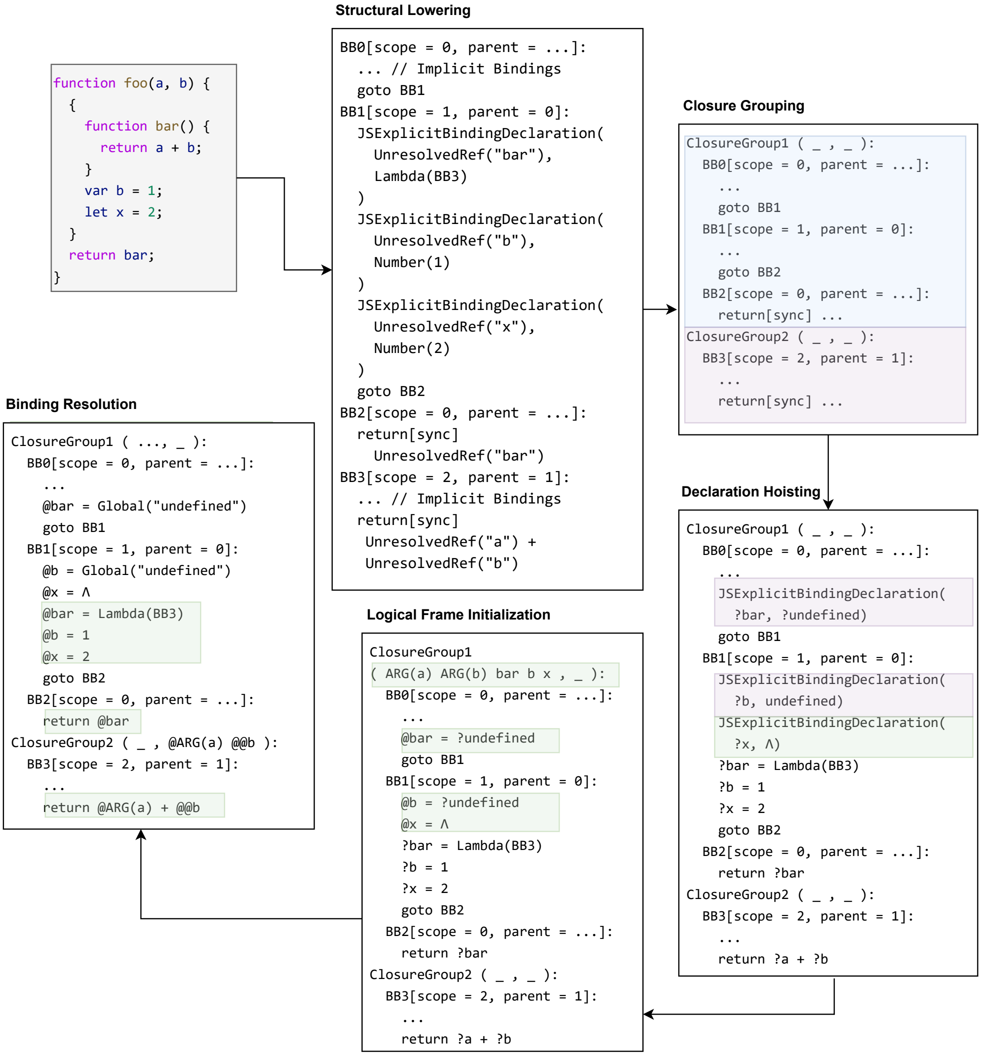
The control flow for the generator function g starts at BB0. The function initially yields after local bindings are initialized. Execution resumes and enters BB1 after the first call to next(), upon which temp$f1 is set to C. The execution then yields before initializing temp$f2, which gets its value after completion of the yield. Finally, the class is assembled using the JSClass node (in BB2). This structure makes a clear distinction between initializing properties on the instance and defining methods on the prototype. Methods are defined on the class prototype via the ClassGetMethod instruction within the JSClass node. In contrast, instance properties are set by the constructor (see BB3) which, after creating a new this object, makes a Call [ContextualCall] to the initializer logic stored in temp$propInit. The Call passes the newly created instance as the this context to the initializer closure (BB4), ensuring that the property is added directly onto the object instance.

5 NORMALIZATION TO IRI

The second stage of the IRIDIUM pipeline takes the preliminary TAC representation from the structural-lowering stage and subjects it to a series of core passes. These passes are crucial for resolving language-specific semantics, normalizing the IR to fully conform to the IRIDIUM specification. The major passes in this stage are listed below.

1. Closure Grouping. This pass groups the basic blocks into separate closure groups and initializes a blank logical stack frame which is used as the lookup environment.

2. Declaration Hoisting. This pass hoists variable and function declarations into their respective scope boundaries. let and const bindings are hoisted to the enclosing lexical scope while function and var declarations are hoisted to the enclosing closure boundary. If the declaration initializes the



```
function foo(a, b) {
  {
    function bar() {
      return a + b;
    }
    var b = 1;
    let x = 2;
  }
  return bar;
}
```

Structural Lowering

```
BB0[scope = 0, parent = ...]:
  ... // Implicit Bindings
  goto BB1
BB1[scope = 1, parent = 0]:
  JSExplicitBindingDeclaration(
    UnresolvedRef("bar"),
    Lambda(BB3)
  )
  JSExplicitBindingDeclaration(
    UnresolvedRef("b"),
    Number(1)
  )
  JSExplicitBindingDeclaration(
    UnresolvedRef("x"),
    Number(2)
  )
  goto BB2
BB2[scope = 0, parent = ...]:
  return[sync]
    UnresolvedRef("bar")
BB3[scope = 2, parent = 1]:
  ... // Implicit Bindings
  return[sync]
   UnresolvedRef("a") +
   UnresolvedRef("b")
```

Closure Grouping

```
ClosureGroup1 ( _ , _ ):
  BB0[scope = 0, parent = ...]:
    ...
    goto BB1
  BB1[scope = 1, parent = 0]:
    ...
    goto BB2
  BB2[scope = 0, parent = ...]:
    return[sync] ...
ClosureGroup2 ( _ , _ ):
  BB3[scope = 2, parent = 1]:
    ...
    return[sync] ...
```

Declaration Hoisting

```
ClosureGroup1 ( _ , _ ):
  BB0[scope = 0, parent = ...]:
    ...
    JSExplicitBindingDeclaration(
      ?bar, ?undefined)
    goto BB1
  BB1[scope = 1, parent = 0]:
    JSExplicitBindingDeclaration(
      ?b, undefined)
    JSExplicitBindingDeclaration(
      ?x, Λ)
    ?bar = Lambda(BB3)
    ?b = 1
    ?x = 2
    goto BB2
  BB2[scope = 0, parent = ...]:
    return ?bar
ClosureGroup2 ( _ , _ ):
  BB3[scope = 2, parent = 1]:
    ...
    return ?a + ?b
```

Logical Frame Initialization

```
ClosureGroup1
( ARG(a) ARG(b) bar b x , _ ):
  BB0[scope = 0, parent = ...]:
    ...
    @bar = ?undefined
    goto BB1
  BB1[scope = 1, parent = 0]:
    @b = ?undefined
    @x = Λ
    ?bar = Lambda(BB3)
    ?b = 1
    ?x = 2
    goto BB2
  BB2[scope = 0, parent = ...]:
    return ?bar
ClosureGroup2 ( _ , _ ):
  BB3[scope = 2, parent = 1]:
    ...
    return ?a + ?b
```

Binding Resolution

```
ClosureGroup1 ( ..., _ ):
  BB0[scope = 0, parent = ...]:
    ...
    @bar = Global("undefined")
    goto BB1
  BB1[scope = 1, parent = 0]:
    @b = Global("undefined")
    @x = Λ
    @bar = Lambda(BB3)
    @b = 1
    @x = 2
    goto BB2
  BB2[scope = 0, parent = ...]:
    return @bar
ClosureGroup2 ( _ , @ARG(a) @@b ):
  BB3[scope = 2, parent = 1]:
    ...
    return @ARG(a) + @@b
```

Fig. 9. Example to show the effect of normalization passes.

bindings, they are reduced down to simple environment-write operations and are marked as "safe" writes, indicating that they do not sit behind any write barriers.

3. Logical Frame Initialization. Arguments passed to the closure frame and bindings initialized within it are populated in the logical stack frame and all the transitional nodes are reduced into simpler write statements and get marked as safe.

4. Binding Resolution. This pass resolves all unresolved binding resolutions to their exact locations on local or remote stack frames.

Example. Figure 9 shows the effects of the aforementioned passes on the Intermediate Representation (IR) for a sample JavaScript program. The process begins with the initial IR from Stage 1, where bindings are represented as unresolved references (`UnresolvedRef`) and declarations are explicit nodes. The first pass, *Closure Grouping*, organizes the Basic Blocks (BBs) into logical units corresponding to function scopes. As shown, `BB0`, `BB1`, and `BB2` are grouped into `ClosureGroup1` for the outer function `foo`, while `BB3` forms `ClosureGroup2` for the inner function `bar`.

Next, *Declaration Hoisting* modifies the IR within these groups. In `ClosureGroup1`, the declaration for `var b` is hoisted and initialized with `undefined`, while the `let x` declaration is hoisted but remains uninitialized (`JSNUBD` refers to No-Use Before-Def semantics), reflecting its temporal dead zone. The function declaration for `bar` is also hoisted. The original initialization assignments are converted into `EnvWrite` operations, which are simple writes to the environment frame.

The *Logical Frame Initialization* pass then populates the abstract stack frame for each closure. `ClosureGroup1`'s frame is initialized with slots for its arguments (`ARG(a)`, `ARG(b)`) and its local bindings (`bar`, `b`, `x`). At this stage, the transitional declaration nodes are fully removed, and the frame slots are explicitly represented.

Finally, *Binding Resolution* resolves all symbolic references to their precise memory locations within the logical frames. All `UnresolvedRef` nodes are replaced with direct frame references. For instance, in `BB2`, return `UnresolvedRef("bar")` becomes return `@bar`. More importantly, this pass resolves captured variables in closures. In `ClosureGroup2`, the reference to a is resolved to `@ARG(a)` from the parent frame, and the reference to the outer `b` is resolved to `@@b`. The double-at symbol (`@@`) signifies that the binding is resolved to a slot in a remote (parent) stack frame, making the closure's data dependency explicit. This scheme simplifies optimizations like inlining by ensuring all bindings have a unique, resolved location.

At this stage, it is worth noting how IRI simplifies writing various analysis passes. IRI models logical stack frames explicitly, which enables scope pre-resolution and simplifies later analysis passes to a direct specification of transfer functions (see Section 6). Other IRs (such as SAFE [29], JSAI [15], and TAJS [14]) necessitate scope resolution as part of each analysis being specified using the framework, which makes different analyses not only difficult to write, but also error prone and slow. IRIDIUM solves this problem and makes the analysis writer free from complex JavaScript semantics (e.g. by making the reference to the binding `b` inside the closure `bar` unambiguous).

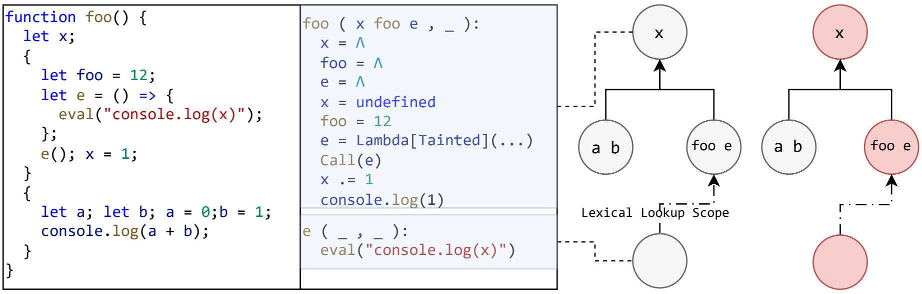
5.1 Auxiliary Transformations

Apart from the major transformation passes described earlier, the normalization stage also performs a range of auxiliary transformations that ensure JavaScript semantics are faithfully represented in IRIDIUM. These transformations address subtle behaviors unique to JavaScript and align them with the expectations of the target runtime.

1. Constructor semantics and super calls. JavaScript imposes fewer restrictions on constructor functions than class-based languages like Java. In particular, there is no syntactic requirement that `super()` must be the first statement in a derived class constructor. However, at run-time, the `super` call may only initialize the object once. To preserve this invariant, IRIDIUM decorates each `super` call to ensure single initialization. Subsequent `super` calls are recognized as illegal and can be flagged either statically or postponed to the runtime by generating explicit checks.

2. Private fields and brand checks. Private properties and methods in modern JavaScript classes are not stored on the object itself but instead live as lexical bindings in the surrounding class scope. Access to these members is protected by a run-time "brand check", ensuring that private state cannot be accessed from outside the declaring class. During this stage, IRIDIUM introduces explicit brand-check decorators to each private member access, thereby making the implicit access control semantics of JavaScript explicit in the intermediate representation.

3. Exception handling. JavaScript's structured exception handling allows control flow to exit a `try-catch-finally` construct via statements such as `break`, `continue`, or `return`. To preserve proper



```
function foo() {
  let x;
  {
    let foo = 12;
    let e = () => {
      eval("console.log(x)");
    };
    e(); x = 1;
  }
  {
    let a; let b; a = 0;b = 1;
    console.log(a + b);
  }
}
```

```
foo ( x foo e , _ ):
  x = Λ
  foo = Λ
  e = Λ
  x = undefined
  foo = 12
  e = Lambda[Tainted](...)
  Call(e)
  x .= 1
  console.log(1)

e ( _ , _ ):
  eval("console.log(x)")
```

Fig. 10. Handling of evals in IRIDIUM.

finalization semantics, IRIDIUM inserts explicit decorators that ensure catch contexts are correctly unwound and that finalizers execute before control leaves the protected region. This guarantees that even in non-local control transfers, cleanup semantics are preserved accurately.

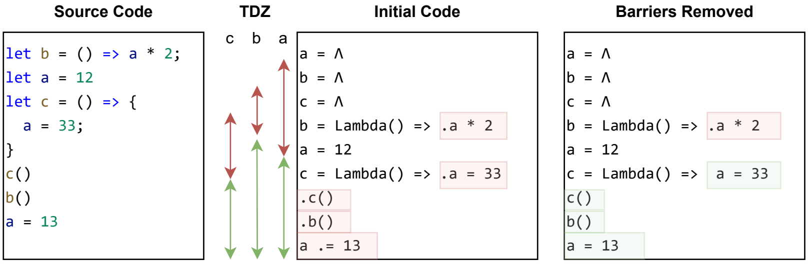
4. Module imports. Modules in JavaScript are declarative and must be fully resolved before execution begins. IRIDIUM parses each `import` declaration and synthesizes a corresponding *import metadata record*. The metadata includes the source module specifier, a list of imported bindings, and any exported names. This information allows the runtime to construct the correct dependency graph, ensuring that all required modules are loaded prior to initialization. When IRIDIUM is used in a bundling context, this metadata also guides cross-file linking and module composition.

5. Strict versus sloppy mode. The auxiliary transformation phase also normalizes the semantic differences between JavaScript's strict and sloppy modes. In sloppy mode, variable declarations (`var`) implicitly bind to the global object instead of the current lexical frame. IRIDIUM models this by lowering such declarations into property writes on the global `this` binding instead of true frame bindings. In contrast, strict mode enforces lexical scoping, so variable declarations are lowered into explicit frame bindings. This distinction ensures that the runtime can faithfully emulate JavaScript's dynamic scope resolution and error semantics.

Together, these auxiliary transformations enrich IRIDIUM's representation with precise semantic details that are often implicit or context-sensitive in JavaScript. They ensure that higher-level analyses and optimizations can operate on a uniform, well-defined model of program behavior.

5.2 Handling Eval

JavaScript's `eval` function presents a significant challenge for compilers. It can execute arbitrary code as a string, and crucially, with full access to the lexical scope in which it is called; further, there are two calling conventions for `eval`: *direct*, which allows unrestricted access to the executing frame; and *indirect*, which does not. This dynamic nature means the compiler cannot statically analyze what variables will be accessed, read, or modified by the `eval` call [18]. Consequently, as a variable that appears unused could be referenced within the eval string, optimizations like dead-code elimination and constant propagation become impossible. To handle this, IRIDIUM maintains the program's scopes in a tree structure. When it detects a call to `eval`, it "taints" the current scope and any parent scopes that `eval` can reach. Tainting a scope is a signal to static analyses that variables within it cannot be safely optimized away. As illustrated in Figure 10, `eval("console.log(x)")` inside function `e` taints the lexical scope in `foo` where bindings `foo`, `e` and `x` are defined (bindings `a` and `b` remain untainted and can be optimized away).



```
let b = () => a * 2;
let a = 12
let c = () => {
  a = 33;
}
c()
b()
a = 13
```

```
a = Λ
b = Λ
c = Λ
b = Lambda() => .a * 2
a = 12
c = Lambda() => .a = 33
.c()
.b()
a .= 13
```

```
a = Λ
b = Λ
c = Λ
b = Lambda() => .a * 2
a = 12
c = Lambda() =>  a = 33
c()
b()
a = 13
```

Fig. 11. Temporal Dead Zone (TDZ) Analysis enabling removal of redundant read/write barriers. Red arrows indicate code regions where a variable might be in its TDZ, and green ones indicate safe regions.

6 ANALYSIS, OPTIMIZATION AND CODE GENERATION

We have implemented several analysis passes in IRIDIUM, enabling both conventional and JavaScript-specific optimizations. While one could always implement more, our goal here is to demonstrate the kinds of static analyses that are made feasible by IRIDIUM's explicit intermediate representation. They also serve to validate that our infrastructure can support realistic optimization pipelines capable of improving JavaScript performance through static reasoning.

Most analyses and optimizations currently focus on bindings that neither belong to *tainted scopes* nor are captured by closures, as those require more precise interprocedural modeling. Among the standard analyses implemented are *Constant Propagation*, which tracks statically known constant values for bindings, with the lattice {NAC, String, Number, Null, Boolean, JSBigInt}, allowing constant folding and elimination of redundant computations; *Copy Propagation*, which tracks equivalent bindings where one variable directly aliases another, thereby reducing unnecessary indirections introduced by lexical transformations or transpilation; and *Liveness Analysis*, a classic backward dataflow analysis that determines which variables may be used before being redefined, preserving only live bindings while enabling later elimination of dead ones.

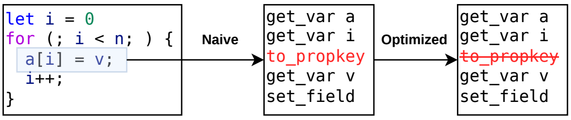
We now describe three JavaScript-specific optimizations that we have implemented in IRIDIUM, as examples of optimizations made feasible by IRI. The dataflow equations of the analyses that enable these optimizations are described in Appendix C of the supplementary material.

1. Barrier Elimination. Lexical bindings in JavaScript sit behind *read* and *write barriers*. This is because JavaScript enforces *Temporal Dead Zone* (TDZ) semantics: a variable declared with `let` or `const` cannot be accessed before its declaration is executed. As the runtime cannot always statically determine whether a binding is initialized before use, JIT engines and interpreters often emit runtime TDZ checks around every lexical access. These barriers ensure correctness but introduce substantial overhead. To safely eliminate these barriers, IRIDIUM performs a dedicated static analysis pass called the Temporal Dead Zone Analysis (TDZA). TDZA is a *forward dataflow analysis* that computes, for each program point, whether each lexical binding is:

$$\mathsf{TDZLatticeVal} = \{\,\mathsf{TDZ},\ \mathsf{SAFE},\ \bot\,\}$$

Here, TDZ indicates that the binding is in its temporal dead zone (access would be illegal); SAFE indicates that the binding is definitely initialized and can be accessed safely; and ⊥ indicates that the state is unknown or uninitialized (entry condition).

During analysis, each variable transitions from ⊥ to TDZ upon declaration, and to SAFE upon its first definite assignment. Figure 11 illustrates TDZA applied to a simple program. In the initial source, `b` is declared before `a` but references it within its body (`a * 2`), which would normally trigger a runtime TDZ check when `b()` is invoked. If the function is called before `a` is initialized, this check



```
let i = 0
for (; i < n; ) {
  a[i] = v;
  i++;
}
```

```
get_var a
get_var i
to_propkey
get_var v
set_field
```

```
get_var a
get_var i
~~to_propkey~~
get_var v
set_field
```

Fig. 12. Redundant cast removal identifying unnecessary to_propkey operations in array access expressions.

correctly raises a `ReferenceError`. However, `c` is initialized only after `a` has safely escaped its TDZ, so any invocation of `c` can safely skip the TDZ check. Similarly, all subsequent accesses to `a`, `b`, and `c` after `c`'s declaration are considered safe, and their barriers can be removed.

The TDZ analysis is particularly valuable because eliminating barrier checks also makes traditional analyses, such as liveness, more effective. Without barrier elimination, writes that would normally render a variable dead might appear live due to the synthetic reads introduced by TDZ guards. By statically proving the safety of bindings, IRIDIUM can simplify control flow, improve dataflow precision, and safely remove redundant checks that a conventional interpreter would otherwise necessarily perform at run-time.

2. Redundant Cast Removal. When operating on computed fields in JavaScript, property keys are implicitly cast to property key types through the internal `ToPropertyKey` operation. In IRIDIUM, computed field operations are by default marked as unsafe, i.e. a cast if required. However, these casts are often redundant and can be safely eliminated when the key expression is provably of a type that already satisfies the requirements of `ToPropertyKey`. For example, numeric indices or string literals used in property accesses do not require an additional cast, yet typical transpilers and interpreters conservatively emit them.

To identify such opportunities, IRIDIUM performs a forward dataflow analysis pass that tracks the type safety of property key expressions. The analysis is defined over the lattice:

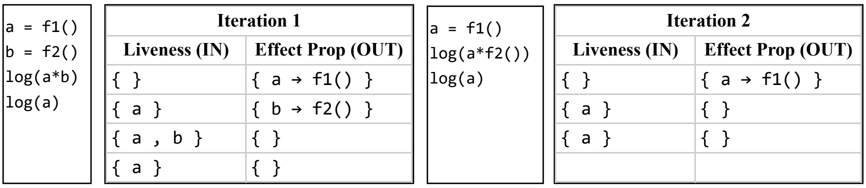
$$\mathsf{SafeKeyLatticeVal} = \{\,\mathsf{SAFE},\ \mathsf{UNSAFE},\ \bot\,\}$$

Here, SAFE indicates that the key expression is guaranteed to be a valid property key (string or number); UNSAFE indicates that the key expression may not be a valid property key and requires a cast; and $\bot$ indicates that the state is uninitialized or unknown.

This pass effectively models the flow of property key values across expressions, propagating SAFE whenever the operand is known to be a literal, numeric index, or derived from a prior SAFE expression. At merge points, the standard meet operation ensures safety only if all incoming paths are SAFE. Any UNSAFE or unknown input reintroduces the conservative cast.

Figure 12 illustrates the redundant cast removal optimization. In the source code, the loop assigns elements to an array `a[i] = v`. During lowering, the key expression `i` is conservatively wrapped in a to_propkey cast to ensure correctness across all possible runtime types. However, the analysis proves that `i` is always a numeric binding incremented within the loop, and therefore inherently a valid property key. The redundant coercion is safely removed, as shown in the optimized bytecode on the right, where `get_var i` flows directly into `set_field` without the intermediate cast.

3. Effect Propagation. During bytecode emission, specialized bytecodes are often used for common access patterns. For instance, the `get_field` operation expects two values on the stack—the receiver object and the field key—and produces the retrieved value as the result. To avoid redundant stack manipulations, bytecode emitters employ peephole optimizations that fuse common access sequences. For example, when emitting bytecode for expressions like `console.log(a,b,c)`, the emitter may use specialized instructions (say `get_field2`), which preserve the receiver context on the stack instead of repeatedly pushing and popping it. However, once source code is lowered



```
a = f1()
b = f2()
log(a*b)
log(a)
```

Iteration 1	
Liveness (IN)	**Effect Prop (OUT)**
{ }	{ a → f1() }
{ a }	{ b → f2() }
{ a , b }	{ }
{ a }	{ }

```
a = f1()
log(a*f2())
log(a)
```

Iteration 2	
Liveness (IN)	**Effect Prop (OUT)**
{ }	{ a → f1() }
{ a }	{ }
{ a }	{ }

Fig. 13. Effect propagation removing redundant temporaries and inlining effectful expressions.

into three-address code (TAC), these localized optimizations become harder to recover, as new temporaries obscure the relationship between effects and their uses.

To reestablish these relationships, IRIDIUM performs a static optimization called *effect propagation.* This transformation is powered by the EffectAtStmt analysis—a forward dataflow analysis that tracks, at each program point, the most recent side-effect-producing operation that remains valid. Unlike constant or copy propagation, which reason only about side-effect-free values, EffectAtStmt maintains a single abstract summary describing whether a valid, effectful computation is currently active in the flow of execution. Conceptually, it attempts to answer the question: *"Is the previous store still valid, or has it been invalidated by an effectful statement?"*

Formally, the transformation propagates an effectful computation into another statement if and only if the following predicates hold: (i) the effect is currently *available*, as indicated by the EffectAtStmt analysis; (ii) the *target binding is dead* after the statement, according to the Liveness lattice; and (iii) there are *no intervening side effects* between the effect's evaluation point and its intended propagation site—i.e., the expression subtree being rewritten is side-effect free. If all three conditions are satisfied, the effect can be safely be substituted in place of the binding, and the original statement that computed it can be removed. This enables the collapse of redundant temporaries and restores locality to effectful computations without altering observable behavior.

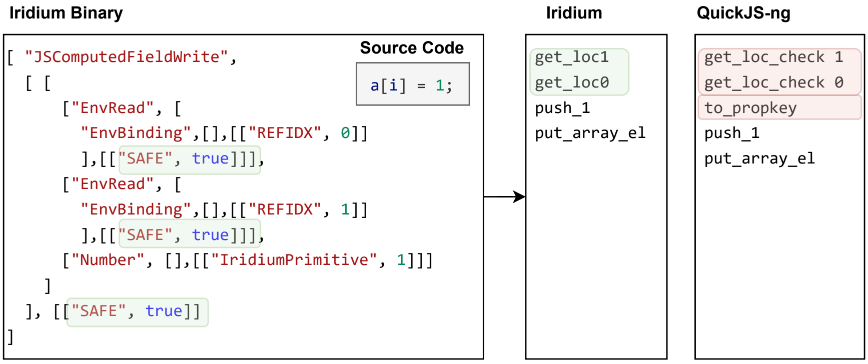
Figure 13 illustrates this optimization. In the initial program, `a = f1()` and `b = f2()` produce two effectful results that are later used in `log(a * b)`. The first iteration of analysis detects that `b`'s definition (`f2()`) is an available effect, that `b` is dead after the multiplication, and that no intervening effects occur inside the expression tree. Hence, the effect from `f2()` can be safely propagated, yielding `log(a * f2())`. Conversely, `a` cannot be propagated because it remains live after the statement (it is read again by the subsequent `log(a)`), thus violating the liveness condition.

The pass manager executes effect propagation as the last optimization stage in the pipeline, just before generating executable code. This preserves full analyzability during earlier analysis and optimization phases, as the TAC form keeps side effects represented explicitly. Once all higher-level analyses are complete, this pass restores compact, stack-like effect locality that benefits downstream code generation and enables further low-level peephole optimizations in the back-end.

Note that in this section, we have only described a subset of the passes implemented in IRIDIUM. IRI's explicitization of complex control structures, in-built support for asynchronous primitives, and ability to generate executable code have allowed us implement several more dataflow and peephole optimizations, and opened up venues to implement many more useful passes in the future (such as type and pointer analysis [14, 20], shape analysis [30], taint analysis [3], and so on).

6.1 Instruction Selection

As our first backend and a testbed for this paper, we chose QuickJS(-NG) [7], a lightweight, JIT-less JavaScript VM used in embedded systems and Amazon's LLRT runtime [4]. We chose QuickJS for its modular design and ease of integration, which makes it an ideal target for early-stage compiler



```
a[i] = 1;
```

```
[ "JSComputedFieldWrite",
  [ [
      ["EnvRead", [
        "EnvBinding",[],[["REFIDX", 0]]
        ],[["SAFE", true]]],
      ["EnvRead", [
        "EnvBinding",[],[["REFIDX", 1]]
        ],[["SAFE", true]]],
      ["Number", [],[["IridiumPrimitive", 1]]]
    ]
  ], [["SAFE", true]]
]
```

```
get_loc1
get_loc0
push_1
put_array_el
```

```
get_loc_check 1
get_loc_check 0
to_propkey
push_1
put_array_el
```

Fig. 14. Example of instruction selection for the QuickJS backend.

research. IRIDIUM currently emits executables in a JSON-based intermediate format containing complete structural and semantic information. While not space-efficient, this format simplifies debugging and rapid iteration, and can be trivially migrated to a compact binary representation in the future. We achieve integration with QuickJS via a small loader built on cJSON [10]. The loader reconstructs the IRIDIUM program representation (IRIDIUM uses a Lisp inspired S-Expression format) and runs an *instruction-selection* pass that maps intermediate operations to native QuickJS bytecode. While doing so, we take a non-invasive approach: the VM's execution core, garbage collector, and runtime libraries remain untouched. This preserves compatibility with upstream releases while allowing controlled experimentation with instruction lowering and effect modeling. Future work will extend this interface to additional backends, such as Google's V8, positioning IRIDIUM as a portable compilation and analysis framework across JavaScript runtimes.

Figure 14 illustrates the translation of `a[i] = 1` from IRIDIUM executable into QuickJS Bytecode. Notice how the IRIDIUM bytecode sequence removed redundant guards and coercions using the SAFE flag, thus allowing better instruction selection.

6.2 Translation Correctness

Recall that IRIDIUM converts ECMAScript 2025 [1] code to QuickJS bytecode in three steps: translation to 3JS, conversion to IRI, (optional) optimizations over IRI, and then target bytecode generation. We check the correctness of translation in three ways:

(i) Differential testing (for 3JS and IRI) over the official ECMA Test262 test suite. 3JS is 99.7% compliant with respect to V8 (which has the highest compliance among popular JavaScript runtimes), while IRI is about 90% compliant with respect to QuickJS, our target back-end.

(ii) While generating IRI, if we find a syntactic construct that we do not support, we throw a compile-time error and do not proceed to code generation.

(iii) Finally, we check the behavior of the generated bytecode by executing it using QuickJS. Most of our benchmark suites consist of a verify function that throws an error on unexpected behavior. We found that all the benchmarks part of the popular suites that we consider pass this test and lead to the same observable behavior as the default QuickJS baseline.

Many of the yet-unsupported features (see Appendix E for details) are simple engineering extensions left just because of nuanced niche usecases (such as lookup logic for eval when called in argument-initialization scope), and can be implemented in the future without changing IRI's semantics. On the other hand, we do not plan to support deprecated features such as with.

7 EVALUATION

The goal of our evaluation is to assess both the performance and the quality of code generated by the IRIDIUM framework. Recall that our current prototype implements a set of static-analysis–driven optimizations and a simple instruction-selection pass. Hence through this evaluation, we aim to observe the end-to-end behaviour of our current implementation, quantify the benefits it provides, and identify future scopes of improvement. Specifically, we address the following research questions:

RQ1: How does the performance of programs generated by IRIDIUM compare to those executed under QuickJS natively?

RQ2: Does IRIDIUM successfully eliminate typical overheads introduced due to JavaScript-to-3JS transpilation?

RQ3: What is the quality of IRIDIUM-generated code, particularly in terms of removing redundant guards and casts, and enabling specialized instruction selection?

7.1 Experimental Setup

We use the following benchmarks from the widely adopted JetStream 2 suite [2]:

- SunSpider — a collection of microbenchmarks focusing on core language features such as arithmetic, string manipulation, and control flow.
- Kraken — medium to large workloads including audio processing, physics simulations, and cryptographic kernels.
- Octane2 (RexBench) — a large-scale benchmark emphasizing dynamic language features and regular expression handling.
- Gaussian Blur (SeaMonster) — includes image processing and numerical computation workloads such as Gaussian blur and edge detection.
- ML Load (ARES-6) — a machine-learning and numerical computation benchmark that stress modern optimization pipelines.
- UniPoker (RexBench) — a large-scale simulation benchmark modeling probabilistic decision-making and object-heavy workloads.

For space, we selected 16 representative JavaScript programs spanning the above sub-suites to ensure a balanced mix of microbenchmarks, medium-sized simulations, and modern large-scale workloads that do machine learning and graphics processing.

Each benchmark is compiled and executed under three configurations: (i) baseline default QuickJS (QJS+SRC); (ii) baseline QuickJS running transpiled 3JS code (QJS+3JS); and (iii) augmented QuickJS that processes IRIDIUM-generated code. We test in the second configuration to induce a fairer comparison of IRIDIUM with baseline QuickJS, in context of the usual practice of transpiling JavaScript code through a parser such as Babel. All experiments were performed on an isolated Ubuntu 22.04 docker container on a system with two AMD EPYC 7713 64-core processors and 1TB of memory. We used LMA [22] to isolate 64 cores to the container with memory randomization disabled to reduce variance. We first compile all the benchmarks into IRIDIUM executables using our pipeline and compare their execution times with the reference QuickJS implementation. Each experiment was repeated 15 times, and mean values are reported with interquartile ranges.

7.2 Performance Evaluation

Figure 15 shows the execution times for the benchmarks under consideration, in the three modes described above. As can be seen, IRIDIUM improves performance noticeably (on average, 1.18×) for most of the benchmarks, across different sizes and categories. For some of the large programs (such as *Beat Detect*, *Gaussian Blur*, *G-Blur* and *UniPoker*), we notice significant improvements that run in order of seconds. We found that these programs make significant usage of guarded

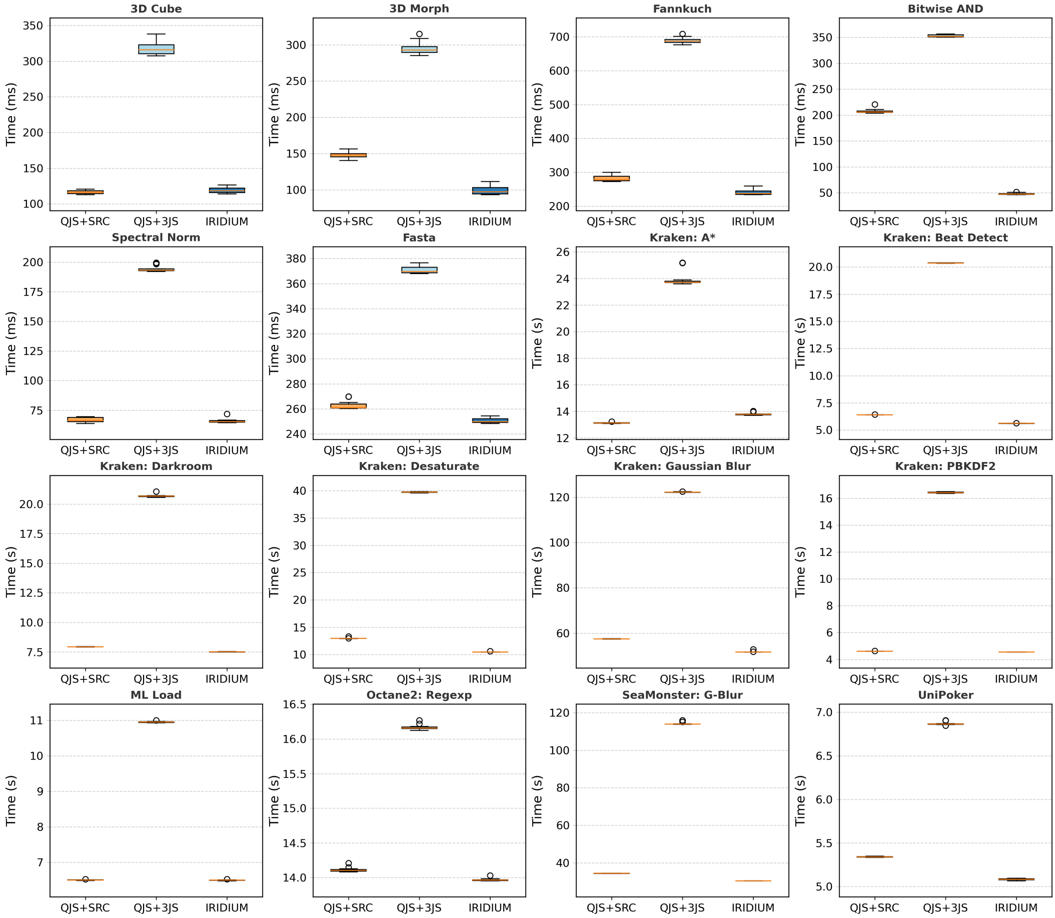


Fig. 15. Performance comparison amongst various QuickJS configurations. Lower the better.

stack operations and implicit coercions, which get eliminated by IRIDIUM's static optimization passes. On the other hand, we also notice improvements for microbenchmarks from the SunSpider suite (such as *3D Morph*, *Fannkuch* and *Bitwise AND*) that use language features not particularly targeted by IRIDIUM, indicating they had redundancies that could be eliminated using classic control-flow optimizations. We do notice minor regressions (e.g. the program *A**), which arise out of unoptimized instruction selection (studied further in Section 7.3).

Reckon that QuickJS by default converts JavaScript source code directly to its bytecode (instead of going through an easy-to-transpile subset like 3JS), which may allow it to take advantage of source-code structure and generate more efficient bytecode patterns. Hence, to make a more neutral comparison, the middle plot in each plot shows the execution time if QuickJS was provided 3JS code for each benchmark as input. The results here are quite encouraging: we see that IRIDIUM outperforms QuickJS by huge margins, for all the benchmarks. This shows the significant improvements imparted by static analysis and optimizations (just the ones we currently have implemented).

Overall, we find that IRIDIUM-generated code was able to not only alleviate the transpilation overheads but also managed to outperform the existing system. Our implementation was on average 2.5× faster than transpiled code (QJS+3JS) and 1.18× faster than the reference implementation

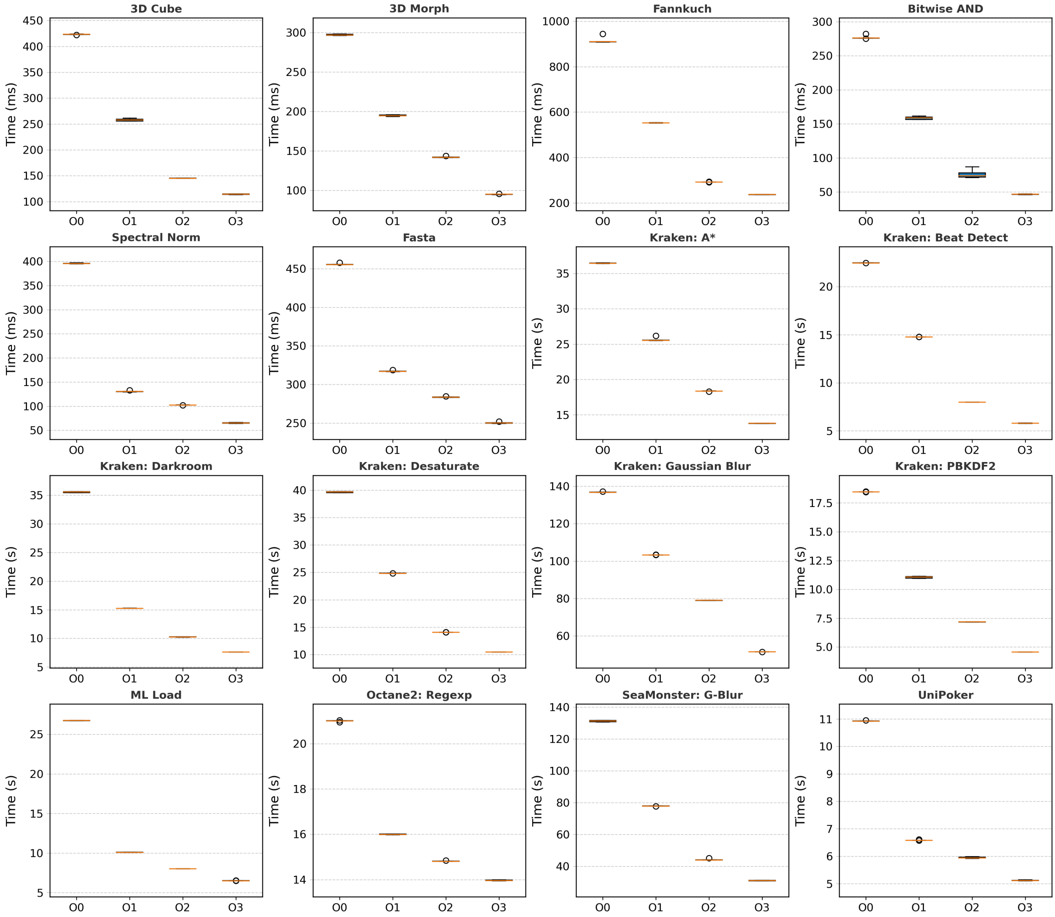
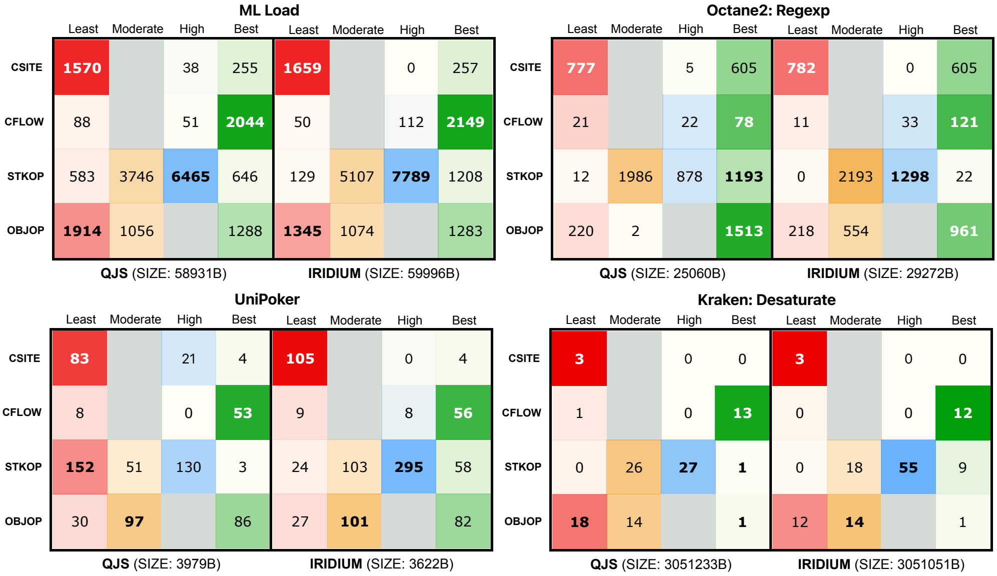


Fig. 16. Performance under different optimization levels (O0-O3). Lower the better.

(QJS+SRC). In the next sections, we perform detailed investigations that allow us to find where exactly does IRIDIUM improve, and where all do we have scope to improve further.

7.2.1 Ablation study. To understand the impact of various optimizations on the benchmarks, we measured the performance by enabling them sequentially on top of each other. We made the following four categories of optimizations for this study: (i) O0, no optimizations; (ii) O1, classic compiler optimizations (excludes JavaScript-specific optimizations); (iii) O2, high-level JavaScript optimizations such as write-barrier elimination on top of O1; and (iv) O3, low-level optimizations such as effect propagation that helps instruction selection, on top of O2; see Figure 16.

As we can see, each optimization level increases the performance incrementally, for all the benchmarks under consideration. Classic compiler optimizations in O1 yield significant improvement by allowing the removal of obvious redundancies introduced during transpilation. Next, write-barrier elimination in O2 enables precise liveness analysis, which in turn improves the impact of classic optimizations. Finally, effect propagation in O3 allows better instruction selection. These results indicate the potential of static optimizations for JavaScript programs, and we plan to add more optimizations in each bucket as IRIDIUM matures further.



ML Load

	Least	Moderate	High	Best	Least	Moderate	High	Best
CSITE	**1570**		38	255	**1659**		0	257
CFLOW	88		51	**2044**	50		112	**2149**
STKOP	583	3746	**6465**	646	129	5107	**7789**	1208
OBJOP	**1914**	1056		1288	**1345**	1074		1283
	QJS (SIZE: 58931B)				**IRIDIUM** (SIZE: 59996B)			

Octane2: Regexp

	Least	Moderate	High	Best	Least	Moderate	High	Best
CSITE	**777**		5	605	**782**		0	605
CFLOW	21		22	**78**	11		33	**121**
STKOP	12	1986	878	**1193**	0	2193	**1298**	22
OBJOP	220	2		**1513**	218	554		**961**
	QJS (SIZE: 25060B)				**IRIDIUM** (SIZE: 29272B)			

UniPoker

	Least	Moderate	High	Best	Least	Moderate	High	Best
CSITE	**83**		21	4	**105**		0	4
CFLOW	8		0	**53**	9		8	**56**
STKOP	**152**	51	130	3	24	103	**295**	58
OBJOP	30	**97**		86	27	**101**		82
	QJS (SIZE: 3979B)				**IRIDIUM** (SIZE: 3622B)			

Kraken: Desaturate

	Least	Moderate	High	Best	Least	Moderate	High	Best
CSITE	**3**		0	0	**3**		0	0
CFLOW	1		0	**13**	0		0	**12**
STKOP	0	26	**27**	**1**	0	18	**55**	9
OBJOP	**18**	14		**1**	12	**14**		1
	QJS (SIZE: 3051233B)				**IRIDIUM** (SIZE: 3051051B)			

Fig. 17. Bytecode profiles of *ML Load*, *Regexp*, *UniPoker* and *Desaturate*. For each program, larger numbers toward the right-hand side indicate more performant instructions in the corresponding category.

7.3 Qualitative Analysis

To assess the quality of the code generated by IRIDIUM vis-a-vis the default QuickJS implementation (QJS+SRC), we examined the bytecode profiles of four large programs drawn from our benchmark suite. We first classified the emitted bytecodes into four operational categories: (i) CSITE representing call sites; (ii) CFLOW representing control-flow; (iii) STKOP representing stack operations; and (iv) OBJOP representing object operations. We then grouped these operations into four optimization tiers—*least optimized*, *moderately optimized*, *highly optimized*, and *most optimized*. See Appendix D for the full classification. As an example, there are multiple bytecode instructions to represent control-flow jumps: OP_goto (5 bytes), OP_goto16 (3 bytes) and OP_goto8 (2 bytes). Here, based on the size, we classify these three bytecodes into least optimized, highly optimized and most optimized, respectively. Figure 17 shows the classification of all bytecodes across these categories for the benchmarks under consideration.

Recall that our analyses and optimizations primarily target two inefficiencies: redundant barriers (on lexical binding accesses) and redundant casts (on computed object keys). For barriers (STKOP), we observe significant reductions in *least optimized* instructions (e.g. from 583 to 129 for *ML Load*). We also observe a significant shift to more optimized instructions (e.g. from 130 to 295 for *Unipoker*). We also see a good improvement in the profile for casts (OBJOP), e.g. a reduction from 1914 to 1345 for *ML Load*; in Section 7.3.1, we study if this could be improved even further. These figures confirm that the analysis and optimization passes of IRIDIUM are achieving their intended effects.

More interestingly, this study reveals the performance headroom that remains. Our current instruction selection pass is relatively naive—it mostly performs a direct translation from IRI to target bytecode. This limitation is especially visible in *Regexp*, where redundant barriers (see STKOP) are successfully eliminated, yet the pass fails to select the most optimal instruction sequence; for instance, a contiguous sequence of put and get operations on the same binding could be compacted into a single `set` instruction that retains the top-of-stack object. In the case of *Desaturate*, we observe that even though there are no barriers to eliminate, the overall number of stack operations increases, which indicates that there are temporaries introduced during compilation that could not be removed due to the presence of tainted scopes or conservative static analysis. Another interesting observation in *Desaturate* is the reduction in overall number of CFLOW instructions, including the removal of an unoptimized instruction in the process (which came from a large goto-offset). We also found that the QuickJS ISA includes a special `OP_return_undef` instruction, which appears in the *highly optimized* counts of *ML Load*, *Regexp* and *UniPoker*. This could be easily implemented as a peephole optimization in IRIDIUM's instruction selection phase in the future.

On one hand these results suggest that there is ample scope for optimizing the bytecode generated by IRIDIUM through simple peephole optimizations, while on the other hand we may also be able to design generalized analyses and optimizations that subsume such peephole optimizations altogether, because IRI exposes stack operations as first-class primitives.

7.3.1 Missed opportunities. Although our static analysis successfully identifies many instances where redundant property key conversions could be avoided, the QuickJS backend sadly does not allow us to realize the full potential of our analysis results. As an instance, we found that when a computed field (`a[x] = ...`) is being written to, the QuickJS VM emits the `OP_to_propkey` instruction to cast the value obtained upon evaluation of `x` into a valid type for object's field reference. This explicit check can be removed if the cast is identified as redundant. However, when such computed fields are being read from, no such explicit instruction is emitted by QuickJS, and instead the check is baked into the `OP_get_array_el` instruction. So this means even though we know statically that the cast is redundant, it cannot be avoided due to the ISA limitation of QuickJS.

Upon measuring such missed opportunities, we found that in total we were only making use of 17% of the opportunities identified by our analyses. This result, while shocking, makes sense; for instance, large arrays may be created once and be referenced multiple times later on. This limitation may be elided by changing or extending the ISA, but our current implementation intentionally preserves the reference VM unchanged, serving solely as an auxiliary optimization layer. Nevertheless, these observations tell that higher gains may be achievable when we extend our implementation to target new back-ends that offer more flexible or extensible instruction sets.

7.4 Additional Studies

7.4.1 Effect on microbenchmarks. To closely evaluate the impact of IRIDIUM's analysis and optimization passes from Section 6, we designed simple microbenchmarks to trigger the associated optimizations; see Figure 18. The first microbenchmark (function `microbm1`) performs writes to closure-captured variables `a`-`d`. Our analysis eliminates the associated *write barriers*, resulting in a measured speedup of approximately 1.5×. The second microbenchmark (function `microbm2`) writes to an array stored in `a`. Here, the analysis identifies a *redundant cast* while writing to `a[i]`, as the iterator variable `i` is guaranteed to be an integer, producing a speedup of 2.3×. In the third microbenchmark (function `microbm3`), a new binding `b` is introduced inside the `for` loop, which can be propagated through *effect propagation*. Applying this optimization yields a speedup of 1.4×.

7.4.2 Potential for JIT compilers. We studied the optimizations performed by Mozilla's SpiderMonkey [27] runtime, and found that its JIT compiler also supports removal of TDZ checks. However, we observed that for our microbenchmark `microbm1` (Figure 18), the SpiderMonkey JIT is unable to remove any TDZ check (because, naturally, it lacks analysis capability beyond the compilation scope), while IRIDIUM eliminates the associated write barriers. To measure the impact, we modified the `emitTDZCheckIfNeeded` routine in SpiderMonkey to manually elide these barriers, and found that the performance improved by 2×, which is even more than the improvement observed

```
function microbm1() {
   let a = 12, b = 13, c = 14, d = 15;
   let f = (x) => {
      a = x; b = x + 1;
      c = x + 2; d = x + 5; };
   for (let i = 0; i < 1e7; i++)
      f(i);
}
```

```
function microbm2() {
   let a = [];
   for (let i = 0; i < 1e7; i++)
      a[i] = i;
}
```

```
function microbm3() {
   let a = 12, b = 13, c = 14, d = 15;
   let f = (x) => {
      a = x; b = x + 1;
      c = x + 2; d = x + 5; return d; };
   let g = (x) => {
      a = x; b = x + 1;
      c = x + 2; d = x + 5; return d; };
   let h = (x) => { };
   for (let i = 0; i < 1e7; i++) {
      let a = f(i); let b = g(i);
      h(a * b); h(a);
   }
}
```

Fig. 18. Three microbenchmarks to show scenarios where IRIDIUM's optimization passes work well.

Repository	**Safe**	**Total**	**Safe %**
FRONTEND_VIZ	60	261	23.0
ORIGAMI_SIMULATOR	1718	2477	69.4
QS	682	1378	49.5
REACT_MOTION	290	828	35.0
TOTAL	**2750**	**4944**	**55.6**

Fig. 19. Potential TDZ-check removal in SpiderMonkey.

previously for QuickJS. Motivated by this observation, we took a few modern JavaScript programs from GitHub that heavily use lexically captured let/const-bindings [11, 13, 19, 33] (our benchmarks are unfortunately older-styled and use var-bindings), and counted the number of interprocedural barrier-removal opportunities identified by IRIDIUM statically. see Figure 19 where "Safe" indicates a barrier that can be removed.

Out of the total number of barriers (order of thousands), we found that 23-69% were identified as removable (many of them in hot functions such as those that track mouse movement). This indicates that there likely are a plethora of opportunities in traditional JITted systems that could benefit (significantly) through an approach like IRIDIUM's. We believe this shows the promise of a future IRIDIUM+JIT optimization system, and intend to facilitate the same with further research.

8 RELATED WORK

Analyzing JavaScript programs is known to be extremely challenging, because of several features that are difficult to model statically [35]. Consequently, various attempts to analyze JavaScript programs struggle to satisfy the triad goals of soundness, precision and scalability, and end up sacrificing on at least one of them. In this section, we review some of the representative works that have tried to solve one or more of these problems.

The widest undertakings in JavaScript static-analysis research have been the TAJS [14] and the JSAI [15] frameworks. JSAI, though not being maintained actively, was one of the first attempts to formally model a concrete subset of JavaScript, and has been the basis for much of future research in the area. TAJS, rechristened recently as Jelly [26], aims to perform static type analysis of JavaScript programs and has been used to produce call graphs and to write security analyses, primarily as an aid to programmers in IDEs, for several years. Jelly supports flow-insensitive pointer analysis based call-graph construction, and is used for detecting security vulnerabilities in JavaScript and TypeScript programs. With a rich body of work, these tools have been extended to model several

JavaScript features; however, they are known to be unsound in their results, which they can afford because they do not optimize code or generate backend executables. Our framework IRIDIUM, on the other hand, is designed with the aim of static optimization for JavaScript programs, and already supports optimized bytecode-generation for a reasonably popular JavaScript backend, QuickJS.

The most popular approach of improving the precision of JavaScript static analysis has been to feed to it the results of targeted dynamic analyses. Dolby et al. [36] first showed, using localized dynamic traces, that the root causes of imprecision could be attributed to a small percentage of functions. More recently, Møller et al. [17] use the idea of dynamic tracing to record actual properties at sites with dynamic property accesses, and propagate these to the static analyzer for more precise call-graph construction. Ko et al. [16] determine important execution paths to tune the precision of static analysis for relevant parts of the program. Finally, to ease the engineering of such hybrid analyses, Park et al. [28] present a framework that allows a static and a dynamic analyzer to communicate with each other and produce precise information as a result. As discussed in this paper, our goal via IRIDIUM is to test the limits of static (and static+dynamic) optimization, as a result of which we can only mark all these works as complementary and not as tools that can be used to generate correct-by-requirement code for JavaScript backends.

Our closest related work is a recent attempt to design an AOT toolchain (called Hopc) that transpiles JavaScript programs to Scheme [32]. Similar to us, it restricts itself to parts of JavaScript that do not require lazy binding, but contrary to us, it cannot integrate with existing JavaScript runtimes. Nevertheless, we believe our research stands on the observations made by Hopc: that it is worth trying to optimize programs written in dynamic languages like JavaScript statically. Over time, we hope IRIDIUM would serve its goal of providing a novel middle layer that can be extended to provide AOT optimizations for multiple JavaScript runtimes.

A promise for our approach comes from recent literature on speculative optimizations that have been enabled with a combination of static and dynamic analyses, with options to fall back (or deoptimize) during run-time in case of the speculation going wrong. A recent work [5] utilizes this idea for allocating Java objects on stack: it performs a best-effort analysis statically, allocates determined objects on stack in a JVM, and uses run-time checks to move incorrect stack allocations to the heap. Another recent work [31] generates optimized code (interestingly, for QuickJS itself) for class-based JavaScript programs, and proposes the idea of despecialization to perform dynamic rewrites in case of potential changes during the execution of the optimized program. We believe IRIDIUM-extensions on these lines, which perform speculative optimizations enabled by precise static analyses and guarded by deoptimizable checks, to be one of the most promising directions for future dynamic-language research. As far as we are aware, IRIDIUM is the first attempt that can enable such exciting research for JavaScript programs.

9 CONCLUSION

In this paper, we presented IRIDIUM: a first-of-its-kind target-independent static optimization framework for JavaScript that can transparently integrate into existing JavaScript pipelines. The paper brought to the fore several challenges that make JavaScript static analysis and transformation difficult, and facilitated the same by designing an IR that makes various language semantics explicit. A cherry on the cake was the integration with a decent JavaScript backend, which allowed us to show that static optimizations can indeed be used to improve JavaScript performance.

We see three clear future directions opened up by our work: addition of more analysis and transformation passes to improve performance; integration with more JavaScript backends and reduction of JIT overheads using synergistic AOT+JIT optimizations; and possibly, extensions to IRIDIUM itself to allow static optimization of many more dynamic languages.

Data Availability Statement

The artifact for this paper, including source code, benchmarks, scripts and language specification, is available publicly with a relaxed license [23].

Acknowledgements

We thank the reviewers for their encouraging and useful feedback during the review process.

References

[1] 2025. ECMAScript 2025 Language Specification (ECMA-262, 16th edition). https://tc39.es/ecma262/2025/.

[2] 2025. JetStream 2: JavaScript and WebAssembly Benchmark Suite. https://browserbench.org/JetStream2.0/.

[3] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2023. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 153, 4 pages. doi:10.1145/3551349.3559522

[4] Amazon. 2025. LLRT: AWS Low Latency Runtime JavaScript Engine. https://github.com/awslabs/llrt.

[5] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. 2024. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. *Proc. ACM Program. Lang.* 8, PLDI, Article 159 (June 2024), 24 pages. doi:10.1145/3656389

[6] Apple. 2025. JavaScriptCore: Apple's Native JavaScript Engine Framework. https://developer.apple.com/documentation/javascriptcore.

[7] Fabrice Bellard, Charlie Gordon, and the QuickJS-NG Authors. 2025. QuickJS-NG: A Mighty JavaScript Engine (Next-Generation). https://quickjs-ng.github.io/quickjs/.

[8] Ecma. 2025. Test262: ECMAScript Conformance Test Suite. https://github.com/tc39/test262.

[9] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 2019. R melts brains: an IR for first-class environments and lazy effectful arguments. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. Association for Computing Machinery, New York, NY, USA, 55–66. doi:10.1145/3359619.3359744

[10] Dave Gamble and cJSON Contributors. 2025. cJSON: Ultralightweight JSON parser in ANSI C. https://github.com/DaveGamble/cJSON.

[11] Amanda Ghassaei. [n. d.]. OrigamiSimulator. https://github.com/amandaghassaei/OrigamiSimulator.

[12] Google. 2025. V8: Google's Open Source JavaScript and WebAssembly Engine. https://v8.dev/.

[13] Jordan Harband. [n. d.]. QS. https://github.com/ljharb/qs.

[14] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS) (LNCS, Vol. 5673)*. Springer-Verlag.

[15] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 121–132. doi:10.1145/2635868.2635904

[16] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 541–551. doi:10.1109/ASE.2015.28

[17] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI, Article 194 (June 2024), 24 pages. doi:10.1145/3656424

[18] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805

[19] Cheng Lou. [n. d.]. React-Motion. https://github.com/chenglou/react-motion.

[20] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 499–509. doi:10.1145/2491411.2491417

[21] Sebastian McKenzie. 2025. Babel: The JavaScript compiler / transpiler. https://babeljs.io/.

[22] Meetesh. 2026. LMA: Leave Me Alone. https://launchpad.net/~meetesh06/+archive/ubuntu/radio-banana.

[23] Meetesh Kalpesh Mehta, Anirudh Garg, Aneeket Yadav, and Manas Thakur. 2026. Artifact of "IRIDIUM: A Framework for Statically Optimizing JavaScript Programs". https://doi.org/10.5281/zenodo.18444575.

[24] Meetesh Kalpesh Mehta, Anirudh Garg, Aneeket Yadav, and Manas Thakur. 2026. Iridium 3JS Specification. https://github.com/CompL-Research/js3gen.

[25] Meetesh Kalpesh Mehta, Anirudh Garg, Aneeket Yadav, and Manas Thakur. 2026. Iridium Intermediate Representation Specification. https://compl-research.github.io/Iridium-Docs/.

[26] Anders Møller and Oskar Haarklou Veileborg. 2025. *Jelly*. https://github.com/cs-au-dk/jelly

[27] Mozilla. 2025. SpiderMonkey: Mozilla's JavaScript and WebAssembly Engine. https://spidermonkey.dev/.

[28] Joonyoung Park, Jihyeok Park, Dongjun Youn, and Sukyoung Ryu. 2021. Accelerating JavaScript Static Analysis via Dynamic Shortcuts. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1129–1140. doi:10.1145/3468264.3468556

[29] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript web applications using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion* (Buenos Aires, Argentina) *(ICSE-C '17)*. IEEE Press, 59–62. doi:10.1109/ICSE-C.2017.4

[30] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages. doi:10.1145/3133879

[31] Tadashi Saito and Hideya Iwasaki. 2025. Integrating Static Optimization and Dynamic Nature in JavaScript. In *Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Bergen, Norway) *(GPCE '25)*. Association for Computing Machinery, New York, NY, USA, 41–53. doi:10.1145/3742876.3742877

[32] Manuel Serrano. 2021. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.* 5, ICFP, Article 70 (Aug. 2021), 30 pages. doi:10.1145/3473575

[33] Anmolpreet Singh, Aayush Sharma, Meetesh Kalpesh Mehta, and Manas Thakur. 2023. Debugging Dynamic Language Features in a Multi-tier Virtual Machine. In *Proceedings of the 15th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL 2023, Cascais, Portugal, 23 October 2023*, Andrea Rosà and Martin Henz (Eds.). ACM, 18–28. doi:10.1145/3623507.3623549

[34] Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 140 (Oct. 2019), 29 pages. doi:10.1145/3360566

[35] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 50, 4, Article 59 (Aug. 2017), 34 pages. doi:10.1145/3106741

[36] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript Static Analysis via Localization and Remediation of Root Causes of Imprecision. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 487–498. doi:10.1145/2950290.2950338

APPENDIX A: SPECIFICATION OF 3JS

The 3JS language is defined as a constrained, formal subset of ECMAScript, engineered to prioritize differential testing of the transformations and allow simplified static analysis. The typespec of 3JS listed below is autogenerated from a specification that can be found online [24] (see the `languageRestriction.json` file for the complete specification). Rather than extending the language, each valid 3JS node maps directly to a standard JavaScript counterpart but imposes stricter requirements on its child properties. For example, while standard JavaScript permits any arbitrary expression within a conditional test, 3JS may restrict that same slot to a single `Identifier`.

```
JS3ConditionalExpression :=
  test = Identifier
  consequent = JS3ContainedExprKey
  alternate = JS3ContainedExprKey

JS3DebuggerStatement :=
  (empty)

JS3ArrayPattern :=
  elements = [Identifier | JS3RestElement]

JS3ObjectPattern :=
  properties = [JS3AssnObjectProperty | JS3RestElement]

JS3RestElement :=
  argument = Identifier

JS3WithStatement :=
  object = Identifier
  body = JS3BlockStatement

JS3TaggedTemplateExpression :=
  tag = JS3MemberExpression | Identifier
  quasi = JS3TemplateLiteral
  typeParameters = null

JS3TemplateLiteral :=
  quasis = [TemplateElement]
  expressions = [Identifier]

JS3UpdateExpression :=
  argument = Identifier | JS3MemberExpression

JS3SwitchCase :=
  test = JS3ContainedExprKey | null
  consequent = [JS3AllowedBlockStatement]

JS3SwitchStatement :=
  discriminant = Identifier
  cases = [JS3SwitchCase]

JS3BreakStatement :=
  label = Identifier | null
```

```
JS3ContinueStatement :=
  label = Identifier | null

JS3DoWhileStatement :=
  test = JS3ContainedExprKey
  body = JS3BlockStatement

JS3ForStatement :=
  init = VariableDeclaration | JS3ContainedExprKey | null
  test = JS3ContainedExprKey | null
  update = JS3ContainedExprKey | null
  body = JS3BlockStatement

JS3ForInStatement :=
  left = VariableDeclaration | LVal
  right = Identifier
  body = JS3BlockStatement

JS3ForOfStatement :=
  left = VariableDeclaration | LVal
  right = Identifier
  body = JS3BlockStatement

JS3WhileStatement :=
  test = JS3ContainedExprKey
  body = JS3BlockStatement

JS3VariableDeclarator :=
  id = JS3VarDeclLVal
  init = null | undefined | JS3AssnInit

JS3VariableDeclaration :=
  declarations = [JS3VariableDeclarator]

JS3ArrayExpression :=
  elements = [null | JS3ArrayTerminals | JS3SpreadElement]

JS3CallExpression :=
  callee = JS3Import | Identifier | Super | V8IntrinsicIdentifier
  arguments = [ Identifier | JS3SpreadElement ]
  typeArguments = null
  typeParameters = null

JS3UnaryExpression :=
  argument = Identifier | JS3MemberExpression

JS3IfStatement :=
  test = Identifier
  consequent = JS3BlockStatement
  alternate = null | undefined | JS3BlockStatement
```

```
JS3BinaryExpression :=
  left = Identifier | JS3PrivateName
  right = Identifier

JS3AssignmentExpression :=
  left = JS3AssnLVal
  right = JS3AssnInit
  operator = "="

JS3MemberExpression :=
  object = Identifier | ThisExpression | Super
  property = Identifier | JS3PrivateName

JS3ReturnStatement :=
  argument = undefined | null | Identifier

JS3RegExpLiteral :=
  (empty)

JS3PrivateName :=
  (empty)

JS3EmptyStatement :=
  (empty)

JS3ExportNamespaceSpecifier :=
  (empty)

JS3ObjectExpression :=
  properties = [JS3ObjectMethod | JS3ObjectProperty | JS3SpreadElement]

JS3ObjectMethod :=
  key = Identifier | StringLiteral | NumericLiteral | BigIntLiteral
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  decorators = null
  returnType = null
  typeParameters = null

JS3ObjectProperty :=
  key = Identifier | StringLiteral | NumericLiteral | BigIntLiteral
  value = JS3ObjectTerminals
  decorators = null

JS3SpreadElement :=
  argument = Identifier

JS3MetaProperty :=
  (empty)

JS3ClassExpression :=
  superClass = null | undefined | JS3ContainedExprKey
  body = JS3ClassBody
  decorators = null
  implements = null
  mixins = null
  superTypeParameters = null
  typeParameters = null

JS3ClassBody :=
  body = [JS3StaticBlock | JS3ClassProperty |
          JS3ClassMethod | JS3ClassPrivateProperty
          | JS3ClassPrivateMethod]

JS3ClassProperty :=
  key = JS3ContainedExprKey
  value = JS3ClassPropValue
  typeAnnotation = null
  decorators = null
  variance = null

JS3ClassMethod :=
  key = JS3ContainedExprKey
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  decorators = null
  returnType = null
  typeParameters = null

JS3ClassPrivateProperty :=
  key = JS3PrivateName
  value = JS3ClassPropValue
  decorators = null
  typeAnnotation = null
  variance = null

JS3ClassPrivateMethod :=
  key = JS3PrivateName
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  decorators = null
  returnType = null
  typeParameters = null

JS3StaticBlock :=
  body = [JS3AllowedBlockStatement]

JS3YieldExpression :=
  argument = Identifier | null

JS3AwaitExpression :=
  argument = Identifier

JS3NewExpression :=
  callee = Identifier | Super | V8IntrinsicIdentifier
  arguments = [Identifier | JS3SpreadElement]
  typeArguments = null
  typeParameters = null

JS3LabeledStatement :=
  body = JS3AllowedBlockStatement

JS3BlockStatement :=
  body = [JS3AllowedBlockStatement]

JS3FunctionDeclaration :=
  id = Identifier
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  predicate = undefined | null
  returnType = undefined | null
  typeParameters = undefined | null

JS3FunctionExpression :=
  id = null | Identifier | undefined
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  predicate = undefined | null
  returnType = undefined | null
  typeParameters = undefined | null

JS3ArrowFunctionExpression :=
  params = [JS3AllowedFunctionArgs]
  body = JS3BlockStatement
  predicate = null
  returnType = null
  typeParameters = null

JS3ExportSpecifier :=
  (empty)

JS3ExportNamedDeclaration :=
  declaration = null
  specifiers = [JS3ExportSpecifier | JS3ExportNamespaceSpecifier]
  assertions = null
  attributes = null

JS3ExportAllDeclaration :=
  assertions = null
  attributes = null

JS3ExportDefaultDeclaration :=
  declaration = Identifier

JS3ThrowStatement :=
  argument = Identifier
```

```
JS3CatchClause :=
  body = JS3BlockStatement
  param = Identifier | null

JS3TryStatement :=
  block = JS3BlockStatement
  handler = null | undefined | JS3CatchClause
  finalizer = null | undefined | JS3BlockStatement

JS3Import :=
  (empty)

JS3ImportExpression :=
  source = Identifier
  options = Identifier | null

JS3ImportDeclaration :=
  specifiers = [ImportSpecifier | ImportNamespaceSpecifier]
  assertions = undefined | null
  attributes = undefined | null

JS3Program :=
  body = [ JS3AllowedProgStatement ]

JS3File :=
  program = JS3Program
```

APPENDIX B: SPECIFICATION OF IRI

The complete documentation of IRI can be found at [25]. Each node in IRI is an S-Expression node of the kind:

```
IridiumSEXP = ("TAG", [...IridiumSEXP], [["FlagName", IridiumPrimitive]...])
IridiumPrimitive = number | boolean | string | null
```

The BNF for IRI is shown below:

```
FileSEXP :=
    JSModuleExtensions*
    BBContainerSEXP+
BBContainerSEXP :=
    BindingsSEXP
    List<BBSEXP>
BindingsSEXP :=
    List<EnvBindingSEXP>
    List<RemoteEnvBindingSEXP>
    List<PoolBindingSEXP>
RVAL :=
    StackPopSEXP
    | UnopSEXP
    | BinopSEXP
    | ListSEXP
    | AwaitSEXP
    | EnvReadSEXP
    | FieldReadSEXP
    | FieldWriteSEXP
    | EnvWriteSEXP
    | ApplySEXP
    | CallSiteSEXP
    | BooleanSEXP
    | LambdaSEXP
    | NullSEXP
    | NumberSEXP
    | StringSEXP
    | NUBDSEXP
    | {JS_EXTENSIONS}*
STMT :=
    StackRejectSEXP
    | StackRetainSEXP
    | GotoSEXP
    | IfElseJumpSEXP
    | YieldSEXP
    | ThrowSEXP
    | PopCatchContextSEXP
    | PushCatchContextSEXP
    | ReturnSEXP
    | ReturnAsyncSEXP
    | {JS_EXTENSIONS}*
BBSEXP := {STMT}*
```

APPENDIX C: TRANSFER FUNCTIONS FOR STATIC ANALYSES

This appendix describes the transfer functions for the various analyses and optimization passes currently implemented in IRIDIUM. All the passes (except TDZA) work with local variables that do not outlive the stack frame and are not reachable by any tainted scope (see Section 5.2); TDZA, being particularly useful, is performed interprocedurally within top-level closure scopes. We use the set Blacklist to denote captured/tainted bindings.

Analysis 1: ConstantsAtStmt. The ConstantsAtStmt pass is a forward dataflow analysis that identifies constants by tracking the abstract values of variables at each program point. It operates on a lattice that maps variables to values representing an uninitialized state (⊥), a known constant type (e.g., Number), or a non-constant value (⊤).

$$\text{LatticeVal} = \{\top,\ \Lambda,\ \mathsf{Null},\ \mathsf{Boolean},\ \mathsf{Number},\ \mathsf{String},\ \mathsf{JSBigInt},\ \bot\}$$

$$\text{Lattice} = \{\text{EnvBinding} \mapsto \text{LatticeVal}\}$$

$$\begin{aligned}\text{merge}(m_1, m_2) = {} & \{\, (x \mapsto \text{cprop}(y_1, y_2)) \mid (x \mapsto y_1) \in m_1 \wedge (x \mapsto y_2) \in m_2 \,\} \\ & \cup \{\, (x \mapsto y) \mid [(x \mapsto y) \in m_1 \wedge (x \mapsto y) \in m_2] \,\}\end{aligned}$$

$$\text{cprop}(v_1, v_2) = \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{if } v_2 = \bot \\ \Lambda & \text{if } v_1 = v_2 = \Lambda \\ \mathsf{Null} & \text{if } v_1 = v_2 = \mathsf{Null} \\ v_1 & \text{if } \text{val}(v_1) = \text{val}(v_2) \\ \top & \text{otherwise} \end{cases}$$

$$\text{gen}(v) = \begin{cases} \Lambda & v \text{ is a non-constant value} \\ \mathit{Null} \mid \mathit{Boolean} \mid \mathit{Number} \mid \mathit{String} \mid \mathit{JSBigInt} & \mathit{v\ is\ a\ constant} \\ \top & \text{otherwise} \end{cases}$$

$$\text{In}(n) = \begin{cases} \{\, x \mapsto \bot \mid x \in \text{uncaptured(scope(n))} \,\} & \text{if } n = \text{StartBB} \\ \displaystyle\bigcup_{p \in \text{Preds}(n)} \text{Out}(p) & \text{otherwise} \end{cases}$$

$$\text{Out}(n) = \{\, x \mapsto y \mid [x \mapsto y] \in \text{In}(n),\ x \neq \text{def}(n) \,\} \ \cup\ \{\, \text{def}(n) \mapsto \text{gen}(\text{rval}(n)) \,\}$$

Analysis 2: CopyPropInfo. The CopyPropInfo pass implements a forward dataflow analysis that identifies copy relationships between environment bindings.

$$\begin{aligned}
\text{Lattice} &= \{\, \text{EnvBinding} \mapsto \text{EnvBinding} \,\} \\
\text{merge}(m_1, m_2) &= \{\, (x \mapsto y) \mid (x \mapsto y) \in m_1 \land (x \mapsto y) \in m_2 \,\} \\
\text{kill}(n) &= \mathsf{Blacklist} \cup \{\, x \mid x \in \text{lhs}(n) \,\} \\
\text{gen}(n) &= \begin{cases} \{\, \text{lhs}(n) \mapsto \text{rhs}(n) \,\} & \text{if rhs}(n) \text{ is an EnvBinding} \\ \emptyset & \text{otherwise} \end{cases} \\
\text{In}(n) &= \begin{cases} \emptyset & \text{if } n = \text{StartBB} \\ \text{merge}\big(\{\text{Out}(p)\}_{p \in \text{Preds}(n)}\big) & \text{otherwise} \end{cases} \\
\text{Out}(n) &= \big(\text{In}(n) \setminus \{\, (x \mapsto y) \mid x \in \text{kill}(n) \lor y \in \text{kill}(n) \,\}\big) \cup \text{gen}(n)
\end{aligned}$$

Analysis 3: EffectAtStmt. The EffectAtStmt pass is a forward dataflow analysis that tracks the most recent effectful operation within a basic block. Unlike traditional constant or copy propagation analyses, which maintain sets or maps of abstract bindings, this analysis maintains a single abstract summary describing whether a valid, possibly side-effect-producing statement is currently "active" in the flow of execution. Its main objective is to identify opportunities for effect merging, reordering, and elimination of redundant writes when no intervening effectful statement invalidates the last stored effect. In essence, it attempts to answer the question: *"Is the previous store still valid, or has it been invalidated by an effectful statement?"*

$$\begin{aligned}
\text{LatticeVal} &= \{\, \bot,\ \langle \text{store}, \text{effect} \rangle,\ \top \,\} \\
\text{merge}(v_1, v_2) &= \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{if } v_2 = \bot \\ v_1 & \text{if } v_1 = v_2 \\ \top & \text{otherwise} \end{cases} \\
\text{In}(n) &= \begin{cases} \bot & \text{if } n = \text{StartBB} \\ \text{merge}\big(\{\text{Out}(p)\}_{p \in \text{Preds}(n)}\big) & \text{otherwise} \end{cases} \\
\text{Out}(n) &= \begin{cases} \langle \text{store}(n), \text{effect}(n) \rangle & \text{if } n \text{ is } \mathsf{EnvWrite} \land n \notin \textit{Blacklist} \land \text{rhs}(n) \text{ may be effectful} \\ \top & \text{if } n \text{ is } \mathsf{EnvWrite} \lor n \text{ is effectful} \\ \text{In}(n) & \text{otherwise} \end{cases}
\end{aligned}$$

Analysis 4: Liveness. The Liveness pass is a classic backward dataflow analysis that determines, for each program point, which variables may be used along some path before being redefined. Additionally, the bindings in the Blacklist are always considered live to prevent incorrect eliminations.

$$\begin{aligned}
\text{Lattice} &= \{\,\text{EnvBinding}\,\} \\
\text{merge}(v_1, v_2) &= v_1 \cup v_2 \\
\text{In}(n) &= \text{Use}(n) \cup \big(\text{Out}(n) - \text{Def}(n)\big) \cup \text{Blacklist} \\
\text{Out}(n) &= \begin{cases} \text{Blacklist} & \text{if } n = \text{StartBB} \\ \displaystyle\bigcup_{s \in \text{Succs}(n)} \text{In}(s) & \text{otherwise} \end{cases}
\end{aligned}$$

Analysis 5: SafeKeyPropertyAccess. The SafeKeyPropertyAccess pass is a forward dataflow analysis that computes, for each program point, whether property key expressions used in object property accesses are provably safe (i.e., string or number) to prevent redundant casts. For the bindings in the Blacklist, updates are skipped to maintain soundness.

$$\begin{aligned}
\text{LatticeVal} &= \{\, \mathsf{SAFE}, \mathsf{UNSAFE}, \bot \,\} \\
\text{Lattice} &= \{\, \text{EnvBinding} \mapsto \text{LatticeVal} \,\} \\
\text{merge}(m_1, m_2) &= \{\, (x \mapsto \text{cprop}(y_1, y_2)) \mid (x \mapsto y_1) \in m_1 \land (x \mapsto y_2) \in m_2 \,\} \\
&\quad \cup \{\, (x \mapsto y) \mid [(x \mapsto y) \in m_1 \land (x \mapsto y) \in m_2] \,\}
\end{aligned}$$

$$\text{cprop}(v_1, v_2) = \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{if } v_2 = \bot \\ \mathsf{SAFE} & \text{if } v_1 = v_2 = \mathsf{SAFE} \\ \mathsf{UNSAFE} & \text{otherwise} \end{cases}$$

$$\text{isSafeKey}(k, m) = \begin{cases} \text{true} & k \text{ is string or numeric} \\ m[k] \neq \mathsf{UNSAFE} & k \text{ is an EnvBinding} \\ \text{isSafeKey}(k', m) & k = unop(k') \\ \text{isSafeKey}(k_1, m) \ \land\ \text{isSafeKey}(k_2, m) & k = binop(k_1, k_2) \\ \text{false} & \text{otherwise} \end{cases}$$

$$\text{gen}(k, m) = \begin{cases} \mathsf{SAFE} & \text{if isSafeKey}(k, m) \\ \mathsf{UNSAFE} & \text{otherwise} \end{cases}$$

$$\text{In}(n) = \begin{cases} \{\, x \mapsto \bot \mid x \in \text{uncaptured(scope(n))} \,\} & \text{if } n = \text{StartBB} \\ \displaystyle\bigcup_{p \in \text{Preds}(n)} \text{Out}(p) & \text{otherwise} \end{cases}$$

$$\text{Out}(n) = \{\, x \mapsto y \mid [x \mapsto y] \in \text{In}(n),\ x \neq \text{def}(n) \,\} \ \cup\ \{\, \text{def}(n) \mapsto \text{gen}(\text{rval}(n), \text{In}(n)) \,\}$$

Analysis 6: TDZA. The Temporal Dead Zone Analysis (TDZA) pass is a forward dataflow analysis that computes, for each program point, whether variable bindings are safely initialized or risk access in the TDZ (Temporal Dead Zone), following JavaScript's semantics. The lattice tracks three states: TDZ (unsafe), SAFE, and ⊥ (uninitialized/unknown).

$$
\begin{aligned}
\text{TDZLatticeVal} &= \{\, \mathsf{TDZ},\ \mathsf{SAFE},\ \bot \,\} \\
\text{TDZLattice} &= \{\, \text{EnvBinding} \mapsto \text{TDZLatticeVal} \,\} \\
\text{merge}(m_1, m_2) &= \{\, (x \mapsto \text{tdz}(y_1, y_2)) \mid (x \mapsto y_1) \in m_1 \ \wedge\ (x \mapsto y_2) \in m_2 \,\} \\
&\quad \cup \{\, (x \mapsto y) \mid [(x \mapsto y) \in m_1 \wedge (x \mapsto y) \in m_2] \,\} \\
\text{gen}(v) &= \begin{cases} \mathsf{TDZ} & v = \Lambda \\ \mathsf{SAFE} & \text{otherwise} \end{cases} \\
\text{tdz}(v_1, v_2) &= \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{if } v_2 = \bot \\ \mathsf{TDZ} & \text{if } v_1 = \mathsf{TDZ} \vee v_2 = \mathsf{TDZ} \\ \mathsf{SAFE} & \text{otherwise} \end{cases} \\
\text{In}(n) &= \begin{cases} \{\, x \mapsto \mathsf{SAFE} \mid x \in \text{uncaptured}(\text{scope}(n)) \,\} & n = \text{StartBB} \\ \displaystyle\bigcup_{p \in \text{Preds}(n)} \text{Out}(p) & \text{otherwise} \end{cases} \\
\text{Out}(n) &= \{\, x \mapsto y \mid [x \mapsto y] \in \text{In}(n),\ x \neq \text{def}(n) \,\} \\
&\quad \cup \begin{cases} \{\text{def}(n) \mapsto \text{gen}(\text{rval}(n))\} & \text{if the write at } n \text{ is } \mathsf{SAFE} \\ \varnothing & \text{otherwise} \end{cases}
\end{aligned}
$$

APPENDIX D: QUALITATIVE ANALYSIS CLASSIFICATION

Category	Tier	Opcodes
C-SITE	Least Optimized	OP_call, OP_call_method, OP_return
	Moderately Optimized	—
	Highly Optimized	OP_return_undef
	Most Optimized	OP_call0, OP_call1, OP_call2, OP_call3, OP_tail_call, OP_tail_call_method
CFLOW	Least Optimized	OP_if_false, OP_if_true, OP_goto
	Moderately Optimized	—
	Highly Optimized	OP_goto16
	Most Optimized	OP_if_false8, OP_if_true8, OP_goto8
STKOP	Least Optimized	OP_get_loc_check, OP_put_loc_check, OP_put_loc_check_init, OP_get_var_ref_check, OP_put_var_ref_check, OP_put_var_ref_check_init
	Moderately Optimized	OP_get_loc, OP_put_loc, OP_set_loc, OP_get_arg, OP_put_arg, OP_set_arg, OP_get_var_ref, OP_put_var_ref, OP_set_var_ref, OP_get_loc8, OP_put_loc8, OP_set_loc8
	Highly Optimized	OP_get_loc0, OP_get_loc1, OP_get_loc2, OP_get_loc3, OP_put_loc0, OP_put_loc1, OP_put_loc2, OP_put_loc3, OP_get_arg0, OP_get_arg1, OP_get_arg2, OP_get_arg3, OP_put_arg0, OP_put_arg1, OP_put_arg2, OP_put_arg3, OP_get_var_ref0, OP_get_var_ref1, OP_get_var_ref2, OP_get_var_ref3, OP_put_var_ref0, OP_put_var_ref1, OP_put_var_ref2, OP_put_var_ref3
	Most Optimized	OP_get_loc0_loc1, OP_set_arg0, OP_set_arg1, OP_set_arg2, OP_set_arg3, OP_set_var_ref0, OP_set_var_ref1, OP_set_var_ref2, OP_set_var_ref3, OP_set_loc0, OP_set_loc1, OP_set_loc2, OP_set_loc3
OBJOP	Least Optimized	OP_get_array_el, OP_to_propkey, OP_to_propkey2
	Moderately Optimized	OP_put_field, OP_get_field
	Highly Optimized	—
	Most Optimized	OP_get_array_el2, OP_get_field2, OP_get_length

Fig. 20. Bytecode classification into various tiers for qualitative analysis.

APPENDIX E: IRIDIUM SUPPORT

Table 1. Features not yet supported by IRIDIUM.

Feature	**Status**	**Remarks**
`with`	Unsupported	Deprecated from ECMAScript.
Yield delegation	Unsupported	Yet to be implemented.
Optional chaining	Unsupported	Yet to be implemented.
Dynamic imports	Partially supported	Lacks testing infrastructure.
Niche usages of `eval`	Partially supported	Special lookup semantics in arg-init scope.

Received 2025-10-10; accepted 2026-02-17