Static Analysis of JavaScript Programs

Annual Progress Report-2

Meetesh Kalpesh Mehta
23D0361

10th September, 2025

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

Contents

Abstract

Static analysis of JavaScript remains notoriously difficult due to the language's dynamic typing, 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. We introduce Iridium, an intermediate representation (IR) designed to make JavaScript's semantics explicit and analyzable. Iridium systematically lowers JavaScript into a structured, three-address-code–style form where bindings, environments, and control flow are uniformly represented. This explicitness enables more predictable analyses and transformations, ranging from dataflow tracking to optimization passes, that are otherwise hindered by the language's complexity. By bridging the gap between JavaScript's expressive surface syntax and the requirements of static analysis, Iridium lays the groundwork for a new generation of tools that can reason effectively about modern JavaScript applications.

Chapter 1

Introduction

Modern software development increasingly leverages dynamically typed languages. Today, JavaScript is popularly used for web applications, server-side logic, and rapid prototyping. Despite its ubiquity and expressive power, JavaScript remains one of the most challenging languages to analyze statically. Unlike statically typed ecosystems with mature toolchains such as LLVM [7] for C/C++ or Soot [11] for Java, comparable infrastructure for dynamic languages is fragmented or underpowered. The root of this disparity lies in JavaScript's runtime flexibility $\rightsquigarrow$ dynamic typing, unstructured scoping, implicit coercions, and runtime meta-programming. ([6], [8] look at these problems in context of the R programming language.)

Our work (called the *Iridium* framework) addresses the aforementioned problems through a multi-stage architecture that lowers JavaScript into an explicit, uniform, S-expression–based IR. It then applies a sequence of transformation passes to resolve scopes, make explicit control flows, and resolve bindings to their logical stack frames. In this chapter, we first explore the foundational challenges of statically analyzing dynamic languages like JavaScript. We then position Iridium within the landscape of existing IRs and tools, and explain how its architectural design seeks to reconcile flexibility with analyzability.

1.1 Challenges

Dynamic languages like JavaScript defer many decisions to runtime; variable types are determined on the fly, scopes can be shadowed or modified, constructs like `eval`, dynamic imports, and reflection blur static boundaries. As a result, static analysis systems often resort to aggressive over-approximation, creating imprecise results or missing critical behaviors altogether.

1.1.1 Dynamic Types

```
1 if (foo)
2   var x = "hello";
3 else
4   var x = 42;
5 o.p = x;
```

Statically, `x` can be either string or number at the assignment point (Line 5), and any precise dataflow analysis must model this ambiguity.

1.1.2 Selective Scope Lookups

```
1 let x = "I am outside x";
2 let A = "I am A";
3 function foo(
4     A,
5     B = function() { console.log(A); },
6     C = function() { console.log(x); }
7 ) {
8   let x = "I am inner x";
```

```
  B(); // ?
  C(); // ?
}
foo("Meetesh");
```

In this program, JavaScript creates a *parameter scope* during function initialization, which is distinct from the function body scope, and default parameter initializers are evaluated in this parameter scope. As a result, the parameter `A` is directly visible to the closure `B`, whereas the `let x` declared inside the body of `foo` is not yet available when evaluating the default value of `C`, causing `C` to resolve `x` to the outer binding `"I am outside x"` (see Figure 1.1).

```
Meetesh
I am outside x
```

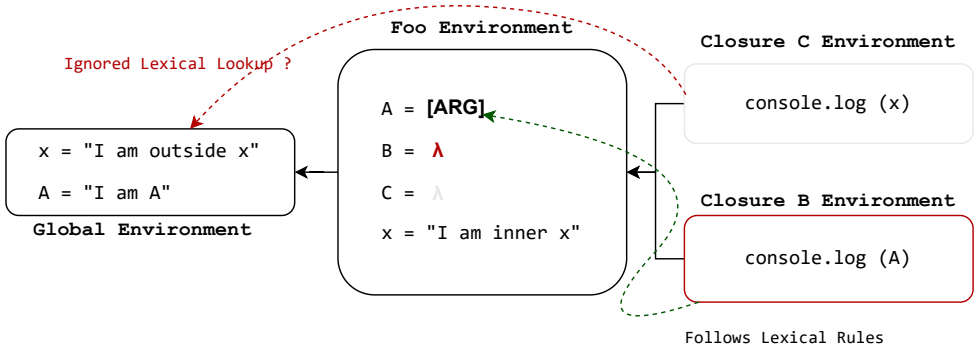


Figure 1.1: This figure shows the actual lexical lookup performed at run-time.

This example illustrates a counterintuitive but standard behaviour of ECMAScript: the lookup rules inside default parameter initializers do not align with the rules inside the function body, resulting in seemingly inconsistent closure capture.

1.1.3 Hoisting and Initialization

```
function foo(A) {
  var A;
  var B;
  console.log(A, B);
}

foo("Arg A");
```

Consider the code shown above. When the function (`foo`) is instantiated, the parameter `A` is first bound to the argument `"Arg A"`, after which `var`-declarations are hoisted. The statement `var A;` becomes a no-op because the parameter binding already exists in the same variable environment, while `var B;` introduces a new variable initialized with `undefined`. As a result, the program prints `Arg A undefined`.

Note

Although such cases may seem unusual and are rarely encountered in practical JavaScript code, we discovered these quirks while validating our framework against official language conformance tests. Since Iridium not only analyzes programs but also generates correct executable output, it is essential that our intermediate representation precisely models these semantics.

When compounded over larger codebases, especially with nested closures, dynamic property access, or exception-based control flow, these ambiguities proliferate and render static analysis brittle. Additionally, JavaScript's control flow is not always structured. Constructs such

as implicit fallthroughs, hidden exception edges, and continuation-passing through callbacks or asynchronous constructs generate undocumented paths through the program's CFG. Without an IR that makes these paths explicit, static analysis tools cannot reliably reason about correctness or security properties.

1.2 Motivation and Progress

IRs such as those employed in static typed ecosystems (LLVM, Soot, Graal IR [10]) are insufficient for JavaScript without extensive adaptation, yet such adaptation often sacrifices crucial dynamic behaviors. Tools like TypeDevil [9] target JavaScript's dynamic nature by operating at runtime to detect type inconsistencies. Other efforts focus on specific problem domains, such as string analysis or staged meta-programming [5]. What sets Iridium apart is its goal: to construct a general-purpose, analyzable, and executable IR for JavaScript that retains full semantic fidelity while being amenable to static conversion and transformation.

1.2.1 Current Progress

- **Setting up the pipeline**: We realized that without a clear end-to-end pipeline, working on individual components in isolation would lead us astray. Our plan is therefore straightforward: start from source code, transform it iteratively, and finally integrate a backend.
  - The main project, **Iridium**, accepts source code and converts it into an intermediate Iridium representation. At this stage, the code is not yet executable but is ready to be serialized.
  - The intermediate project, **Forge**, consumes this representation and reduces transitional (not-yet-resolved) nodes into concrete nodes.
  - The backend (parser and instruction selection pass) then converts Iridium code into executable bytecode for the QuickJS virtual machine.
- **Testing**: JavaScript conveniently provides extensive official language testing [4] and benchmarking suites. We currently employ the Test262 language conformance suite to measure compliance (positive testing against QuickJS, which itself sits at around 85%). In addition, we make use of the SunSpider, Kraken, and V8 benchmark suites [3], though we are still in the process of identifying the most suitable benchmarks for our project [1].
- **Thoughts**: We observed that a simple heuristic-based merge pass ("Chapati") for flattening canonical CFGs into runtime–friendly code yielded measurable performance improvements. This suggests that further exploration into optimal placement of basic blocks could be valuable, though we refrain from drawing premature conclusions without more systematic evidence.
- **Failures**: The progress reported here mainly stems from work conducted after March 2025. During the three months prior, we pursued a top-down design of the Iridium language that focused on developing precise pointer analysis techniques capable of scaling to large JavaScript programs. This effort ultimately failed for two reasons: (i) developing such an IR without a proper pipeline meant that many semantic issues went undetected until very late, making the conversion from JavaScript to Iridium brittle and impractical; and (ii) the more ambitious goal of making Iridium both forward and backward compatible (i.e., supporting both transformation back into JavaScript and targeting backends like V8 or QuickJS) proved unreachable. Nevertheless, this attempt yielded valuable insights into precise pointer analysis for JavaScript, which we intend to explore further in the future.

1.3 Is it only JavaScript?

No. Iridium is designed to support additional frontends and backends in the future. Conceptually, we separate Iridium into a core language and a set of language-specific operations, which we call *extensions*. Our hypothesis is that many dynamic languages such as Lua and Python will share this common core. Each pass in Iridium systematically strips away language-specific constructs

[1] Benchmarks like JetStream 2.2 [2] have a more wider range of benchmarks, but a lot of these are meant to be representative of browser *like* workloads which QuickJS does not target. We are looking into cherry picking the most relevant workloads for future performance evaluations.

into these core operations, which are intended to capture the essential semantics of a general dynamic language.

Is there a formal proof that Iridium can faithfully represent all such languages? Not at present. However, empirical observation suggests that these languages share significant structural and semantic similarities. Building semantics for each language independently from the ground up is prohibitively complex; instead, our approach is to iteratively and systematically derive the common language through implementation and practice.

Chapter 2

Design Principles of Iridium

At the heart of Iridium is a uniform representation of program structure based on *S-expressions*. Every Iridium node has exactly three components:

- **Tag:** a string identifying the kind of S-expression (e.g., `EnvWriteSEXP`, `StringSEXP`, `IfJumpSEXP`).
- **Args:** a list of child nodes (`IridiumSEXP[]`) representing the recursive structure of the program.
- **Flags:** a list of key–value pairs (`[string, IridiumPrimitives][]`) that encode primitive values or metadata annotations. Unlike `Args`, `Flags` cannot contain nested Iridium nodes.

2.1 Uniform Node Representation

All Iridium nodes share a common structure defined by three fields: *tag*, *args*, and *flags*. This design gives Iridium a Lisp-like uniformity, where every program fragment is represented as a tree node, regardless of whether it corresponds to a control-flow construct, a literal value, or a binding declaration.

Another benefit of such a structure is *ease of serialization and deserialization across different languages*. Our framework uses a TypeScript project to load source files and emit Iridium S-expressions, which are then consumed by a separate C++ project to perform analysis, transformations, and optimizations. The serialization format is a nested array that can be written to disk or transmitted between processes with minimal overhead.

TypeScript Implementation. The root definition of Iridium nodes is written in TypeScript. The `IridiumSEXP` class provides the base structure, together with methods for serialization, flag management, and string rendering:

```
export type IridiumPrimitives = number | boolean | string | null;

export class IridiumSEXP {
  tag: string = "IridiumSEXP";
  args: Array<IridiumSEXP> = [];
  flags: Array<[string, IridiumPrimitives]> = [];

  constructor(tag: string) { this.tag = tag; }

  serialize(): Array<any> {
    return [
      this.tag,
      this.args.map((e) => e.serialize()),
      this.flags.map(([flagName, e]) => [flagName, e])
    ];
  }

  setFlag(flag: string, val: IridiumPrimitives = null) { ... }
  getFlag(flag: string): IridiumPrimitives { ... }
  toString(space = 0): string { ... }
}
```

In this model, the `tag` identifies the node type, `args` holds child nodes, and `flags` stores primitive metadata such as names, types, or source locations. Because flags are limited to primitive values, the boundary between tree structure and annotation is kept clean and analyzable.

C++ Specialization. On the analysis side, a C++ library consumes serialized Iridium S-expressions and specializes them into typed representations. The `ParseIridiumTypes` utility maps tags to their corresponding C++ subclasses:

```
class ParseIridiumTypes {
public:
  static IRISEXP specialize(IRISEXP obj) {
    auto & tag = obj->tag;
    if (tag == "File") return FileSEXP::generateFrom(obj);
    if (tag == "ResolveEnvBinding")
      return ResolveEnvBindingSEXP::generateFrom(obj);
    ...
    throw std::runtime_error(
      "ParseIridiumTypes::specialize unhandled Tag: " + tag
    );
  }
};
```

For example, the `ResolveEnvBinding` node has a dedicated C++ class that exposes flag-based accessors (`getNAME`, `setASW`, etc.), while preserving the underlying uniform structure:

```
class ResolveEnvBindingSEXP : public IridiumSEXP {
public:
  static std::shared_ptr<ResolveEnvBindingSEXP>
  generateFrom(IRISEXP obj) {
    assert(obj->tag == "ResolveEnvBinding");
    auto res = std::shared_ptr<ResolveEnvBindingSEXP>(
        new ResolveEnvBindingSEXP());
    res->tag   = obj->tag;
    res->args  = std::move(obj->args);
    res->flags = std::move(obj->flags);
    return res;
  }
  void setNAME(const std::string &value) { setFlag("NAME", value); }
  std::string getNAME() { return getFlagString("NAME"); }
};
```

Implications. This architecture separates the *language-neutral core* (defined in TypeScript) from the *analysis and transformation logic* (implemented in C++). Serialization bridges the two, enabling cross-language pipelines and ensuring that Iridium remains extensible, portable, and analyzable with minimal tooling overhead.

> Note
>
> This process is automated by using a side project which takes as input the language specification (in JSON format) and automatically generates the parser for C++. Currently this is a work in progress and will target TypeScript in the future. This project is very similar to the `JS3Gen` project presented last year (which we use to generate support classes while lowering JavaScript into 3JS).

2.1.1 Node Classification

Iridium IR organizes its node types into a clear taxonomy, enabling structured transformation and precise static analysis. Following the official Iridium documentation [1], the node classifications are as follows:

1. Abstract Operations These nodes represent operations that are not yet fully resolved. They serve as placeholders during desugaring or early IR construction. For instance, symbolic references to variables whose exact location (e.g., stack vs. global) is not yet known. Such nodes are later rewritten into concrete environment- or stack-access operations via the ‘Bindings‘ mechanism.

2. Calls Nodes in this category model invocation semantics:

- *Generic Calls*: Represent calls to user-defined or external functions, respecting various calling conventions.
- *VSysCalls (Virtual System Calls)*: Expose internal VM-level operations, providing a desugared form of complex or built-in language constructs.

3. Environment These nodes explicitly represent and manipulate closure environments:

- The 'Bindings' object models logical stack frames, capturing both local variables and external references.
- Environment-related nodes handle declaration, assignment, and reading of values in a controlled, analyzable manner.

4. Flow Flow nodes control how programs execute, covering both synchronous and asynchronous constructs:

- *Synchronous Flow*: Includes conditionals, loops, early exits ('return', 'break'), etc.
- *Asynchronous Flow*: Captures 'await', promise-related constructs, and other VM-level event patterns.

5. RVal (Right-hand Values) These nodes represent computations that return a value and are valid only on the right-hand side of assignments. They include:

- Primitive values like literals or identifiers.
- Language-specific value-producing expressions that don't cause side-effects.

6. Structural Structural nodes define the broader organization of an Iridium program:

- *Compilation Units*: Top-level IR program entry points.
- *TopLevelContainers*: Modules or containers holding declarations and metadata (e.g. imports/exports).
- *Basic Blocks*: Consecutive instruction sequences crucial for control- and data-flow analyses.
- They also embed JavaScript-specific module semantics to support future inter-modular analyses.

2.2 BNF Representation of Iridium IR

To make the structure of Iridium programs explicit, we describe the language of Iridium nodes using a BNF-style grammar. This formalization ensures that analyses and transformations can be applied systematically, while also providing a concise reference for implementers. The grammar distinguishes between program structure (files, containers, and bindings), expressions that return values, and statements that affect control-flow or the environment.

2.2.1 Top-Level Structure

```
FileSEXP ::= List<ModuleRequest>
           | List<StaticImport>
           | List<LocalStaticExportSEXP | NamedReexportSEXP>
           | List<StarExport>
           | BBContainerSEXP+
```

A file consists of module-related directives (imports, exports, re-exports) and one or more `BBContainerSEXP` units, which hold the executable code.

2.2.2 Basic Block Containers

$$\begin{aligned} \texttt{BBContainerSEXP} ::= \ & \texttt{BindingsSEXP} \\ & \mid \texttt{List<BBSEXP>} \end{aligned}$$

$$\begin{aligned} \texttt{BindingsSEXP} ::= \ & \texttt{List<EnvBindingSEXP>} \\ & \mid \texttt{List<RemoteEnvBindingSEXP>} \\ & \mid \texttt{List<PoolBindingSEXP>} \end{aligned}$$

Bindings represent the logical environment (locals, captured variables, or pooled resources). Each basic block container encapsulates these bindings alongside a sequence of basic blocks.

2.2.3 Right-Hand Values (RVAL)

$$\begin{aligned} \{RVAL\} ::= \ & \texttt{StackPopSEXP} \mid \texttt{UnopSEXP} \mid \texttt{BinopSEXP} \mid \texttt{ListSEXP} \mid \texttt{JSArraySEXP} \\ & \mid \texttt{JSBinopSEXP} \mid \texttt{JSClassSEXP} \mid \texttt{JSSpreadSEXP} \mid \texttt{JSTemplateSEXP} \mid \texttt{JSUnopSEXP} \\ & \mid \texttt{UNOPDelMemberExprSEXP} \mid \texttt{UNOPDelVarSEXP} \mid \texttt{AwaitSEXP} \\ & \mid \texttt{JSComputedFieldReadSEXP} \mid \texttt{JSComputedFieldWriteSEXP} \\ & \mid \texttt{JSPrivateFieldReadSEXP} \mid \texttt{JSPrivateFieldWriteSEXP} \\ & \mid \texttt{JSSuperFieldReadSEXP} \mid \texttt{JSSuperFieldWriteSEXP} \\ & \mid \texttt{PVTEnvReadSEXP} \mid \texttt{EnvReadSEXP} \mid \texttt{FieldReadSEXP} \mid \texttt{FieldWriteSEXP} \mid \texttt{EnvWriteSEXP} \\ & \mid \texttt{CallSiteSEXP} \mid \texttt{JSToObjectSEXP} \mid \texttt{JSCatchContextSEXP} \\ & \mid \texttt{JSADDBRANDSEXP} \mid \texttt{JSCheckConstructorSEXP} \\ & \mid \texttt{JSForInNextSEXP} \mid \texttt{JSForInStartSEXP} \\ & \mid \texttt{JSForOfIteratorCloseSEXP} \mid \texttt{JSForOfNextSEXP} \mid \texttt{JSForOfStartSEXP} \\ & \mid \texttt{JSAppendSEXP} \mid \texttt{JSCopyDataPropertiesSEXP} \mid \texttt{JSDefineObjMethodSEXP} \\ & \mid \{RVALTERMINAL\} \end{aligned}$$

2.2.4 RVAL Terminals

$$\begin{aligned} \{RVALTERMINAL\} ::= \ & \texttt{BooleanSEXP} \mid \texttt{LambdaSEXP} \mid \texttt{NullSEXP} \mid \texttt{NumberSEXP} \mid \texttt{RegExpSEXP} \\ & \mid \texttt{StringSEXP} \mid \texttt{JSBitIntSEXP} \mid \texttt{JSNUBDSEXP} \mid \texttt{JSObjectSEXP} \mid \texttt{JSPrivateSEXP} \end{aligned}$$

2.2.5 Statements (STMT)

$$\begin{aligned} \{STMT\} ::= \ & \texttt{StackRejectSEXP} \mid \texttt{StackRetainSEXP} \mid \texttt{GotoSEXP} \mid \texttt{IfElseJumpSEXP} \mid \texttt{IfJumpSEXP} \\ & \mid \texttt{InvokeFinalizerSEXP} \mid \texttt{PopCatchContextSEXP} \mid \texttt{PushCatchContextSEXP} \\ & \mid \texttt{RetSEXP} \mid \texttt{ReturnSEXP} \mid \texttt{ThrowSEXP} \\ & \mid \texttt{JSInitialYieldSEXP} \mid \texttt{ReturnAsyncSEXP} \mid \texttt{YieldSEXP} \\ & \mid \texttt{JSDefineObjPropSEXP} \mid \texttt{JSFuncDeclSEXP} \\ & \mid \texttt{JSImplicitBindingDeclarationSEXP} \mid \texttt{JSSloppyDeclSEXP} \end{aligned}$$

2.2.6 Basic Blocks

$$\texttt{BBSEXP} ::= \{STMT\}^*$$

A basic block is simply a sequence of statements. This structure makes Iridium amenable to traditional compiler optimizations like dead code elimination and block reordering, while preserving the JavaScript semantics via explicit node types.

2.3 Examples

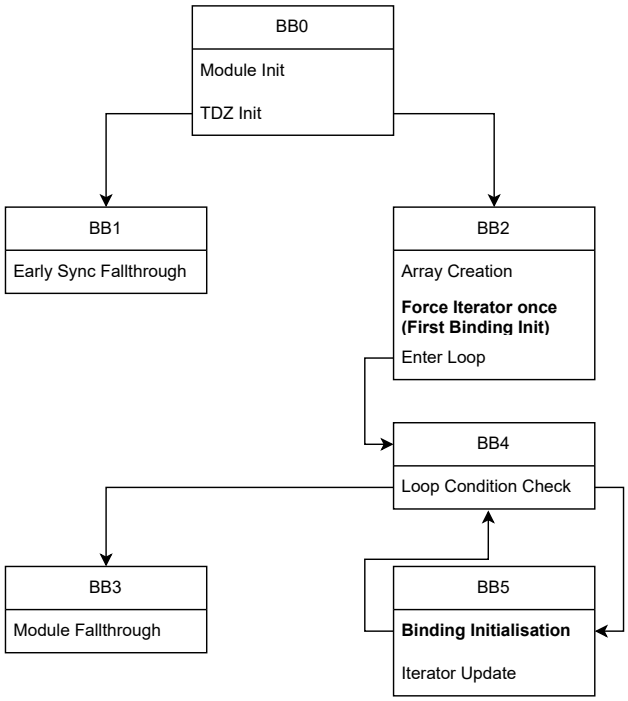
The full Iridium CFGs were moved to the appendix due to size constraints.↘

2.3.1 Array Destructuring

```
let [a, ...b] = [1,2,3,4];
```

Destructuring assignments allow developers to extract elements from arrays (or other iterable values) in a concise manner. However, despite the apparent simplicity of the source-level construct, the corresponding translation in Iridium is considerably more involved. The seemingly straightforward binding operation expands into a rich control-flow structure that includes the initialization of an iterator, explicit setup of catch contexts, and multiple error-propagation paths. These operational details, implicit in the JavaScript semantics, are made explicit in Iridium's representation (see Figure A.1 in the appendix).

Simplified View



2.3.2 Classes as Closures

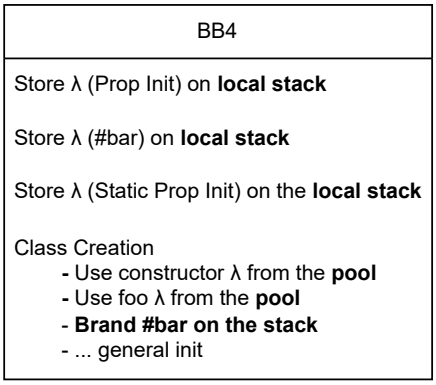
```
class A {
  #foo = 12
  #bar() {}
  foo()
  {
    let a = this.#bar;
    return a;
  }
}
```

In JavaScript, classes and functions are technically the same underlying construct. While the `class` syntax provides a convenient abstraction for developers, its runtime behavior relies on a collection of implicit closures and initialization mechanisms. Iridium makes these otherwise hidden structures explicit, enabling precise analysis and transformation. Specifically, Iridium introduces the following closure forms:

1. **Class Initialization Object.** When a class is created, an initialization object sets up closure properties such as the `home_object`, prototype chain, and branding metadata (see Figure A.2 in the appendix).
2. **Constructor Closure.** The constructor function is not a conventional closure over its defining scope. Instead, it references initializer closures that provide access to the enclosing scope. While this design appears to be influenced by QuickJS, Iridium adopts the same convention to ensure consistency (see Figure A.3 in the appendix).
3. **Property Initialization Closure (`#foo = 12`).** Instance property initialization executes in a dedicated closure where special bindings exist: `this` points to the instance under construction, and the class name is directly available (see Figure A.4 in the appendix).
4. **Static Initialization Closure.** Static properties are initialized in a separate closure where both `this` and the class name are re-bound as aliases, reflecting the ECMAScript specification (see Figure A.5 in the appendix).
5. **Private Method Closures.** Private methods (e.g., `#bar`) are stored on the declaring stack frame rather than the prototype. A branding mechanism enforces correct access semantics, preventing unauthorized calls (see Figure A.6 in the appendix).
6. **Public Method Closures.** Public methods (e.g., `foo`) are explicitly attached to the class prototype, thereby ensuring consistent inheritance and dynamic dispatch (see Figure A.7 in the appendix).

Taken together, these six closure forms illustrate how JavaScript classes are little more than syntactic sugar layered over a system of hidden functions and environments. While standard JavaScript engines implicitly manage these mechanisms, Iridium *reifies* them as explicit entities in its intermediate representation. This not only exposes the precise control-flow and scoping behavior of classes, but also provides a uniform substrate for advanced static analyses and transformations that were previously difficult to express.

Simplified View



Note

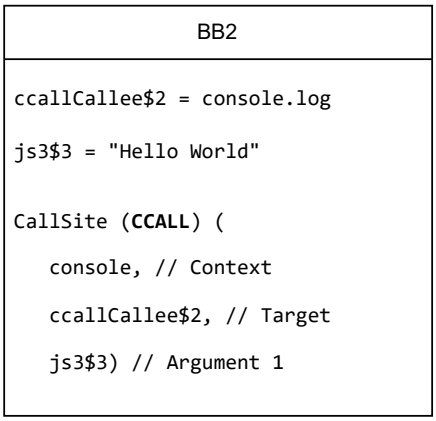
Some closures are closed inside the current execution scope while some closures are loaded from the pool (refers to the constant pool of the current scope). This separation is needed to prevent invalid lexical lookups during instantiation.

2.3.3 Contextual Calls

```
1 console.log("Hello World");
```

In JavaScript, function invocation is not always a simple matter of passing arguments. Every call site may carry with it an implicit *context*, such as the global object, a bound `this` value, or a lookup scope derived from the call expression. These contexts influence the semantics of the call, often in subtle ways that complicate static reasoning. For instance, calling a method through `console.log` implicitly sets the receiver of the call to be the `console` object. Iridium makes such contexts explicit: each call site in the intermediate representation is annotated with its associated context, ensuring that analyses and transformations can account for these effects in a uniform manner (see Figure A.8 in the appendix).

Simplified View



2.4 Documentation Notes

A detailed description of all Iridium nodes is available on the project's GitHub pages: Iridium Documentation. This section highlights selected examples for illustration.

2.4.1 JSForOfStartSEXP

For a given an object, this call stores it's For-Of on the stack.
`[for_of_iterator,loop_method,catch_offset = 3] = JSForOfStartSEXP(RVal)`
The reason for not popping the stack is the presence of the custom catch handler which can break if stack is restructured. Maybe some analysis passes can simplify this logic in the future.

Node Structure

- ARG(obj): The object instance.

2.4.2 CallSiteSEXP

Represents a call made to a closure. First N arguments are used to form the calling context, the value of N is 1 by default. Depending on the marked flag (see CallSiteSEXPFlags) different values of N are selected. Also, when calling a closure, different calling conventions might apply, these are broadly classified into the following three categories:

- Basic Closure Application: No flag set, N = 1 by default i.e. the first argument is the callee object. The rest of the args are passed as arguments to the callee.
- Contextual Closure Application:
  - CCall: N = 2, the first two args form the calling context (first arg is mapped to the this object and the second arg is callee). The rest of the args are passed as arguments to the callee.
  - PrivateCall: N = 2, same as CCall but an additional brand check is needed before actual execution.
- Constructor Special Closure Application: This kind of call expects two objects to form the calling context, the callee object and a new_target special object.
  - ConstructorCall: N = 1, (even though N = 1, the first argument is duplicated before the actual call). The rest of the args are passed as arguments to the closure.
  - super: N = 2. The rest of the args are passed as arguments to the closure.

Node Structure

- ...ARG(...callContext, ...args): The first N args are used form the calling context while the rest are mapped to formals.
- FLAG(CCall): Contextual Call where this binding is provided by the call site.
- FLAG(ConstructorCall): Contextual Call where this binding is provided by the call site.
- FLAG(PrivateCall): Contextual Call with added brand check.
- FLAG(Import): Dynamic import call.
- FLAG(Super): Call to the parent class constructor.
- FLAG(V8Intrinsic): Call to a V8 Intrinsic.
- FLAG(JSDirectEval): Marks the call as direct eval, i.e. in sloppy mode the evaled code can modify the enclosing environment.

2.4.3 BBContainerSEXP

BBContainer represents a logical piece of compilable code, it consists of two parts. The first arg contains a BindingsSEXP which is used to create and populate a stack frame. The second arg contains a ListSEXP of type BBSEXP which contains the instructions to execute.

Node Structure

- ARG(bindings): Stores a BindingsSEXP.
- ARG(BB): Stores a ListSEXP of type BBSEXP.
- FLAG(ECMAArgs): Used to set the length property for functions, which indicates the number of expected arguments as per the ECMAScript spec.
- FLAG(StartBBIDX): IDX of the entry BB.
- FLAG(ScopeIDX): Scope/environment IDX of top level scope for this container.
- FLAG(ARGUMENTS): Is the arguments object allowed in this scope?
- FLAG(ASYNC): Is this an asynchronous function/module?
- FLAG(STRICT): Is the code in strict mode?
- FLAG(GENERATOR): Is this a generator function?
- FLAG(PROTO): Does the function have a prototype?
- FLAG(NEW): Is new.target lookup allowed in this function?
- FLAG(SCALL): Is super call allowed in this function?
- FLAG(SOBJ): Is super object allowed in this function?
- FLAG(HOME): Does this function need the home object?
- FLAG(DERIVED): Is this a derived constructor function?
- FLAG(TopLevel): Is top Level?
- FLAG(ContainerFlagID): A number.

Closure Type Map

- 1. Regular Closure =
- 2. Constructor = PROTO, NEW
- 3. Derived Constructor = PROTO, NEW, SCALL, SOBJ, HOME, DERIVED
- 4. Derived Method = SOBJ, HOME
- 5. Private Method = HOME
- 6. Prop Init + no_private =
- 7. Prop Init Derived + no_private = SOBJ, HOME
- 8. Prop Init + private = HOME
- 9. Prop Init Derived + private = SOBJ, HOME
- 10. Private Derived Method = SOBJ, HOME
- 11. Static Prop Init =
- 12. Static Prop Init Derived = SOBJ, HOME

Chapter 3

Architecture

The Iridium framework is designed as a modular toolchain for translating JavaScript programs into a uniform intermediate representation (IR) that enables static analysis and transformation. The architecture, shown in Figure 3.1, is built around a series of cooperating components that progressively refine the representation of JavaScript from raw source to analyzable control-flow graphs.

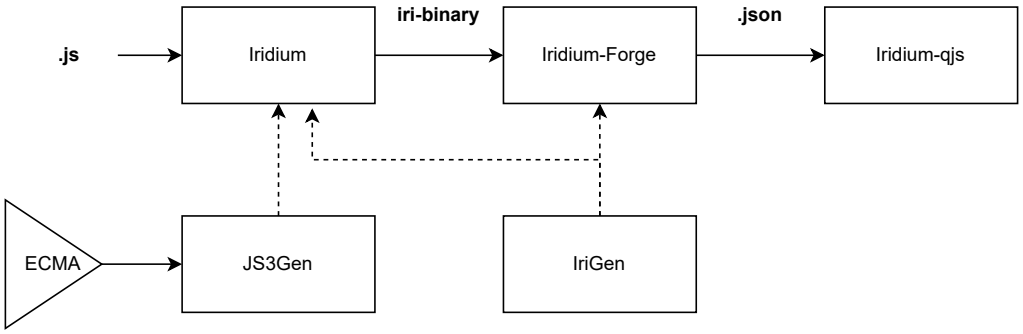


Figure 3.1: Overview of the Iridium architecture.

3.1 Iridium

The *Iridium* frontend is the entry point of the system. It accepts raw JavaScript source files (`.js`) and translates them into a normalized representation based on Iridium S-expressions (SEXP). This step is responsible for desugaring higher-level constructs, enforcing a uniform node structure (tag, arguments, and flags), and ensuring that control-flow and data-flow edges are made explicit.

The output of this stage is serialized into the `iri-binary` format, a compact binary encoding of the S-expression tree suitable for further processing.

3.2 Iridium-Forge

Iridium-Forge is the central compiler backend. It consumes the `iri-binary` format and generates analyzable and optimizable intermediate representations. Forge applies structural transformations, performs scope resolution, inserts environment bindings, and constructs the explicit control-flow graph (CFG).

The processed representation is exported as `.json`, which is both human-readable and suitable for downstream tools.

3.3 Iridium-qjs

The *Iridium-qjs* module consumes the JSON IR and executes it on top of a modified QuickJS runtime. This provides a reference interpreter for Iridium programs, enabling validation of transformations and comparison with native JavaScript execution.

3.4 JS3Gen

JS3Gen is an auxiliary project that bridges the ECMA specification and Iridium. It parses the ECMAScript standard and automatically generates support classes that correspond to JavaScript constructs. These classes are used by Iridium when lowering source code into the S-expression form.

By automating this process, JS3Gen reduces the manual effort of encoding the complex JavaScript semantics into the IR.

3.5 IriGen

IriGen is a code generation utility that works closely with Iridium-Forge. It consumes the language specification (in JSON format) and automatically generates parser classes in C++. This ensures that the IR is not only consistent with the specification but also easily extensible as JavaScript evolves.

Currently, IriGen targets C++ backends, with future work extending support to TypeScript for tighter integration with web tooling.

3.6 Workflow Summary

In summary, the Iridium pipeline can be described as:

1. JavaScript source (`.js`) is fed into Iridium, producing `iri-binary`.
2. Iridium-Forge consumes `iri-binary`, applies transformations, and emits `.json`.
3. Iridium-qjs executes `.json` on a QuickJS-based runtime.
4. JS3Gen and IriGen provide specification-driven automation, ensuring semantic fidelity and reducing manual encoding effort.

Together, these components establish a modular, analyzable, and executable framework for JavaScript programs.

3.7 Project Breakdown

Table 3.1 provides an overview of the main projects that constitute the Iridium ecosystem, including their size, implementation language, and development status.

Project	**LOC**	**Languages**	**Status**
Iridium	∼50K	TypeScript	Stable, WIP
JS3Gen	∼1K	TypeScript	Mostly Stable
Iridium-Forge	∼9K	C++	WIP
IriGen	∼200	TypeScript	Mostly Stable
Iridium-qjs (patch)	∼5K	C/C++	WIP, one stable release

Table 3.1: Breakdown of Iridium projects by size, language, and stability.

Chapter 4

Core Transformation Passes

The translation of JavaScript into Iridium S-expressions (SEXP) produces a semantically rich but non-executable form of intermediate code. In this representation, language constructs are explicit but not yet operational: bindings are unresolved, scopes are uninitialized, and control-flow is only partially explicit. To bridge this gap, Iridium applies a series of *core transformation passes* that progressively refine and resolve the program until it is executable. This process mirrors the role of early and mid-level passes in traditional compilers such as LLVM or Soot, but must contend with the additional complexity of JavaScript's dynamic typing, non-lexical scoping rules, and runtime meta-programming facilities.

Originally, these passes were implemented in the TypeScript frontend. While effective on small examples, the frontend quickly became too slow and memory-intensive when analyzing large-scale applications. For example, resolving closures across thousands of modules required walking millions of nodes repeatedly. To overcome these challenges, the passes were migrated into `Iridium-Forge`, a high-performance C++ backend. This provided both speedups (by orders of magnitude on large programs) and finer-grained memory management, enabling Iridium to scale to real-world applications.

4.1 Core Transformation Pipeline

The transformation passes in Iridium operate sequentially, with each pass establishing structural invariants required by the next. Although conceptually distinct, the passes work in concert: scope construction depends on hoisting, closure grouping depends on scope structure, and stack frame initialization depends on closure arguments. Together, they enforce three key invariants:

1. **Executable Control Flow:** Every branch, loop, and function boundary has explicit entry and exit points.
2. **Resolved Bindings:** Every identifier in the program maps to an explicit environment location (local, captured, or imported).
3. **Initialized Scopes:** Every function or module scope has a well-defined stack frame structure prior to execution.

4.1.1 "Making Iridium Executable" Passes

These initial passes ensure that the intermediate representation is well-formed and directly interpretable. They form the foundation upon which later, more sophisticated analyses and transformations operate.

1. **normalizeBBFlags** – Ensures all basic blocks (BBs) within a scope are flagged consistently. Without this, later grouping or control-flow passes may treat some blocks as unreachable or uninitialized.
2. **hoistFunctionDeclarations** – Relocates function declarations to their correct lexical scope, preserving JavaScript's hoisting semantics. This step guarantees that later references to functions will resolve correctly.

3. **filterNOPs** – Cleans up redundant `NOP` instructions introduced during transformation. While semantically harmless, these clutter the IR and impede closure grouping.
4. **groupIntoClosureGroups** – Identifies sets of BBs that must be compiled as closures. For example, nested functions, generators, and async blocks are grouped here, ensuring correct capture of free variables.
5. **populateModuleBindings** – Resolves top-level module structure. Imports, exports, and namespace bindings are rewritten into explicit forms so that the runtime knows which bindings are local and which come from external modules.
6. **populateImplicitBindings** – Adds implicit bindings such as `arguments`, `this`, or hidden runtime structures. These are often omitted by source syntax but required for correct runtime behavior.
7. **reduceFunctionDeclarations** – Simplifies functions when possible. For example, trivial lambdas or anonymous closures are lowered to constant pool entries, avoiding unnecessary stack allocation.
8. **populateExplicitBindings** – Inserts user-declared bindings (variables, constants, classes) into the environment table.
9. **addClosureArgsBindings** – Handles function parameters and the special `arguments` object. This pass also decides whether the `arguments` object is *mapped* (aliasing parameters) or *unmapped* (a copy), depending on strictness.
10. **initializeStackFrame** – Allocates and initializes the stack frame layout, including slots for bindings, temporaries, and hidden runtime objects.
11. **patchHeritageConstructorSuperCalls** – Augments `super()` calls in class constructors with initialization of `this`. This prevents illegal reinitialization and ensures ES6 inheritance semantics.
12. **reduceResolvePrivateEnvBindingSEXP** – Rewrites private field and method references into explicit symbol accesses, capturing JavaScript's private branding semantics.
13. **reduceResolveEnvBindingSEXP** – Resolves standard environment bindings to stack objects, ensuring uniqueness (singletons) across the program.
14. **resolveLambdaTargets** – Moves lambdas into the constant pool, enabling reuse and reducing runtime allocation overhead.
15. **resolveBreakAndContinueTargets** – Annotates loop structures with explicit targets. In complex cases (e.g., nested `for-of` within `try-finally`), decorators are inserted to preserve balanced stack unwinding.
16. **decorateReturnTargets** – Ensures `return` statements have explicit exit decorators, particularly for closures where control may cross scope boundaries.
17. **promoteAsyncReturns** – Converts `return` into `async return` where appropriate, enabling correct suspension and resumption semantics.
18. **markNamespaceImports** – Marks bindings imported as namespaces, ensuring correct handling when initializing runtime frames.
19. **markSloppyWrites** – Identifies writes that occur under sloppy mode (non-strict JavaScript) and marks them for special runtime treatment.
20. **loosenWritestoASWs** – Relaxes writes to *Always Safe Write* (ASW) bindings, allowing the optimizer to skip redundant runtime checks.
21. **markDirectEvals** – Detects calls to `eval` that taint the current scope. The pass updates binding indices to reflect which scopes may be dynamically altered.

4.1.2 Optimization Passes

Once the core pipeline guarantees executability, additional passes can streamline the IR. With fidelity to JavaScript semantics established, we plan to explore opportunities for new analyses and optimizations. In JavaScript, even small optimizations can significantly reduce runtime cost. Currently, the key optimization implemented is:

- **doUnreadBindingRemoval** – Removes bindings that are never read from and do not belong to a tainted scope. For example, unused loop counters or temporaries introduced during lowering can be safely discarded, shrinking stack frames and reducing runtime allocations.

Future optimization passes under consideration include constant folding of numeric expressions, dead code elimination in untainted scopes, and inlining of trivial lambdas.

4.2 Chapati

During CFG construction, Iridium adopts a canonical form where each basic block (BB) explicitly marks a goto destination and disallows any fallthroughs. While this representation makes reasoning about control flow simple and avoids ambiguity, directly emitting such code for runtime is expensive: the generated executable contains redundant jumps, artificial block splits, and unnecessary bookkeeping. To address this, we implement a flattening pass — informally called *Chapati* — which heuristically merges compatible BBs into larger blocks right before emitting the final executable.

Algorithm 1: Chapati: block flattening with height heuristic

```
   Input: CFG G, entry vertex e
   Output: Flattened list of BBs (in traversal order)
 1 H ← ComputeHeights(G)                          // precompute continuation heights
 2 Processed ← ∅
 3 Function Flatten(u)
 4     if u ∈ Processed then
 5         return
 6     Processed ← Processed ∪{u}
 7     Succ ← successors(u)
 8     MergeCand ← { v ∈ Succ | SafeToMerge(u, v) }
 9     if MergeCand ≠ ∅ then
10         v* ← arg max_{v∈MergeCand} H[v]
                                                  // choose best continuation
11         MergeBlocks(u, v*)
                                                  // merge v* into u (update CFG)
12         Processed ← Processed \{u}
                                                  // allow re-processing u
13         Flatten(u)
                                                  // try more merges on the new, larger u
14     else
15         foreach v in Succ do
16             Flatten(v)
17 Function Chapati(G, e)
18     Flatten(e)
19     return list of BBs by performing a final traversal of G (entry-order)
```

The Chapati algorithm relies on two ideas:

- **Safe Merging:** Two blocks can be merged if the successor has exactly one predecessor and the predecessor ends in a direct transfer of control to it (e.g., a `goto`).
- **Height Heuristic:** Among multiple merge candidates, the successor with maximal continuation height is chosen, reducing nesting and flattening the CFG (see Line 10).

The effect of applying this algorithm is shown in Figure A.9 (before) and Figure A.10 (after) (in the appendix).

4.3 Discussion

The transformation passes are the backbone of Iridium's compilation pipeline. They establish a faithful, executable IR that preserves JavaScript semantics while enabling efficient execution on backends such as Iridium-qjs. Compared to traditional compilers, the challenge lies less in optimization and more in enforcing language invariants: correctly hoisting, resolving dynamic

scoping, and decorating control flow. By migrating these passes from TypeScript into a C++ implementation, Iridium has reached a point where large-scale programs can be transformed and analyzed within practical time and memory limits. This scalability is a key enabler for both static analysis research and future compiler optimizations.

Chapter 5

Conclusion and Future Work

In this work, we introduced Iridium, an intermediate representation and toolchain that bridges the gap between JavaScript's dynamic flexibility and the requirements of static analysis and transformation. By explicitly modeling control flow, scoping, bindings, and closures in an S-expression IR, Iridium makes JavaScript analyzable, executable, and transformable in ways previously limited to statically typed languages. Our key contributions include a uniform node structure that facilitates serialization across TypeScript and C++, a modular architecture that cleanly separates frontend parsing, backend transformations, and runtime execution, and a comprehensive suite of C++ passes that resolve JavaScript semantics to enable analyzable yet faithful execution. We also introduced a heuristic CFG-flattening pass ("Chapati") that improves runtime efficiency and built the first working prototype of a reference interpreter (`Iridium-qjs`) over the completed IR. Together, these building blocks establish the foundation for a new generation of JavaScript tools—static analyzers, optimizers, instrumentation frameworks, and security utilities that previously lacked suitable IR infrastructure.

Looking forward, several avenues present themselves. Beyond initial optimizations such as unread-binding elimination, Iridium can benefit from more advanced compiler techniques, including constant folding, loop-invariant code motion, and JIT-oriented specialization. Coupling Iridium with gradual or inferred typing systems could enable type-aware transformations, while its JSON-based IR offers opportunities for cross-language targeting, such as translation into WebAssembly for performance portability. More broadly, the architecture appears general enough to extend beyond JavaScript to other dynamic languages like Python or Ruby, potentially unifying analyzability across multiple ecosystems. Over the course of this thesis, we expect Iridium to demonstrate that, despite the challenges of highly dynamic semantics, systematic static representation and transformation are not only possible but practical, and we invite researchers and developers to build upon this foundation to advance the next generation of language tooling.

Appendix A

Iridium CFGs

Figure A.1: Figure showing the CFG for destructuring array assignments.

Figure A.2: Top level CFG, representing the declaring scope for the class object.

```
BB6
BB[IDX=6, ScopeIDX=4, ClosureBoundary=null] {
 JSImplicitBindingDeclaration[NAME="this", JSCONST=null, OPID=9, THISINIT=false, SAFE=true, SKIPINIT=null](EnvBinding[NAME="this"],
   List {
   }
 )
 StackReject[NVAL=0](JSCheckConstructor)
 StackReject[NVAL=1](CallSite[CCall=null](EnvRead(EnvBinding[NAME="this"]), EnvRead(RemoteEnvBinding[REFIDX=0](RemoteEnvBinding[REFIDX=2](EnvBinding[NAME="PropInitClosure$5"])))))
 Return(GlobalBinding[NAME="undefined"])
}
```

Figure A.3: Constructor closure, a default constructor is created if it was not previously defined by the user.

```
BB5
BB[IDX=5, ScopeIDX=3, ClosureBoundary=null] {
 EnvWrite[SAFE=true, THISINIT=false](EnvBinding[NAME="js3$6"], JSNUBD)
 JSImplicitBindingDeclaration[NAME="this", JSCONST=null, OPID=9, THISINIT=false, SAFE=true, SKIPINIT=null](EnvBinding[NAME="this"],
   List {
   }
 )
 JSImplicitBindingDeclaration[NAME="<home_obj>", JSCONST=null, OPID=4, THISINIT=false, SAFE=true, SKIPINIT=null](EnvBinding[NAME="<home_obj>"],
   List {
   }
 )
 EnvWrite[SAFE=true, THISINIT=false](EnvBinding[NAME="js3$6"], Number[IridiumPrimitive=12])
 StackReject[NVAL=1](JSPrivateFieldWrite[DECL=null](EnvBinding[NAME="this"], EnvRead(RemoteEnvBinding[REFIDX=0](RemoteEnvBinding[REFIDX=0](EnvBinding[NAME="js3$3"]))), EnvRead(EnvBinding[NAME="js3$6"])))
 StackReject[NVAL=0](JSADDBRAND(EnvRead(EnvBinding[NAME="this"]), EnvRead(EnvBinding[NAME="<home_obj>"])))
 Return(EnvRead(GlobalBinding[NAME="undefined"]))
}
```

Figure A.4: Property initialization closure

```
BB13
BB[IDX=13, ScopeIDX=9, ClosureBoundary=null] {
 JSImplicitBindingDeclaration[NAME="this", JSCONST=null, OPID=9, THISINIT=false, SAFE=true, SKIPINIT=null](EnvBinding[NAME="this"],
   List {
   }
 )
 StackReject[NVAL=1](EnvRead(EnvBinding[NAME="this"]))
 Return(EnvRead(GlobalBinding[NAME="undefined"]))
}
```

Figure A.5: Static property initialization closure

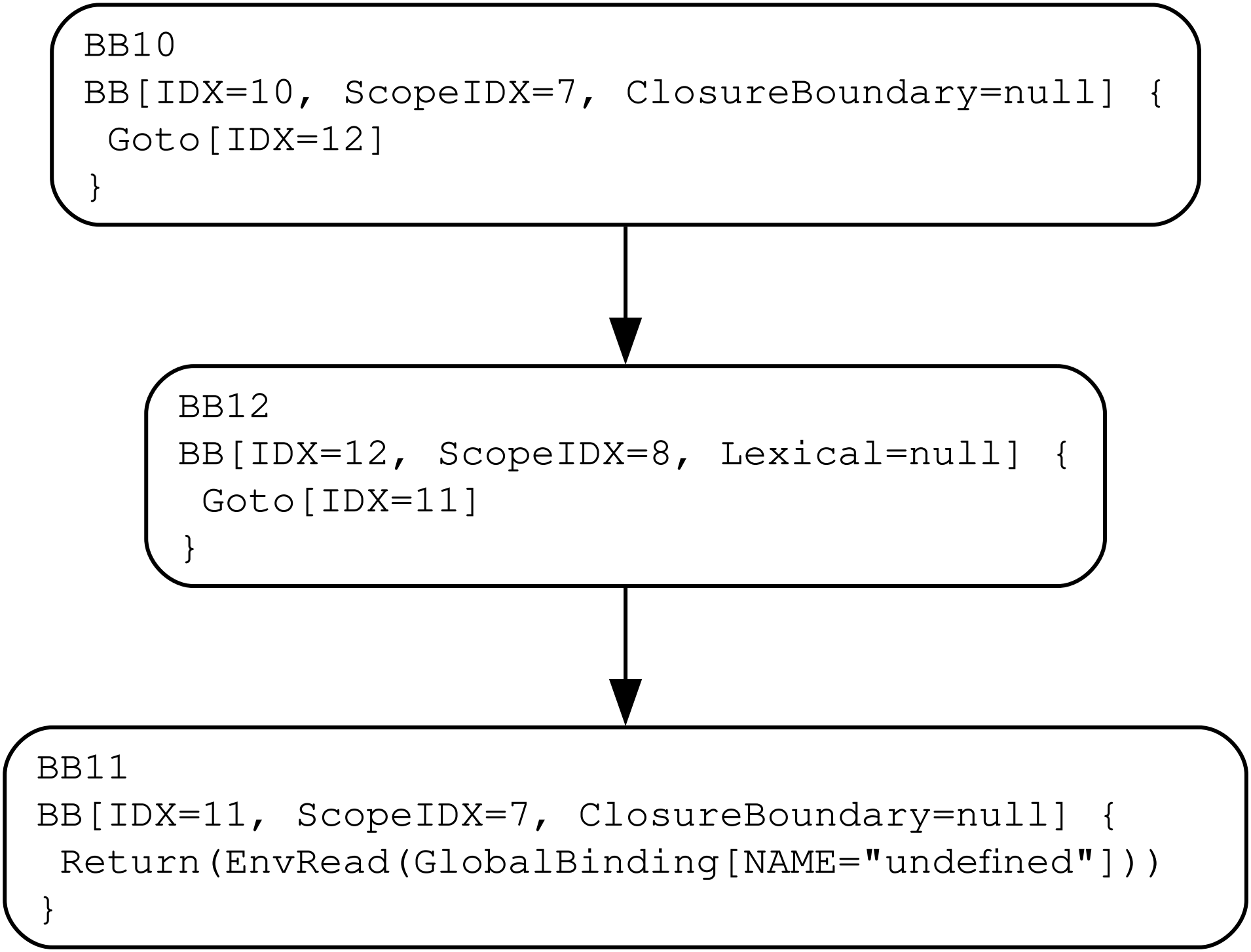


Figure A.6: Private closure `#bar`

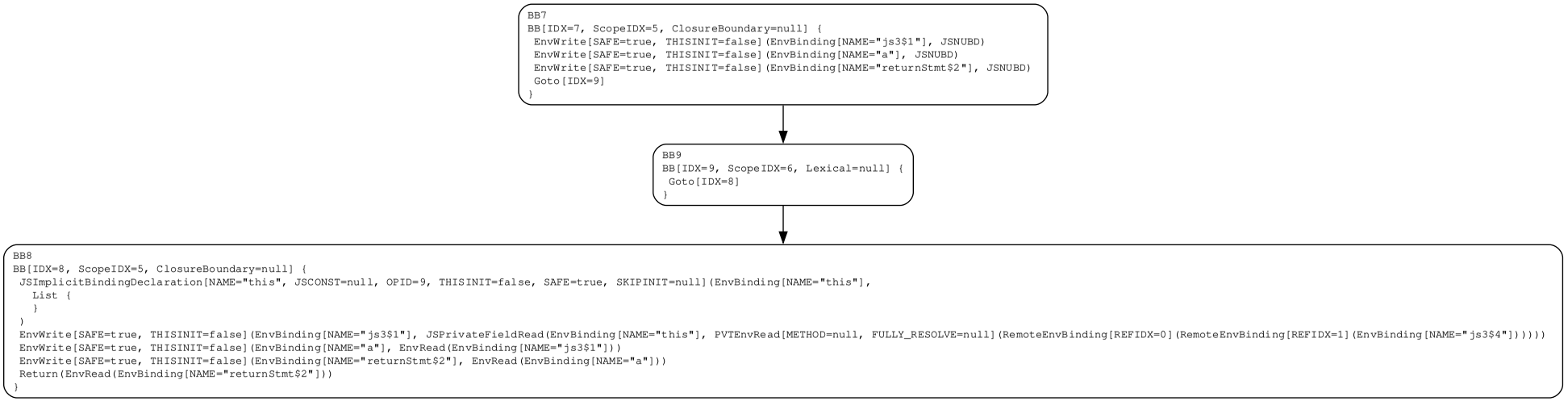


Figure A.7: Public closure `foo`

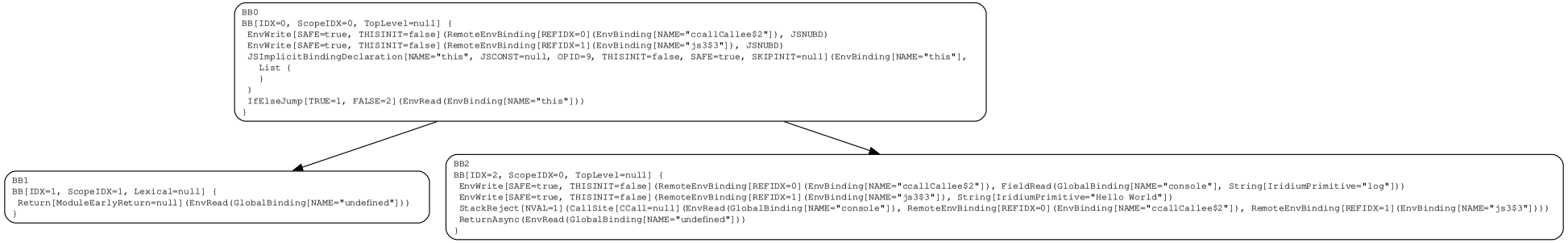


Figure A.8: Contextual call expression

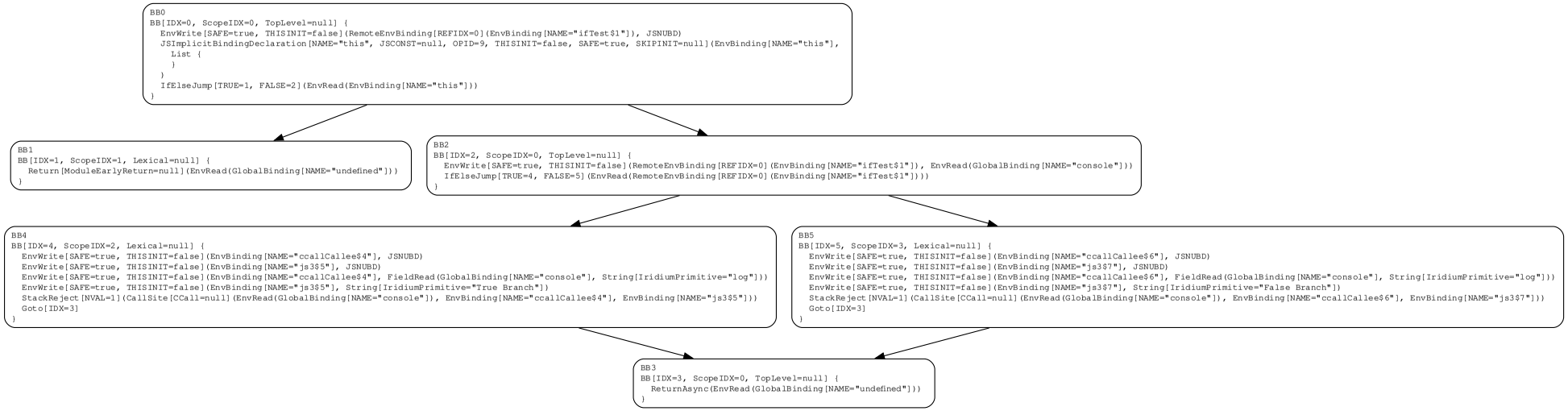


Figure A.9: Before chapati

Figure A.10: After chapati

References

[1] Iridium0.9a. https://compl-research.github.io/Iridium-Docs/. Accessed: 2025-09-10.

[2] Jetstream2.2. https://browserbench.org/JetStream/. Accessed: 2025-09-10.

[3] quickjs-ng/benchmarks. https://github.com/quickjs-ng/benchmarks. Accessed: 2025-09-10.

[4] test262.fyi. https://test262.fyi/. Accessed: 2025-09-10.

[5] Vincenzo Arceri and Isabella Mastroeni. Static program analysis for string manipulation languages. *Electronic Proceedings in Theoretical Computer Science*, 299:19–33, August 2019.

[6] Olivier Flückiger, Guido Chari, Jan Ječmen, Ming-Ho Yee, Jakob Hain, and Jan Vitek. 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*, DLS 2019, page 55–66, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[8] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023.

[9] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: dynamic type inconsistency analysis for javascript. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 314–324. IEEE Press, 2015.

[10] Chris Seaton. Understanding graal ir (invited talk). In *Proceedings of the 12th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, VMIL 2020, page 3, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: a java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, page 214–224, USA, 2010. IBM Corp.