Type-Aware Optimizations with Imperfect Types

by

Jeremiah Ikosin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

JavaScript, a programming language originally designed for web browsers, has become ubiquitous, experiencing adoption across multiple platforms. Its dynamic type system and prototype-based object orientation are well-known properties that make the language applicable to several programming paradigms, particularly functional and object-oriented programming. However, issues such as global scope pollution, implicit type conversion, the absence of native null safety features, and the complexities of asynchronous callback structures, among others, make the language difficult to work with. To address these challenges, particularly within the context of large-scale application development, TypeScript was introduced.

TypeScript incorporates a structural type system and compiles to JavaScript. The design objective is to ensure seamless interoperability with JavaScript, incorporating various ergonomic features, notably static typing. TypeScript introduces improved tooling, IDE support, ES6 features with extensions, and compatibility with existing JavaScript code. Despite these advantages, TypeScript deliberately refrains from optimizing its JavaScript output. Although JavaScript's flexibility can often be useful in practice, a naive implementation of the language would be slow. Modern JavaScript engine implementations are intricate systems that employ cutting-edge optimization techniques to achieve efficient executions.

This thesis introduces a method for improving the runtime performance of JavaScript by utilizing type information from TypeScript. It categorizes TypeScript types based on usage into two groups: nominal (similar to classes in Java) and non-nominal (structural or arbitrary). Although TypeScript's type system is inherently unsound, types tend to be consistent in most nominal use cases. This characteristic renders a significant proportion of type information amenable to optimization with reasonable guarantees.

I modified the TypeScript compiler (`tsc`) to leverage nominal type usage for optimizations. This modification produces optimized code through the utilization of enhanced heuristics for runtime optimizations. Additionally, I integrated WebKit's JavaScript engine, JavaScriptCore (JSC), by introducing a new runtime intrinsic specifically designed to utilize type information from TypeScript.

Performance is assessed by comparing JavaScript programs from the JetStream 2.1 JavaScript test suite with equivalent programs ported to TypeScript. These TypeScript programs are then compiled to JavaScript using the modified TypeScript compiler in two modes: with optimizations enabled and with optimizations disabled. The results show that adopting a nominal typing style in TypeScript leads to improved performance in the resulting JavaScript when compiled with optimizations enabled, by up to 12%.

Acknowledgements

I am thankful to God Almighty, whom I call Father, for guiding me through every step of the journey. I am grateful to my parents for their endless love and support throughout my education. I am grateful to my advisor, Gregor Richards, for his unwavering patience and support throughout my program. I am grateful for the support from friends and family, the ones with whom I share timeless and excellent connections. I am grateful for the McEacherns, the Acorns, the Stewarts, the wonderful people of Bethel Chapel, and the excellent folks from the Thursday Night Study Group. Of course, this page would be incomplete without mentioning friends who have stuck closer than a brother. I am ever grateful for Thomas, Smit, and Andrew. May our stars ever shine so brightly.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Contributions . . . 2

1.2 Overview . . . 3

2 Background 5

2.1 JavaScript . . . 5

2.1.1 JavaScript Syntax and Semantics . . . 5

2.1.2 Dynamic Typing . . . 8

2.2 JavaScript Engines . . . 9

2.2.1 Optimization Techniques and Challenges . . . 10

2.3 JavaScriptCore . . . 11

2.3.1 Execution Pipeline Architecture . . . 13

2.3.2 Bytecode . . . 13

2.3.3 Fast Path / Slow Path . . . 15

2.3.4 OSR . . . 15

2.3.5 LLInt . . . 16

2.3.6 Baseline JIT . . . 18
2.3.7 DFG and FTL JIT . . . 18
2.3.8 Optimization Techniques . . . 19
2.3.9 Speculation . . . 23
2.4 Bun . . . 23
2.5 TypeScript . . . 24
2.5.1 TypeScript Syntax and Semantics . . . 24
2.5.2 Type System . . . 27
2.6 Nominal Typing . . . 28
2.7 Why TypeScript? . . . 29

3 Implementation **30**
3.1 Heuristic Guided Optimization . . . 30
3.2 Type Classification . . . 34
3.3 Heuristics . . . 35
3.4 Case Study: Nominal Types . . . 36
3.4.1 Explicit Type Casts . . . 36
3.4.2 Implicit Type Casts . . . 39
3.4.3 Delete Operations . . . 40
3.5 Modifications . . . 41
3.6 TypeScript . . . 41
3.6.1 Checking Nominality . . . 41
3.6.2 Optimization Metadata . . . 45
3.6.3 Shape Mutation . . . 48
3.6.4 Emitting JavaScript . . . 49
3.6.5 `tsconfig.json` Configuration . . . 52
3.7 JavaScriptCore . . . 52
3.7.1 `get_by_id_offset` Bytecode Instruction . . . 52
3.7.2 LLInt . . . 53
3.7.3 Baseline . . . 54
3.7.4 Intrinsic Instructions . . . 56
3.8 Bringing it All Together . . . 59
3.8.1 A Complete Example . . . 59

4 Evaluation **62**
4.1 Benchmarks . . . 62
4.1.1 Microbenchmarks . . . 65
4.2 Methodology . . . 65
4.3 Results . . . 66
4.3.1 B_{jet} . . . 66
4.3.2 B_{μ} . . . 69
4.3.3 Intrinsics . . . 71
4.4 Discussion . . . 72
4.4.1 B_{jet} . . . 72
4.4.2 B_{μ} . . . 73
4.4.3 Heuristics . . . 74
4.5 Closing Remarks . . . 75

5 Related Work **77**
5.1 JavaScript Performance and Optimization Challenges . . . 77
5.2 Performance Optimization Techniques . . . 79
5.2.1 Concrete Types for TypeScript . . . 80
5.2.2 Leveraging Property Access Optimization in V8 . . . 81
5.2.3 Typed JavaScript . . . 82

6 Future Work **84**

7 Conclusions **86**

References **88**

APPENDICES **96**

A `Symbol` and `PropertyAccessExpression` **97**

B Nominal Type Properties for Object Layout **99**

C `tsconfig.json` support for `optimizeWithTypes` **100**

D `Builtin` mode in JavaScriptCore **102**

List of Figures

2.1 A simple JavaScript function. . . . 6
2.2 Objects in JavaScript. . . . 6
2.3 Creating objects using constructors in JavaScript. . . . 7
2.4 Classes in JavaScript. . . . 8
2.5 Dynamic typing in JavaScript. . . . 8
2.6 A simple JavaScript engine architecture. . . . 9
2.7 A JIT triggering example [48]. . . . 10
2.8 The tiers of JavaScriptCore [15]. . . . 12
2.9 JavaScriptCore architecture [15]. . . . 12
2.10 Example timeline of a simple long loop executing in JavaScriptCore [15]. . . . 13
2.11 A simple JavaScript program compiled to bytecode. . . . 14
2.12 Hypothetical fast path / slow path implementation of an add instruction. . . . 15
2.13 A high-level view of LLInt. . . . 16
2.14 `mov` and `to_string` implementation in LLInt, defined in LowLevelInterpreter64.asm [10] 17
2.15 Objects and structures in JavaScriptCore [15]. . . . 20
2.16 Property access in JavaScript. . . . 21
2.17 Memory layout of an object in JavaScriptCore [15]. . . . 21
2.18 An interface in TypeScript. . . . 24

3.1 JavaScript class with two properties demonstrating property access. . . . 31
3.2 Layout of `obj` in JavaScriptCore. . . . 31
3.3 Regular property access as seen in `obj.x`. . . . 32
3.4 Optimized property access for `obj.x`. . . . 32

3.5 Example of dynamic property assignment. 33
3.6 Deleting an object's property in JavaScript. 33
3.7 Deleting an object's property in TypeScript. 34
3.8 Classification of types according to usage. 35
3.9 Direct property deletion through cast to `any`. 37
3.10 Indirect property deletion through cast to `any`. 37
3.11 Structural type casts in TypeScript. 37
3.12 Class and interfaces with similar types. 38
3.13 Type coercion to `any` through assignment. 38
3.14 Type coercion in function parameters. 39
3.15 Type coercion in assignment. 40
3.16 Deleting the property `y` of the `Point` object `pt`. 40
3.17 Pseudocode for type checking expressions. 42
3.18 Pseudocode for checking if a property access is within a constructor. 43
3.19 |this— property access in a class constructor. 43
3.20 Derivation of `satisfiesClsTy`. 43
3.21 Pseudocode for deriving `isEligibleClsTy`. 45
3.22 Checking for flagged types. 45
3.23 Layout, flagged types, and maximum inline slots definitions. 46
3.24 Implementation of `getOffset`. 46
3.25 Pseudocode for attaching layout and offset metadata to a property access node. 47
3.26 Decomposition of property access expressions. 48
3.27 Pseduocode for detecting shape mutation through deletion. 48
3.28 class `Point` with field initializations matching computed layout information. 49
3.29 Pseudocode for emitting a constructor with extension for layout metadata. 50
3.30 Transformation of class fields to object layout instructions. 50
3.31 Pseudocode for emitting property access expressions. 51
3.32 Transformation of a property access expression to an intrinsic function call. 51
3.33 Definition of `get_by_id_offset` opcode format. 54
3.34 Implementation of `get_by_id_offset` in LLInt. 54

3.35 Pseudocode for `emit_op_get_by_id_offset` in Baseline. 55
3.36 Slow path implementation of `op_get_by_id_offset` in Baseline. 56
3.37 Implementation of `emit_intrinsic_getByIdOffset` by walking the function call AST node. 58
3.38 Intrinsic function call to bytecode. 58
3.39 Extract from the n-body system implementation used in benchmarks. . . . 60
3.40 Optimized JavaScript for the n-body simulation. 61

4.1 Performance comparison of `nts` and `pts` in B_{jet} across T_L, T_B and T_{LB} configurations. Lower is better. 68
4.2 Performance comparison of `nts`, `pts` and `ojs` in B_{jet} across T_L, T_B and T_{LB} configurations. Lower is better. 68
4.3 Performance comparison of `nts` and `pts` in B_{μ} across T_L, T_B and T_{LB} configurations. Lower is better. 70
4.4 Nominal type violation and missed access optimizations. 75

A.1 Definition of `PropertyAccessExpression` AST node. 97
A.2 Representation of |Symbol— in TypeScript. 98

B.1 Computing layout metadata. 99

C.1 `optimizeWithTypes` configuration in `CompilerOptions` type definition. . . 100
C.2 `optimizeWithTypes` configuration in `CommandLineOption` array. 101

D.1 Implementation of `concat` for `ArrayPrototype.js` in JSC [10]. 102

List of Tables

4.1 Benchmark programs and their descriptions. 63

4.2 Benchmark setup. 64

4.3 Microbenchmark programs and their descriptions. 65

4.4 Performance comparison of `nts` and `pts` in B_{jet} across three configurations of JavaScriptCore. Lower is better. 67

4.5 Performance comparison of `nts`, `pts` and `ojs` in B_{jet} across three configurations of JavaScriptCore. Lower is better. 67

4.6 Geometric mean of `nts` and `pts` in B_{jet}. Lower is better. 69

4.7 Speedup of `pts` over `nts` in B_{jet}. For speedup, higher is better. 69

4.8 Performance comparison of `nts` and `pts` in B_{μ} across three configurations of JavaScriptCore. Lower is better. 70

4.9 Geometric mean of `nts` and `pts` in B_{μ}. Lower is better. 70

4.10 Speedup of `pts` over `nts` in B_{μ}. For speedup, higher is better. 71

4.11 Benchmark programs and emitted intrinsics. 71

4.12 Benchmark programs and emitted intrinsics. 71

Chapter 1

Introduction

JavaScript, originally created by Netscape, stands out as a versatile and extensively employed programming language recognized for its pivotal role in constructing dynamic content for the web. It has evolved into a foundational technology for web development, characterized by its high-level, interpreted, and dynamic nature, accommodating multiple programming paradigms. Noteworthy features include client-side scripting, event-driven programming, asynchronous capabilities, dynamic typing, prototype-based object orientation, and cross-browser compatibility. JavaScript's influence extends beyond traditional web development. Its adaptability is evident in various domains, including mobile app development through frameworks like React Native [35] and server-side scripting through Node.js [33]. Furthermore, JavaScript plays a significant role in desktop application development, particularly through frameworks like Electron [22].

Although JavaScript's flexibility can often be useful in practice, a naive implementation of the language will exhibit suboptimal performance. Modern JavaScript engine implementations have evolved into intricate systems that employ cutting-edge optimization techniques, including just-in-time (JIT) compilation [46] and the implementation of sophisticated inline caching techniques [59, 67]. These optimizations have significantly contributed to the improvement of JavaScript performance over time. Additionally, a prevalent practice among JavaScript engines is the adoption of type-feedback-based JIT compilers to unlock further optimization possibilities [93]. In some engines, this concept is taken a step further by incorporating multi-tier JIT compiler pipelines. In these pipelines, each JIT tier enhances performance by focusing on specific areas, building upon the improvements made in the previous tier.

Despite the appealing features of JavaScript, certain challenges diminish its ease of use. Global scope pollution, implicit type conversion, the lack of built-in null safety features, and intricate asynchronous callback structures are notable obstacles. These issues can impede code reliability and maintainability, especially in the development of large-scale applications. Recognizing these shortcomings, TypeScript was introduced as a solution to enhance the robustness and scalability of JavaScript code in large development projects.

TypeScript incorporates a structural type system and compiles to JavaScript. The primary design objective is to ensure seamless interoperability with JavaScript, incorporating several ergonomic features, most notably static typing. TypeScript introduces enhanced tooling, IDE support, ES6 features, and compatibility with existing JavaScript code.

Despite these benefits, TypeScript intentionally avoids optimizing its JavaScript output. Indeed, one of TypeScript's explicit non-goals is the aggressive optimization of runtime performance of compiled programs [16]. In practical terms, while TypeScript identifies types during compilation, this type information is not directed towards producing JavaScript code that can be better optimized for performance by an optimizing JavaScript engine. Consequently, the engine must deduce the types used at runtime even though much of the same type information is available to the TypeScript compiler at compile-time, sacrificing potential performance optimizations. Modern JavaScript engines, such as JavaScriptCore [9] and V8 [5], utilize profiling to dynamically infer types for speculative compilation [15, 63].

1.1 Contributions

This thesis introduces a broadly applicable method for enhancing the performance of JavaScript programs through the utilization of type information derived from TypeScript. To achieve this objective, types in TypeScript are classified according to their usage into nominal (similar to classes in Java) and non-nominal (structural or arbitrary) categories. Despite the unsound nature of TypeScript's type system, a notable consistency in types is observed in the majority of use cases involving classes (nominal typing). This consistency provides a substantial basis for optimization, similar to how type information is utilized for optimizations in statically typed languages such as C and C++.

In this thesis, I made modifications to the TypeScript compiler (`mtsc`) to leverage nominal type usage during type checking, enabling the generation of optimized code guided by special heuristics. Additionally, WebKit's JavaScriptCore, the underlying JavaScript engine for the Safari browser [36] and the Bun runtime [29] are modified. Within JavaScriptCore, a new runtime intrinsic is introduced to harness type information from TypeScript, offering support at both the interpreter (LLInt, Low Level Interpreter) and Baseline JIT levels. The Bun runtime is chosen for performance evaluation because it stands out as the sole non-trivial open source JavaScript runtime compatible with Node.js [33] while utilizing WebKit [29].

The evaluation of the implementation involves a performance comparison of JavaScript programs from the JetStream 2.1 JavaScript benchmark suite [18, 20]. The same programs are then ported to TypeScript and compiled with `mtsc` in two modes: with optimizations and without optimizations. In particular, when TypeScript programs are written in a nominal style, the resulting JavaScript, generated by compiling TypeScript with optimizations, outperforms the JavaScript derived from compiling TypeScript without optimizations, by

up to 12%. Furthermore, when non-nominal typing styles are employed, the performance of JavaScript with optimizations is comparable to the original JavaScript from JetStream 2.1 without significant degradation.

In summary, this thesis makes the following key contributions:

1. The development of an enhanced TypeScript compiler (`mtsc`) tailored for type-guided performance optimization.
2. The introduction of new runtime intrinsics in JavaScriptCore designed to collaborate seamlessly with `mtsc`.
3. The comprehensive evaluation of the performance outcomes produced by the configured setup.

1.2 Overview

The main aim of this thesis is to leverage TypeScript's type information to optimize JavaScript's runtime performance. In the industry, TypeScript is steadily gaining traction in the development of web applications due to its numerous ergonomic improvements and advantages over JavaScript [70, 25]. This trend is evident as several popular web frameworks, such as Angular [28] and NestJS [32], prioritize a TypeScript-first development experience. Additionally, TypeScript has been adopted for large-scale projects by companies like Microsoft, Slack, Airbnb, and Asana [70]. Furthermore, it is increasingly common for new web application projects to adopt TypeScript instead of JavaScript. Given these factors, ensuring performant JavaScript is essential when writing TypeScript code. To achieve this objective, a fundamental understanding of concepts such as JavaScript engines, the JavaScript and TypeScript languages, challenges in JavaScript optimization, and runtime optimization intrinsics is essential (Chapter 2).

The central idea for optimization revolves around utilizing type information in TypeScript to optimize property access operations (Chapter 3). These optimizations involve eliminating object shape checks and full property lookup operations by leveraging nominal type information from TypeScript. These optimizations are implemented at two levels: the TypeScript compiler and the JavaScript engine. In the TypeScript compiler, layout instructions are generated (when necessary) for objects adhering to nominal type usage, facilitating the utilization of runtime intrinsics for optimizing property-access operations on these objects. Additionally, at the JavaScript engine level, a new runtime intrinsic is developed to support the access optimizations applied by the TypeScript compiler.

The changes made to both the TypeScript compiler and the JavaScript engine are evaluated using a set of benchmarks (Chapter 4). The first benchmark is a standard JavaScript benchmark commonly used by popular browser engines [18], while the second is a microbenchmark comprising a series of programs specifically designed to isolate the

performance enhancements resulting from the implemented optimizations. The results from both benchmarks show a significant improvement in performance across multiple programs as a result of the optimizations that were developed.

Chapter 2

Background

This chapter presents fundamental concepts and background information essential for a clear understanding of the thesis. It encompasses discussions on JavaScript and JavaScript engines, with a focus on JavaScriptCore, TypeScript, the Bun runtime environment, and the rationale behind optimizing with TypeScript. Given the complexity of these technologies, the chapter aims to offer a succinct understanding without delving into detailed tutorials or formal specifications.

2.1 JavaScript

JavaScript is a widely used programming language renowned for its pivotal role in web development. Initially conceived for web browsers, it has since transcended its origins and found widespread adoption across multiple platforms. Developed in 1995, JavaScript has become fundamental to web development, offering high-level, interpreted, and dynamic capabilities that support procedural and object-oriented programming paradigms. The language's dynamic nature, coupled with its adaptability, makes JavaScript a powerful tool, albeit one that developers navigate carefully in the pursuit of robust and efficient solutions, as it is very trivial to write code with unintended behaviour.

2.1.1 JavaScript Syntax and Semantics

Although JavaScript supports a number of features, the following discussion emphasizes objects, functions, and classes without delving into in-depth details.

Functions

Functions in JavaScript work similarly to those in most imperative C-family programming languages. Functions are reusable blocks of code that can accept arguments and return

values. The program in Figure 2.1 illustrates how functions can be defined and used.

```
function add(x, y) {
   return x + y;
}

add(1, 2); // returns 3
```

Figure 2.1: A simple JavaScript function.

Unlike many imperative languages, JavaScript functions are first-class values that can be used for object-oriented programming, although the introduction of classes somewhat diminishes this use. Statements in JavaScript are usually terminated with a semicolon, similar to C. However, unlike C, semicolons can be omitted due to automatic semicolon insertion (ASI) [21], although this is usually not recommended.

Objects

JavaScript objects are fundamental data structures that allow developers to represent and organize data in a structured manner. Objects in JavaScript are collections of key-value pairs, where each key serves as an identifier for the associated value. This key-value pairing facilitates the creation of complex and hierarchical data structures. Objects can encapsulate various data types, including primitive values, other objects, and functions, providing a versatile means of modeling real-world entities. In JavaScript, objects can be defined using literal notation or by instantiating a *constructor function*. The program in Figure 2.2 shows how objects may be defined and used.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  greet: function() {
    console.log(`Hello, ${this.firstName} ${this.lastName}!`);
  }
};

// prints "John"
console.log(person.firstName);
// calls the greet function in the object person. Prints "Hello, John Doe!"
person.greet();
```

Figure 2.2: Objects in JavaScript.

In JavaScript, prototypes are the mechanism by which objects inherit features from one another [26]. Every object has a built-in prototype, which is itself an object. This prototype object, in turn, has a prototype of its own, and so on, until an object is reached with `null` as its prototype [24]. `null` has no prototype and acts as the final link in this prototype chain.

Constructors

Functions in JavaScript may also be defined as object constructors. The prototype-based nature of JavaScript allows implementing "methods" on these constructors. The example in Figure 2.3 defines a `Point` constructor with an `add` method invokable on the constructed object.

```
function Point(x, y) { // constructor
   this.x = x;
   this.y = y;
}

Point.prototype.add = function(inc) {
  this.x += inc;
  this.y += inc;
}

const p1 = new Point(1, 2);
p1.add(5);
```

Figure 2.3: Creating objects using constructors in JavaScript.

Classes

Classes were introduced in JavaScript to provide a cleaner and clearer way for supporting object-oriented patterns. Classes provide a way to encapsulate data and behaviour and to create reusable components. The program in Figure 2.3 can be written using classes, as seen in Figure 2.4.

JavaScript is a prototype-based language [44] and does not support true classes, as found in languages like Python [34] and Ruby [65]. Instead, it utilizes prototypes to inherit properties from one object to another. Despite their apparent simplicity, prototypes in JavaScript serve as the fundamental building blocks for many of its language constructs. For example, data structures such as `Array`, `JSON` and `Date` are fundamentally objects in JavaScript.

```
class Point {
   constructor(x, y) {
      this.x = x;
      this.y = y;
   }

  add(inc) {
    this.x += inc;
    this.y += inc;
  }
}

const p1 = new Point(1, 2);
p1.add(5);
```

Figure 2.4: Classes in JavaScript.

Due to JavaScript's prototype-based nature, the programs presented in Figures 2.3 and 2.4 are identical and semantically equivalent. The object created by invoking the `Point` function constructor in Figure 2.3 is identical to the one created using the `Point` class in Figure 2.3. Therefore, when both programs are executed, they exhibit the same behaviour.

2.1.2 Dynamic Typing

The program in Figure 2.5 demonstrates dynamic typing in JavaScript. The error in line 17 is statically detectable, but only occurs at runtime.

```
1  let x = "foobar";
2  let y = [1, 2, 3];
3
4  // x is assigned an array with no errors
5  x = y;
6
7  // x is reassigned an object with no errors
8  x = { name: "John", age: 25 };
9
10 // this errors at runtime, even though this error is statically detectable
11 // -> Uncaught TypeError: x is not a function
12 x(1, false);
13
14 const p = 12;
15 // this errors at runtime since no type errors are statically detectable in JS
16 // -> Uncaught TypeError: Assignment to constant variable.
17 p = 10;
```

Figure 2.5: Dynamic typing in JavaScript.

2.2 JavaScript Engines

In its simplest form, a JavaScript engine functions as an interpreter that executes JavaScript code. However, modern JavaScript engines have evolved into complex pipelines comprising multiple components.

Typically, the front end comprises lexical and semantic analysis. The backend often includes a code generator for a custom instruction set (bytecode) and a (stack- or register-based) virtual machine that executes the instruction set. The code generator traverses the AST to emit bytecode, which is then executed by the virtual machine. The engine also automatically manages memory used at runtime, typically implemented using well-defined garbage collection algorithms. However, engines with a JIT compilation system may introduce additional complexities.

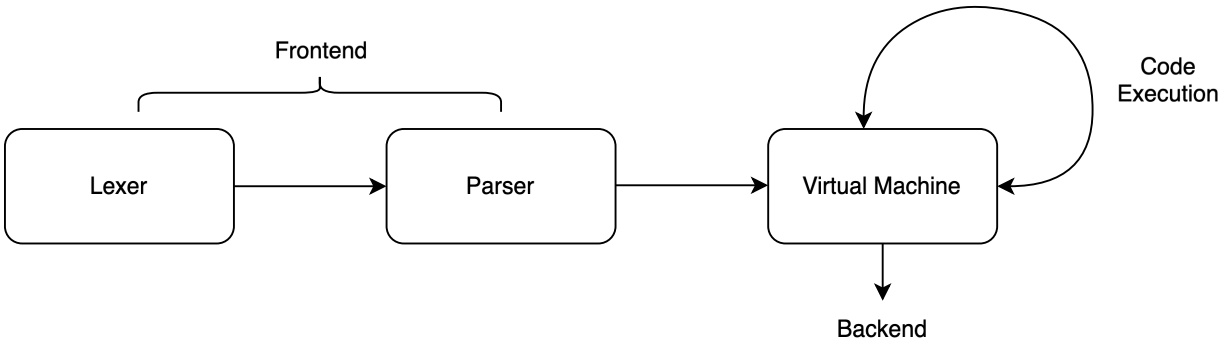

Figure 2.6: A simple JavaScript engine architecture.

A naive implementation of a JavaScript engine may rely entirely on interpretation, as illustrated in Figure 2.6. However, the semantics of JavaScript can make such an approach suboptimal due to its heavy reliance on prototypal objects. Modern JavaScript engines often employ a combination of interpretation and compilation techniques to achieve optimal performance. These engines utilize JIT compilers to gather runtime information, which is then used to optimize the performance of a running program. During program execution, a JIT compiler identifies opportunities to optimize certain code paths for improved performance. It compiles these "hot" code paths (based on heuristics such as frequency of execution) into machine code, which is then executed for optimal performance. JIT compilers may employ traditional compiler optimization techniques (e.g., common subexpression elimination, range analysis, dead code elimination, alias analysis, loop-invariant code motion, etc.) to eliminate or elide runtime checks.

However, because JavaScript is dynamically typed, assumptions made during optimization may no longer hold true during subsequent execution (e.g., due to changes in program state or input of an unexpected type). As a result, the optimized code may become invalid. In such cases, the compiled code may be discarded or reverted to a less optimized form, and execution reverts to interpretation or the less optimized code. This process is known as deoptimization [68].

Chrome's JavaScript engine, V8 [5], initially utilized a non-optimizing baseline compiler to compile JavaScript code just-in-time, before execution. However, this method led to significant memory consumption during execution. To address this issue, V8 introduced the Ignition interpreter [12] to alleviate memory consumption [11].

JavaScriptCore interprets JavaScript bytecode on a register-based virtual machine. However, it employs JIT compilation to convert the bytecode into machine code for native execution, provided that sufficient profiling information has been gathered and certain JIT heuristics are satisfied. Firefox's JavaScript engine, SpiderMonkey [2], follows a similar approach to JavaScriptCore [1].

2.2.1 Optimization Techniques and Challenges

Although modern JavaScript engines utilize JIT compilation for performant JavaScript execution, this is not without its challenges.

```
1  function add(a, b) {
2      return a + b;
3  }
4
5  add("a", 1);
6  add(1, "b");
7
8  for (let i = 0; i < 100000; i++) {
9     add(1, 2);
10 }
11
12 add("a", "b");
```

Figure 2.7: A JIT triggering example [48].

Consider the program in Figure 2.7. Within the `add()` function (line 1), the expression `a + b` exhibits varied semantics based on the operand's type: numerical addition for numbers, concatenation for strings, and so forth. Consequently, the JavaScript engine acquires runtime type information, which is pivotal for optimizing the function concerning frequently encountered types. In this instance, the variable `a` is consistently observed to store values of type `number` across numerous function calls within the `for` loop at line 8. As a result, during the JIT compilation phase, the compiler speculates that these types will persist unchanged and proceeds to emit native code to swiftly execute numerical addition [48]. Such speculative optimizations foster the creation of efficient code tailored to each data type.

Nevertheless, the dynamic typing inherent in JavaScript implies that assumptions regarding observed types may become invalidated over time. In Figure 2.7, this is demonstrated by the function call to `add()` with a string parameter at line 12. Consequently, JIT compilers are compelled to emit supplementary type checks to ascertain the validity of their optimization assumptions at runtime. Failed validations prompt a bailout or deoptimization [68], where the observed type information is revised and factored into potential recompilations [48].

Moreover, runtime validations not leading to deoptimization consume valuable CPU cycles without advancing a program's execution. Consequently, JavaScript engines endeavor to eliminate superfluous checks, such as those grounded on assumptions proven to hold universally. For instance, considering that numerous built-in `Math` function calls return values of type `number`, optimization passes within the code might emit native arithmetic instructions eliding runtime type checks [48]. Challenges like these complicate the task of optimizing JavaScript correctly and contribute to the sophistication required in optimizing JIT compilers.

Type Inference

Modern JavaScript engines utilize runtime type inference in their optimizing JIT compilers to enhance optimizations and opportunistically eliminate runtime checks. JavaScriptCore employs speculation (discussed in Section 2.3.9) to infer types in the DFG and FTL JITs, leveraging value profiling data from the LLInt and Baseline tiers. SpiderMonkey [2] employs an optimistic whole-program, hybrid (static and dynamic) analysis approach to infer types for stack slots, arguments, and local variables [63]. V8 uses dynamic type inference in its optimizing JIT tier, known as Crankshaft, to eliminate redundant IR instructions and runtime checks [50].

2.3 JavaScriptCore

At the core of the WebKit project [3] lies JavaScriptCore, the native JavaScript engine for WebKit [9], implementing the ECMA-262 specification [62]. JavaScriptCore consists of a low-level interpreter called LLInt, written in a portable assembly dialect, and multiple JIT compilers. The Baseline compiler, functioning as a template JIT compiler, occupies the foundational tier among these compilers. It shares a significant amount of code with LLInt and integrates fundamental optimizations, making it highly efficient for native code generation. Succeeding the Baseline JIT is the Data Flow Graph compiler, also known as DFG JIT, which produces more optimal code compared to the Baseline JIT. The Faster Than Light JIT compiler, or FTL JIT, succeeds the DFG JIT. The FTL JIT is a fully optimizing compiler that employs various classical compiler optimizations, resulting in more efficient code compared to the DFG JIT. Figures 2.8 and 2.9 illustrate the tiers of JavaScriptCore.

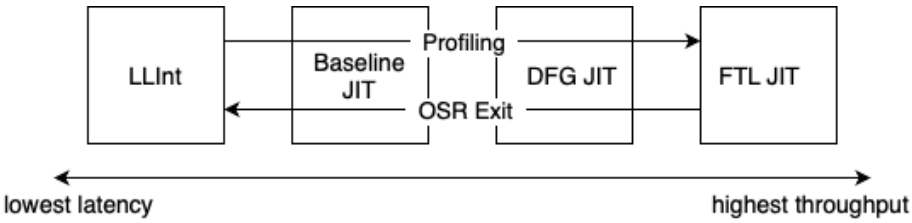

Figure 2.8: The tiers of JavaScriptCore [15].

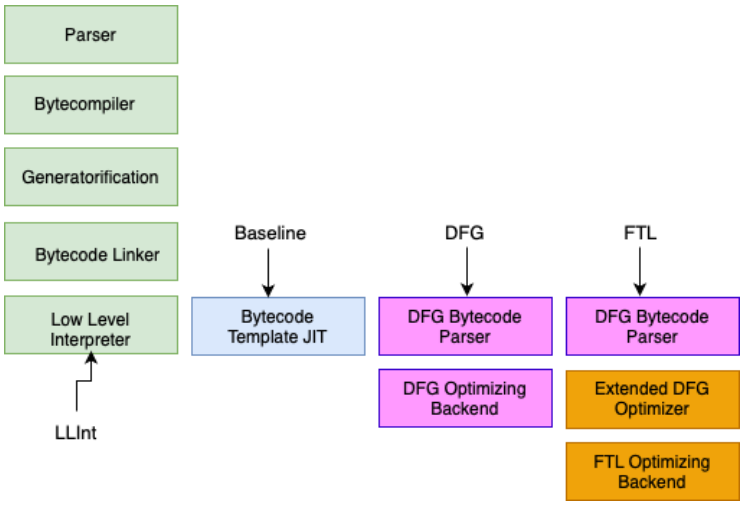

Figure 2.9: JavaScriptCore architecture [15].

The tiered architecture of the engine facilitates speculation [15], enabling the optimizing tiers (DFG and FTL) to produce efficient code at runtime. Speculation, discussed in Section 2.3.9, enables the application of traditional compiler optimization techniques to dynamically typed languages. Additionally, it provides a secondary benefit by allowing a nuanced adjustment of the trade-off between throughput and latency on a per-function basis. Some functions, especially those that execute once, might incur higher costs if subjected to compilation compared to interpretation. Conversely, functions with longer execution times might exceed the overall execution time required for efficient compilation by an aggressive optimizing compiler. However, a nuanced scenario emerges, encompassing functions that fall between these extremes. These functions operate for durations insufficient to warrant an aggressive compiler's intervention but are long enough to benefit from intermediate compiler designs, providing notable speed improvements.

JavaScriptCore employs various optimization strategies to determine when to transition execution between LLInt and the JIT tiers, a process known as *tiering-up*. Section 2.3.8 discusses some of the heuristics guiding this process. The LLInt and JIT tiers form intricate systems designed to collaboratively deliver performance for a dynamically versatile language like JavaScript.

2.3.1 Execution Pipeline Architecture

In JavaScriptCore, bytecode is the source of truth across all tiers, persisting in memory throughout the entire program execution [13]. LLInt initiates execution by interpreting the generated bytecode and collecting profiling data. After gathering sufficient profiling information for frequently executed code, say, a function, LLInt tiers-up to the Baseline compiler for that function.

The Baseline compiler is launched in a separate thread within the same process. It begins compiling the function, leveraging the available profiling information, while execution continues in LLInt. Once the Baseline JIT completes compilation to native code, the function's current execution (and future calls) transitions from LLInt to the Baseline-generated native code. Upon reaching a certain threshold during execution of the function, the Baseline tiers-up to the DFG. Similarly, the DFG tiers-up to the FTL JIT. However, the FTL does not tier-up, which is the highest available tier.

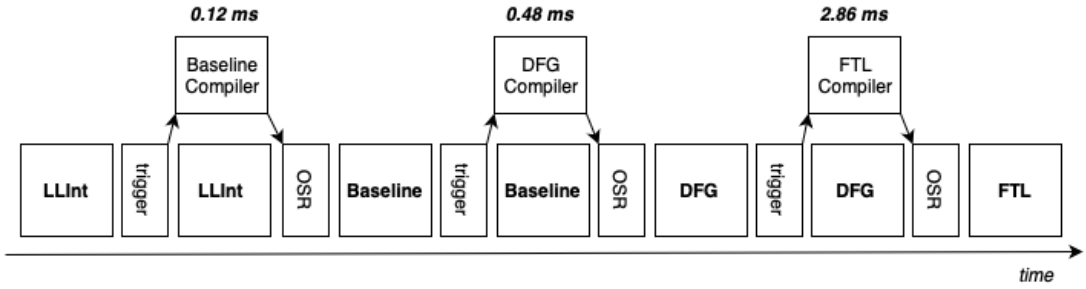

Figure 2.10: Example timeline of a simple long loop executing in JavaScriptCore [15].

Empirically, the performance hierarchy among JavaScriptCore's tiers reveals that the Baseline JIT executes code approximately 2 times faster than LLInt, DFG executes code about 5 times faster than Baseline, and the FTL JIT executes code about 1.5 times faster than DFG [15]. These findings, demonstrated through the execution of a long-running loop [15], are depicted in Figure 2.10. JavaScriptCore also employs a garbage collection strategy that simplifies the implementation of the virtual machine and supports speculation. Some key features of the garbage collector include conservative stack scanning for pointers, non-movement of objects, and fixpoint completion. These features streamline the compiler and make it easier for speculation and other optimizations to be implemented and invoked.

2.3.2 Bytecode

JavaScriptCore's bytecode instructions consist of an opcode along with one to three operands and associated metadata information. They follow a three-address code format and operate over virtual registers [13]. Although only a small number of virtual registers are typically used, the instructions are allowed to assume the presence of an infinite number of registers.

The bytecode closely mirrors the high-level nature of JavaScript, incorporating transformations only when performance remains uncompromised. It is directly interpretable, functioning seamlessly as a unified stream for interpretation, caching, and compilation. Moreover, it is untyped, as virtual registers and the majority of opcodes lack static types. This design choice ensures consistency across tiers and facilitates profiling before type inference.

```
function operate(a, b, c) {
    a.acc *= b;
    a.acc /= c;
    return a.acc;
}

// Compiled bytecode snippet:

Compiled #AsSAYq into bytecode 45 instructions in 0.373708 ms.
operate#AsSAYq:[0x1120dc340->0x11209d700, NoneFunctionCall, 45]: 9 instructions
(0 16-bit instructions, 0 32-bit instructions, 5 instructions with metadata);
141 bytes (96 metadata bytes); 4 parameter(s); 6 callee register(s);
5 variable(s); scope at loc4

bb#1
Predecessors: [ ]
[   0] enter
[   1] get_by_id          dst:loc5, base:arg1, property:0, valueProfile:1
[   7] mul                dst:loc5, lhs:loc5, rhs:arg2, profileIndex:0,
                            operandTypes:OperandTypes(126, 126)
[  13] put_by_id          base:arg1, property:0, value:loc5, flags:
[  19] get_by_id          dst:loc5, base:arg1, property:0, valueProfile:2
[  25] div                dst:loc5, lhs:loc5, rhs:arg3, profileIndex:1,
                            operandTypes:OperandTypes(126, 126)
[  31] put_by_id          base:arg1, property:0, value:loc5, flags:
[  37] get_by_id          dst:loc5, base:arg1, property:0, valueProfile:3
[  43] ret                value:loc5
Successors: [ ]


Identifiers:
  id0 = acc
```

Figure 2.11: A simple JavaScript program compiled to bytecode.

JavaScriptCore compiles programs into a bytecode graph of basic blocks. Figure 2.11 shows a simple function and the bytecode snippet associated with its basic block. In the bytecode snippet, `bb#` identifies the basic block in the graph. `Predecessors` is a list containing the predecessors of the current basic block (`bb#1`) in the graph. The left column

represents the offset of the instruction in the instruction stream. The middle column lists the actual opcodes corresponding to each bytecode instruction, and the right column the operands associated with each opcode. Consider the `mul` instruction that implements a multiplication operation:

```
mul dst:loc5, lhs:loc5, rhs:arg2, profileIndex:0, operandTypes:OperandTypes(126, 126)
```

`dst:loc5` represents the destination register, `lhs:loc5`, and `rhs:arg2` are register operands (and, in fact, registers themselves) corresponding respectively to the left-hand side and right-hand side of the operation. The remaining operands include profiling and type metadata associated with the instruction. At the end of the basic block is `Successors`, a list of reachable blocks from the current basic block.

2.3.3 Fast Path / Slow Path

Bytecode instructions can be implemented prioritizing runtime type checks for what the implementer deems as the most probable scenario.

```
1 function op_add(a, b) {
2     if (isInt32(a) && isInt32(b))
3         return a + b;
4     return slowAdd(a, b);
5 }
```

Figure 2.12: Hypothetical fast path / slow path implementation of an add instruction.

When an opcode is designed to prioritize inline operations for a specific type over other types, it is said to have a fast path for that type and a slow path for others. Figure 2.12 illustrates a hypothetical implementation of an `add` instruction with a fast path for 32-bit integers and an out-of-line slow path for additions involving other value types. This implementation pattern is pervasive in JavaScriptCore, where fast paths are typically implemented inline, while slow paths involve out-of-line calls.

2.3.4 OSR

On stack replacement (OSR) is the process by which execution is transferred from one tier to another. In OSR, values are transferred from the current running tier to the location they should be in the next tier, allowing for continued execution. When transitioning from LLInt to the Baseline, the values are already in the correct locations as specified by the compiled bytecode, requiring no further action. However, in optimizing tiers, the process of transitioning values becomes non-trivial and must be performed.

2.3.5 LLInt

LLInt is an interpreter responsible for executing bytecode produced by the parser. It follows the JIT application binary interface (ABI) of JavaScriptCore and adheres to the calling, stack, and register conventions used by the JIT compilers [9]. This alignment facilitates cost-effective calls between LLInt and JITed functions, as well as OSR between LLInt and the JIT tiers [15]. LLInt also incorporates inline caching optimizations to ensure efficient property access.

LLInt is written in *Offlineasm*, a specialized assembly dialect designed specifically for LLInt. Offlineasm introduces its own mnemonics and register names, aligning with the portable assembly used in the JIT tiers. Some high-level mnemonics require lowering, and Offlineasm designates specific scratch registers (e.g., `t0`, `t1`, `t2`, `t3`, `t4`) for this purpose. Offlineasm features a functional *macro* language that allows the passing of macro closures, offering better abstractions compared to native assembly [15]. In Offlineasm, a macro is a lambda expression, which can be either anonymous or named, capable of taking zero or more arguments.

The Offlineasm compiler, written in Ruby [65], compiles to multiple CPU architectures (x86, ARM, RISC-V) and C++. The decision to implement LLInt in Offlineasm, rather than in C++ as in other tiers, makes it easier for LLInt to utilize the same stack used by the code compiled with the JIT tiers when operating in the interpreter. Consequently, it eliminates the need to manage multiple stacks, such as a C++ and a JavaScript stack, simplifying OSR from LLInt to the Baseline compiler and vice versa.

```
1 LOOP:
2   Inst = InstructionStream[PC++]
3   Decode(Inst):
4     Case Add -> DstReg = doAdd()
5     Case Sub -> DstReg = doSub()
6     // other cases..
```

Figure 2.13: A high-level view of LLInt.

LLInt also follows a conventional virtual machine instruction cycle [66], iterating over bytecode and executing each instruction based on its intended purpose [14]. This process is illustrated in Figure 2.13. Additionally, LLInt gathers profiling information during bytecode execution and maintains counters that measure code execution frequency [14, 15]. These parameters play a vital role in optimizing code and enabling JIT tiering through OSR.

Furthermore, LLInt enables JavaScriptCore to execute code in no-JIT mode (commonly referred to as “mini mode”), offering advantages such as increased security and reduced

memory usage [15]. This mode proves particularly useful on CPUs without JIT support, where LLInt excels.

```
macro llintOpWithReturn(opcodeName, opcodeStruct, fn)
    llintOp(opcodeName, opcodeStruct, macro(size, get, dispatch)
        makeReturn(get, dispatch, macro (return)
            fn(size, get, dispatch, return)
        end)
    end)
end

# mov
llintOpWithReturn(op_mov, OpMov, macro (size, get, dispatch, return)
    get(m_src, t1)
    loadConstantOrVariable(size, t1, t2)
    return(t2)
end)


# to_string
llintOpWithReturn(op_to_string, OpToString, macro (size, get, dispatch, return)
    get(m_operand, t1)
    loadConstantOrVariable(size, t1, t0)
    btqnz t0, notCellMask, .opToStringSlow
    bbneq JSCell::m_type[t0], StringType, .opToStringSlow
.opToStringIsString:
    return(t0)

.opToStringSlow:
    callSlowPath(_slow_path_to_string)
    dispatch()
end)
```

Figure 2.14: `mov` and `to_string` implementation in LLInt, defined in LowLevelInterpreter64.asm [10]

Figure 2.14 presents implementation details for `mov` and `to_string` opcodes. The `op_to_string` opcode has a fast path for values that are already strings and a slow path for other value types. Specifically, `mov` is responsible for assignment storage, while `to_string` handles string conversion in JavaScript. `mov` is implemented by loading a value from a source register to a destination register using temporary registers. The `to_string` operation is implemented using a fast path/slow path pattern. The fast path checks whether the value to be converted into a string is already a `string`; if so, the value is returned. Otherwise, the slow path is taken to convert the value into a string. The following sections discuss the JIT compilers available in JavaScriptCore.

2.3.6 Baseline JIT

The Baseline compiler operates as a method JIT, compiling entire functions and generating a machine code template for each bytecode instruction without considering relationships between multiple instructions in the function [15]. Baseline achieves performance speedups over LLInt by fundamentally eliminating interpreter dispatch for bytecode execution. Interpreter dispatch is a costly aspect of bytecode execution, as the indirect branches used for selecting the implementation of an opcode are challenging for the CPU to predict [86].

The Baseline JIT becomes active for functions that are invoked at least N times or execute a loop body at least L times. Usually, N is 6 and L is 100, but these values may vary as the actual heuristics depend on factors such as function size and current memory pressure [9].

Both LLInt and Baseline collect lightweight profiling information to enable speculative execution in the DFG tier. This information includes recent values loaded into arguments, retrieved from the heap, or obtained from a call return. Moreover, the design of all inline caching mechanisms in both LLInt and Baseline is tailored to facilitate the easy retrieval of type information by the DFG. For instance, the DFG can determine whether a heap access sometimes, often, or always encounters a particular type by examining the current state of an inline cache [9]. This capability proves crucial in determining the most advantageous level of speculation.

Baseline performs minimal optimizations beyond the generation of code templates and avoids register allocation between instruction boundaries [15]. However, it provides support for polymorphic inline caching [67] for heap accesses [9]. Some localized optimizations are implemented by the designers, such as recognizing when an operand for a mathematical operation is a constant or leveraging profiling information gathered by LLInt [15].

2.3.7 DFG and FTL JIT

The Data Flow Graph (DFG) JIT compiler transforms bytecode into the specialized compiler intermediate representation DFG IR, enabling sophisticated reasoning about speculation while prioritizing efficient code generation. The DFG JIT becomes active for functions invoked at least N times or undergoing a loop at least L times. Usually N is 60 and L is 1000; however, the precise values are subject to additional heuristics [9].

The Faster Than Light (FTL) JIT compiler specializes in extensive compiler optimizations, emphasizing peak throughput while purportedly maintaining fast compilation speeds. It builds upon and extends optimizations from the DFG JIT, working with multiple intermediate representations such as DFG IR, DFG SSA IR, Bare Bones Backend (B3) IR, and Assembly IR to enhance overall performance. Activation of the FTL JIT occurs for functions that are invoked thousands of times or loop tens of thousands of times [9].

2.3.8 Optimization Techniques

JavaScriptCore employs various optimization techniques to deliver performant JavaScript execution. It relies on numerous heuristics to identify when specific optimizations are applicable and to determine the most suitable optimizations for particular scenarios. This section discusses one of these strategies.

Inline Caching

Property accesses and function calls pose significant optimization challenges in JavaScript due to the dynamic nature of objects and polymorphic nature of function calls (see Figure 2.7). JavaScriptCore addresses these challenges using inline caches (ICs) [59].

Inline caches in JavaScriptCore precisely record the shape of an object, along with any known properties being accessed on the object. An object's shape refers to the fields of the object. In JavaScript engines such as V8 and JavaScriptCore, an object's shape encompasses its property names along with the corresponding storage location offsets for the property values [51, 15]. In JavaScriptCore, object shapes are known as *structures*.

Inline caches applies not only to monomorphic access, that is, property access involving a single object shape, but also to polymorphic access, covering up to 8 different shapes and can be done inline. Both LLInt and Baseline collect profiling information used to implement ICs, and they also directly implement ICs, contributing to faster execution in these tiers. This dual role makes the cost of collecting profiling information worthwhile, as the gathered data can be immediately utilized while still in the lower-level tiers.

JavaScriptCore utilizes polymorphic inline caches which extend inline caches to have more than one shape [67]. Dynamic property access is implemented by integrating inline caches with structures, inspired by maps in Self [55]. The inclusion of structures is fundamental to optimizing object properties in JavaScript, as objects often serve as ordered mappings from strings to JavaScript values, undergoing operations such as lookup, insertion, deletion, replacement, and iteration.

Structures Structures in JavaScriptCore are hash consed, which implies that objects with the same properties in the same order are likely to share the same structure. Object representation is divided into the object itself, containing property values and a structure pointer, and the structure, which acts as a hash table mapping (string) property names to indices in objects with that structure. This design allows for O(1) checks for object structures and establishes a mechanism for efficient property lookup. For example, checking if an object has a particular structure is achieved by simply loading the structure pointer from the object header and comparing the pointer to a known value.

The key insight from Self [55] is that property access sites in a program often involve objects that share the same structure. Figure 2.15 illustrates objects represented using structures. A structure specifies the property names of an object and their order.

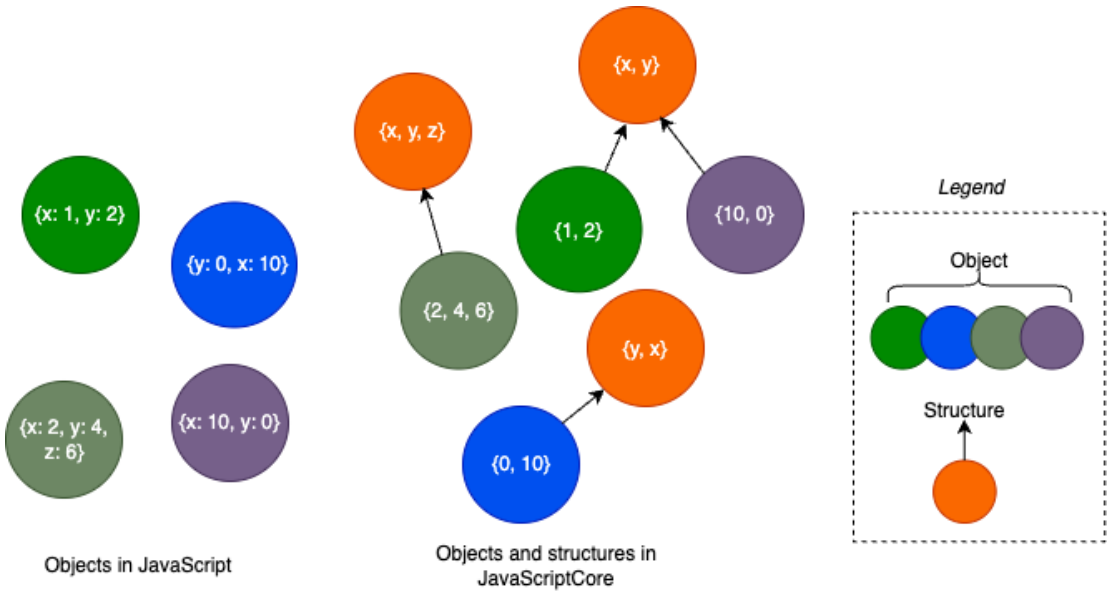

Figure 2.15: Objects and structures in JavaScriptCore [15].

Structures can also indicate whether objects are in *dictionary* or *uncacheable dictionary* mode, which represent two levels of hash table complexities resulting from the dynamic attachment of properties to objects. In both scenarios, the structure ceases to be hash consed and is directly associated on a one-to-one basis with its object. For dictionary objects, new properties can be added in-place without altering the structure. Similarly, uncacheable dictionary objects allow properties to be deleted without affecting the structure.

Objects JavaScriptCore models objects with a 64-bit header, consisting of a 32-bit structure ID and 32 bits for additional object state information, such as type information and array metadata. Each object can have one or two pieces of memory. The first piece of memory contains the structure ID of the object, state information and inline storage slots, while the second piece of memory corresponds to out-of-line storage, accommodating properties that do not fit in the inline slots. Both pieces of memory are allocated in 64-bit chunks, simplifying the memory allocator and garbage collector. Each piece of memory is indexable using 64-bit accesses. An object's structure ID is always accessible from the object pointer (referring to the first piece of memory) by reading the first 32-bit value.

JavaScriptCore employs two tiers of storage for objects, determined by the nature of their properties: inline slots and out-of-line slots. Statically configurable, inline slots act as the primary storage for object properties. The allocation of inline slots is determined by a straightforward static analysis around the object's allocation site. Named object properties may exist in inline slots, out-of-line slots, or both.

When a property exceeds the capacity of inline slots, an additional storage is allocated

to store extra properties out-of-line. This out-of-line storage is also allocated for properties that are not statically detectable or are assigned far from an object's allocation. Access to this storage is available from an object through the out-of-line pointer.

Property Access

Consider the code snippet in Figure 2.16:

```
1    const obj = {x: 2, y: 3};
2    const res = obj.y;
```

Figure 2.16: Property access in JavaScript.

Figure 2.17 depicts the memory layout of the object `obj`. JavaScriptCore stores properties `x` and `y` of `obj` in the object's inline slot. As `obj` has only two statically known properties, JavaScriptCore does not allocate any out-of-line storage; therefore, the out-of-line pointer in this case is `null`.

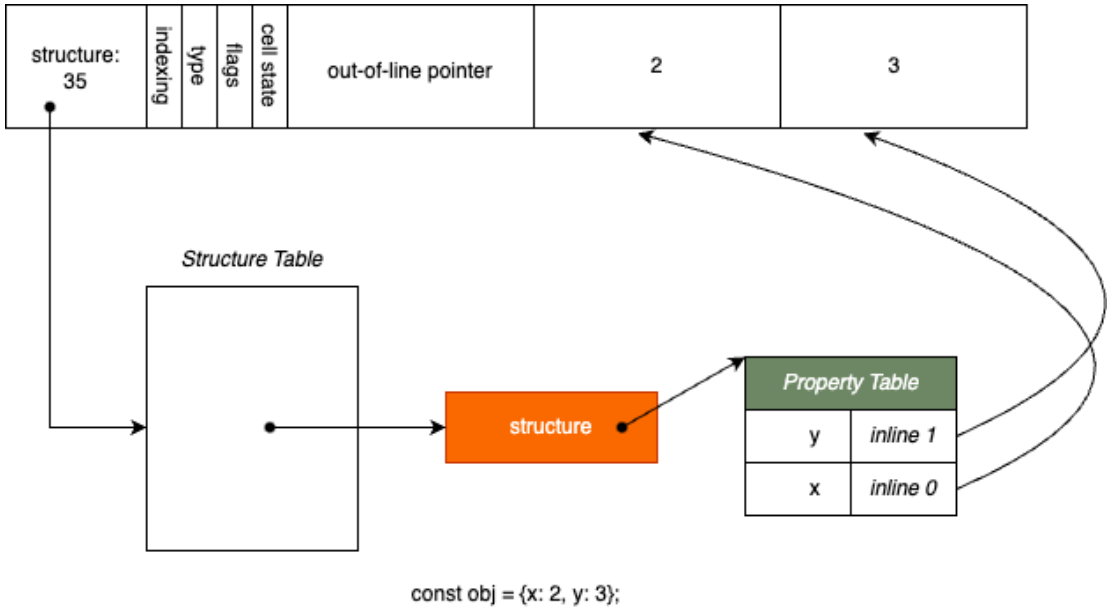

Figure 2.17: Memory layout of an object in JavaScriptCore [15].

When accessing the value of the property `y` in the object `obj`, the structure ID of the object is utilized to look up the object's structure in a structure table maintained by JavaScriptCore for each VM thread state. Fetching the object's structure from this table roughly translates to an index access. Once the specific structure is identified, the property

table associated with that structure is accessed. This property table, implemented as a hash table, contains pointers from property names to indices in the object's inline slots or in its out-of-line storage slots. Since the object `obj` has properties that are statically detectable at compile time, the property names stored in the structure's property table directly correspond to the indices in the object's inline storage. In Figure 2.17, property `y` maps to an inline slot index of 1 in the structure's property table, which corresponds to the value 3 in the object's inline slots.

If the name of a property in an object property access is known, along with the unique structure ID of the object on which the property access is being performed, the property value can be efficiently retrieved. The JavaScriptCore object model is designed to make fast property access possible by capitalizing on scenarios like the one mentioned, using inline caching. Inline caching is implemented using the fast path / slow path pattern.

Inline Caching in LLInt

Consider the JavaScript code in Figure 2.16, assuming all objects flowing into the code have structure ID 35, similar to the object in Figure 2.17, inline caching this property access involves emitting code like the following:

```
if (obj->structureID == 35)
    res = obj->inlineStorage[0]
else
    res = slowGet(o, "y")
```

Determining that object `obj` has structure 35 is not statically available in JavaScript. Inline caches obtain this information during runtime using self-modifying code. LLInt implements property access using the bytecode instruction `get_by_id`. The format of this instruction is defined as follows:

```
get_by_id <dst> <base> <property> <cachedStructureID> <cachedOffset>
```

`dst` represents the destination register for the property access value, `property` is the property name, `cachedStructureID` indicates the cached structure ID, and `cachedOffset` specifies the property offset in the structure's property table. Before execution, the cached structure ID is initialized with a unique value that is unusable by any structure, signifying the absence of a cached value. An example property access bytecode instruction could be:

```
get_by_id  dst:loc5, base:loc6, property:0, cachedStructureID:-1, cachedOffset:-1
```

During the execution of the `get_by_id` instruction, LLInt attempts to follow the fast path. This path involves loading the property at the cached offset if the object's structure

ID matches the cached structure ID. However, since the cached structure ID is initially set to an invalid value, LLInt takes the slow path. This slow path performs a complete property lookup, updating the metadata of the `get_by_id` bytecode instruction by storing the structure ID and property offset obtained during the full lookup. Upon subsequent property access, LLInt takes the fast path of `get_by_id` if the object has the same structure ID as that found in the cached bytecode metadata. In this case, the instruction simply loads the property at the cached offset, optimizing the process.

Inline caches not only enhance performance but also serve as an excellent source of profiling data due to the highly amenable nature of the metadata used for caching. This strategy is pervasive across all execution tiers, serving as a comprehensive optimization technique. Beyond its primary role in accelerating code execution, inline caching acts as a precise profiling source, providing insights into the type cases observed during runtime operations. When coupled with structures, inline caches facilitate the transformation of dynamic property accesses into instructions that are inherently straightforward to optimize.

2.3.9 Speculation

Speculation in language optimization aims to apply traditional compiler techniques to enhance the performance of dynamic languages such as JavaScript. Traditional compiler optimizations are challenging due to the absence of type information in dynamic languages, preventing meaningful optimizations for fundamental operations. Speculative compilers employ profiling to dynamically infer types and generate code with dynamic type checks. If the program diverges from the profiled types during execution, the optimized code is discarded and the process is repeated. This approach allows optimizing compilers to work with a statically typed representation of a dynamically typed program. JavaScriptCore implements *diamond* and OSR speculation [15] in its optimizing tiers.

2.4 Bun

Bun is a JavaScript runtime designed for the modern JavaScript ecosystem [29]. It is built around JavaScriptCore, purportedly offering rapid startup times and efficient code execution. With a streamlined set of APIs, Bun simplifies common tasks such as initializing an HTTP server and handling file operations. Additionally, Bun enhances the JavaScript development experience by adopting a batteries-included approach, providing a comprehensive toolkit for building JavaScript applications, which includes a package manager, test runner, and bundler. Furthermore, Bun serves as a drop-in replacement for Node.js [33], implementing hundreds of its APIs [29].

2.5 TypeScript

TypeScript is a gradually-typed programming language [91, 90, 92] released by Microsoft in 2012 [7, 6]. It was designed as a syntactic superset of JavaScript, introducing static typing capabilities for identifying error-prone JavaScript constructs and addressing the deficiencies of developing large-scale applications in JavaScript.

TypeScript enriches JavaScript by introducing features such as interfaces, (enhanced) classes, a flexible module system, and a static structural type system [49]. Its module and type systems readily accommodate common JavaScript programming practices while also providing tooling and IDE experiences similar to those of languages like Java and C. For example, the type system aids in catching errors statically and offers additional support for program development, such as suggesting potential methods that can be called on an object in editor-like applications.

In its execution process, TypeScript source code is compiled into plain JavaScript code, which can be run on any browser or JavaScript runtime.

2.5.1 TypeScript Syntax and Semantics

TypeScript supports features such as interfaces, well-typed classes, union and intersection types, and every feature supported by JavaScript. This section discusses a few of these features without going into in-depth details.

Interfaces

TypeScript is structurally typed, therefore interfaces can be defined by modeling the shape of an object. Interfaces define contracts that specify what properties an object must have and help make code more maintainable. Figure 2.18 defines a `User` interface consisting of three properties.

```
1 interface User {
2   name: string;
3   age: number;
4   id: string;
5 }
```

Figure 2.18: An interface in TypeScript.

Now an object can be defined to conform to the shape of this interface:

```
    const user: User = {
        name: "Alan Turing",
        age: 23,
        id: "obxff",
    };
```

Since `user` is annotated with the `User` interface, if it happens to be missing any field of the interface, for example, `id`, the TypeScript compiler errors with `Property "id" is missing in type "{ name: string; age: number; }" but required in type "User".`

Classes

Classes in TypeScript work the same as in JavaScript, with the added benefit of static typing:

```
class Account {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

TypeScript supports inheritance using the `extends` keyword:

```
class UserAccount extends Account {
  age: number;
  readonly id: string;
  constructor(name: string, age: number, id: string) {
    super(name);
    this.age = age;
    this.id = id;
  }
}
```

Due to TypeScript's structural typing, the variable `user` can be annotated with type `User`, as it conforms to the shape of the class `UserAccount`.

```
    const user: User = new UserAccount("John Doe", 12, "bridgesort");
```

Fields in classes are public by default, but may be designated as private. Unlike private fields, public fields are accessible directly outside the class. Fields annotated with the `readonly` modifier can be accessed both inside and outside the class but cannot be reassigned after initialization. Since readonly members cannot be modified outside the class, they must either be initialized at declaration or within the class constructor.

Functions

In TypeScript, it is common practice to type function parameters, although it is not strictly required. Untyped parameters are implicitly assigned type `any`. However, in some cases, such as with higher-order functions, functions with untyped parameters may be inferred to more precise types because the surrounding code is already typed or inferable. This concept is known as contextual typing [75].

```
const numbers = [1, 2, 3, 4, 5];

// TypeScript infers the type of the array element `num` as number
// because it is in an array of numbers. This outputs: 2, 4, 6, 8, 10
numbers.forEach((num) => console.log(num * 2));
```

Unions and Intersections

A union type describes a value that can be one of several types:

```
type StringOrNumber = string | number;
let j:  StringOrNumber = "fox";
j = 5;
```

Intersection types enable the combination of multiple types into a single type, allowing for the extension of a type with additional features. Intersection types offer the convenience of extending types without the need to define a new class, as is required in inheritance:

```
type Animal = {
  name: string;
}
type Bear = Animal & {
  hasHoney: boolean;
}

const bear: Bear = {name: "Pooh", hasHoney: true};
bear.name;
bear.hasHoney;
```

2.5.2 Type System

The type system includes a number of advanced constructs and concepts, which include structural type equivalence (rather than nominal type equivalence), types for object-based programming (as in object calculi), gradual typing, subtyping of recursive types and type operators [49]. Collectively, these elements aim to enhance the web development experience.

The primary aim of TypeScript is not to introduce a new programming language but to augment and bolster JavaScript development [49]. This goal leads to a number of distinctive properties of the type system, some of which are discussed below.

Full type erasure [49]

Type erasure occurs when all types are removed from TypeScript code during compilation to JavaScript. TypeScript removes type annotations, interfaces, type aliases, and other type system constructs during compilation [27], leaving no trace of types in the JavaScript emitted by the compiler. Subsequently, there are no runtime representations of types, and hence no run-time type checking [49].

```
interface Cat {
  name: string;
}

class Lion {
  name: string;
}

let pet: Cat;
// OK, because of structural typing
pet = new Lion();
```

Type inference [49]

TypeScript supports type inference, which makes type annotations optional in several programming contexts. Type inference alleviates the need to fully annotate code, eliminating type clutter and improving readability. TypeScript uses a bidirectional type checking algorithm [61] and supports flow-sensitive typing or type narrowing [76, 79]. In practice, often only a small number of type annotations need to be provided for the compiler to infer meaningful type signatures [49].

Gradual typing [49]

TypeScript adds an object-oriented gradual type system to JavaScript [81]. Although gradual typing typically incorporates run-time checks when transitioning between typed

and untyped code, TypeScript does not do this due to type erasure. As a result, typing errors not identified statically may remain undetected until runtime [49].

Unsoundness

TypeScript encourages programming patterns that avoid reliance on runtime metadata and aims to strike a balance between correctness and productivity [16], prioritizing practicality over soundness. Consequently, its type system is considered unsound [49], lacking robust type safety guarantees. This unsoundness allows for the inclusion of unsafe operations that cannot be fully verified at compile-time, necessitating runtime checks. Furthermore, in order to maintain compatibility with the existing JavaScript ecosystem, TypeScript incorporates design decisions such as type casting, which contribute to the unsoundness of its type system [49].

Type casting

Typescript includes an `any` type that permits unsafe casting between types. Types can be upcasted (widened) or downcasted (narrowed). However, since TypeScript fully erases types during compilation, there are no runtime checks to ensure the safety of these casts. The following code snippet demonstrates unsafe type casting valid in TypeScript:

```
let j = "foo";
let b = (j as any as number) * 2;
// prints NaN
console.log(b);
```

The cast to `any` is necessary because TypeScript does not allow casting directly from type `string` to `number`, as they are disjoint types. When executing the code, the result is the value `NaN`, which ideally should not occur if the cast were caught statically or with runtime type checks.

2.6 Nominal Typing

Given two types $\texttt{T}_1$ and $\texttt{T}_2$, in a nominal, or nominative, type system, these types are considered the same if they share the same name. Type $\texttt{T}_1$ is considered a subtype of type $\texttt{T}_2$ if $\texttt{T}_1$ is explicitly declared to be a subtype of $\texttt{T}_2$.

However, in the context of nominal types in this thesis, additional constraints based on heuristics discussed in Section 3.3, which aid in property access optimizations, are imposed due to the semantics of TypeScript's type system. Nominal types remain *stable* and are not subject to mutation. These types enable safe optimization of objects because guarantees about their properties hold consistently throughout their usage.

2.7 Why TypeScript?

Even though the type system is unsound, it has proven to be very useful in practice [49]. Futhermore, integrating soundness into the type system is not without its own costs.

Rastogi et al. [81] developed Safe TypeScript, a sound gradual type system built on TypeScript, by enforcing stricter static checks and embedding residual runtime checks in generated code. Although Safe TypeScript can catch more type errors at compile-time and runtime, it also incurs a runtime overhead of up to 15% [81].

Compiler optimizations are often tailored to specific scenarios based on defined heuristics. For instance, prediction propagation in JavaScriptCore, which predicts types, is inherently unsound because the resulting types are mere predictions. However, this process is optimized to be useful (i.e., to make good predictions) by leveraging the results of value profiling within JavaScriptCore's optimizing JIT compilers [15].

When used in a type-correct manner, TypeScript can be leveraged for cooperative optimization between the compiler and the executing JavaScript engine. In such scenarios, TypeScript produces performance-oriented JavaScript code by harnessing runtime intrinsics of the underlying engine for performance, based on type properties encountered in the TypeScript code.

Consider the following code snippet:

```
class Book {
  constructor(public title: string, public author: string) {}
}

const austen = new Book("Pride and Prejudice", "Jane Austen");
console.log(austen.title);
```

If the `Book` object is always used strictly according to its type as defined by its constructor, it can be optimized to eliminate runtime checks related to the object's shape during property access operations.

This thesis focuses on optimizing JavaScript performance in JavaScriptCore by leveraging TypeScript and its type system, developing and implementing heuristics for cooperative optimization in `tsc` and JavaScriptCore.

Chapter 3

Implementation

This chapter presents the primary contributions of the thesis, focusing on the concept of leveraging type information from TypeScript for optimization in JavaScriptCore. Additionally, it discusses the modifications made to both the TypeScript compiler and the JavaScriptCore engine. Specifically, it delves into the heuristics developed for optimizing JavaScript, changes to TypeScript's type checker and emitter, as well as modifications to JavaScriptCore's LLInt and Baseline tiers. The code presented in this chapter mostly comprises incomplete pseudocode programs, omitting details that do not directly contribute to or advance the explained concepts.

3.1 Heuristic Guided Optimization

The object model, specifically object shapes (structures), plays a pivotal role in the runtime performance of JavaScriptCore. Therefore, this thesis primarily focuses on optimizations related to object structures, particularly property access operations. However, the underlying principle developed in this work is applicable to other forms of optimization.

Figure 3.1 presents a JavaScript program demonstrating property access on an object. The object `obj` is consistently typed, and its memory layout in JavaScriptCore is depicted in Figure 3.2. The properties `x` and `y` of the object are defined near the object's allocation, that is, within the class constructor.

The JavaScript VM uses the structure ID of the object to look up its structure in a table maintained by the VM when accessing the value of the property `x` in the object `obj` in Figure 3.1. Once the specific structure is identified, the VM accesses property table associated with that structure. This property table, implemented as a hash table, contains pointers from property names to indices in the object's inline slots or its out-of-line storage slots. In Figure 3.3, the property `x` maps to an inline slot index of 0 in the structure's property table, and inline slot 0 holds the value 5.

```
class Point {
   constructor(x, y) {
      this.x = x;
      this.y = y;
   }
}
const obj = new Point(5, 2);
obj.x;
```

Figure 3.1: JavaScript class with two properties demonstrating property access.

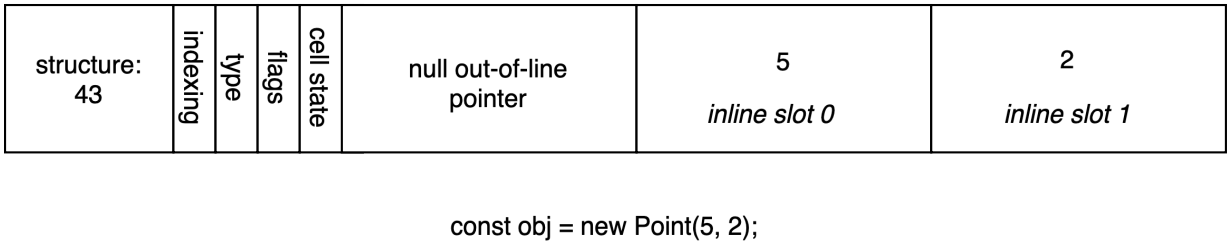

Figure 3.2: Layout of `obj` in JavaScriptCore.

This method of property access in JavaScriptCore is optimized for inline caching. Objects' structure IDs and property offsets accessed on such objects are easily cached. As a result, repeated property accesses do not traverse the entire lookup chain; instead, they use the cached structure ID and property offset after a successful structure check. However, when objects are assigned new properties at runtime or far from their allocation site, the assignment mutates the structure of these objects. Consequently, their previously cached structure ID and property offsets no longer apply. Therefore, future property accesses must traverse the full lookup chain again if the object's new structure ID and the offset of the property being accessed do not already exist in the inline caches. This makes inline caches more polymorphic. The LLInt and Baseline tiers of JavaScriptCore impose limits on the extent of polymorphism their inline caches can handle, limiting the use of inline caches for polymorphic objects and functions.

If the object layout, as well as the offset of the property being accessed, is statically known in the underlying engine, it becomes possible to optimize property access operations to completely avoid the full lookup chain without relying on inline caches and structure checks. This optimization is achievable using TypeScript.

TypeScript can bridge the gap between statically or compile-time known types, which are not immediately available to the JavaScript engine for runtime optimizations, and the additional runtime type inference performed by the JavaScript engine. Since type information is statically known and available in TypeScript, it can be leveraged at the engine level to eliminate redundant runtime operations and improve runtime performance, especially around object structures and property accesses.

By utilizing type information, an object's layout can be carefully constructed so that its known properties are stored at statically computable offsets within the object's structure at runtime. Property access can then be made efficient by transforming such operations into C-like indexing operations using the structure ID and the compile-time known offset of the property. Figure 3.4 illustrates this optimization process.

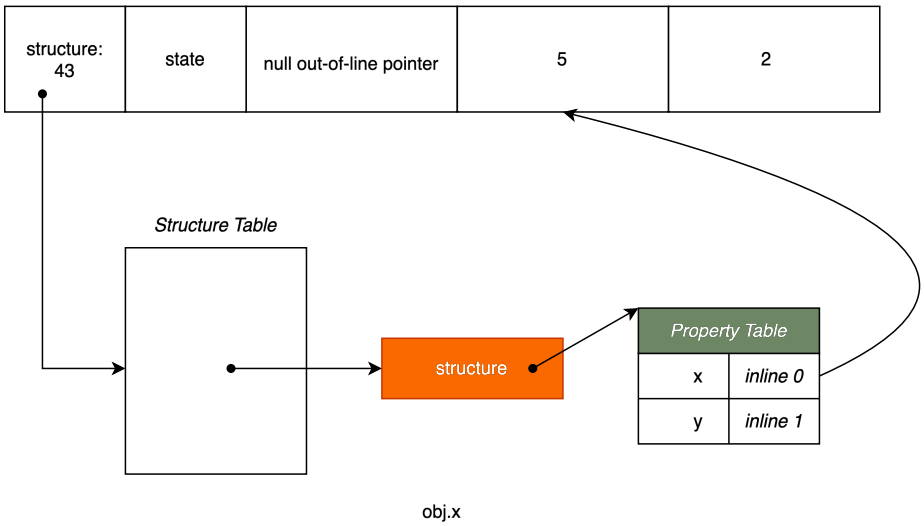

Figure 3.3: Regular property access as seen in `obj.x`.

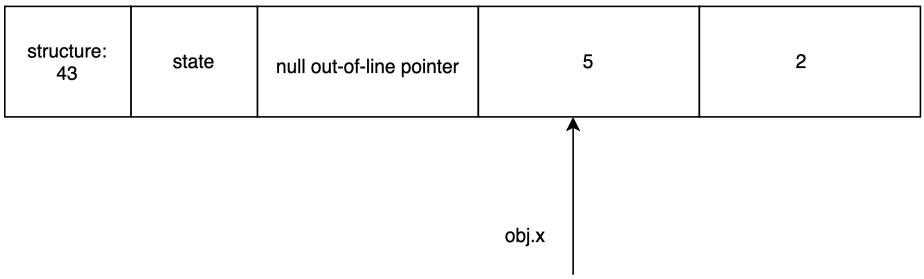

Figure 3.4: Optimized property access for `obj.x`.

The problem with the aforementioned optimization is twofold: the dynamic nature of JavaScript and the unsound (unsafe) nature of TypeScript. Due to its dynamic nature, JavaScript allows properties to be defined on objects dynamically or far from the object's allocation site. There is no guarantee that such properties can be stored in the (limited) inline slots of the object. While defining properties in this manner is valid in JavaScript, it compromises the integrity of the object's structure by modifying its shape.

In JavaScriptCore, objects are allocated a specific number of inline slots based on a straightforward analysis conducted at the allocation site [15]. For objects instantiated using classes, this analysis is typically performed within the constructor. Therefore, when

```
class Cat {
   constructor(name) {
      this.name = name;
   }
}

const cat = new Cat("Tangerine");
// ... several lines later ...
cat.eats = true;
```

Figure 3.5: Example of dynamic property assignment.

a property is assigned to an object outside or far from its constructor, as illustrated in Figure 3.5, it is unlikely that the value of the property is stored in the object's inline slots. Properties that are not stored in inline slots are stored out-of-line [15]. Furthermore, the optimization process is complicated by the ability to remove object properties at runtime using the `delete` keyword in JavaScript, as demonstrated in Figure 3.6.

```
class Cat {
   constructor(name) {
      this.name = name;
   }
}

const cat = new Cat("Tangerine");
delete cat.name;

// shows undefined
console.log(cat.name);
```

Figure 3.6: Deleting an object's property in JavaScript.

While TypeScript does not permit attachment of properties not belonging to an object's type (though type casting can sometimes circumvent this behaviour), as an extension of JavaScript, it does allow the deletion of properties. However, only optional properties or properties with types containing the value `undefined` can be deleted. Deleting an optional property or a property with the type `undefined` from an object can also alter the object's shape. As a result, there is no guarantee that a property actually exists in either storage (inline or out-of-line) during property access. In Figure 3.7, the type `Cat` includes a property `name` of type `undefined`. Therefore, deleting this property is considered valid in TypeScript.

```
class Cat {
   constructor(name: string | undefined) {
      this.name = name;
   }
}

const cat = new Cat("Tangerine");
delete cat.name;

// shows undefined
console.log(cat.name);
```

Figure 3.7: Deleting an object's property in TypeScript.

To address these issues, the TypeScript compiler must invoke optimizations only when object types are consistently used and when access operations involve properties that are guaranteed to exist on an object. Type consistency can be achieved by using classes in TypeScript as if they are classes written in languages like Java and C++. For this purpose, TypeScript types are classified into two categories: nominal and non-nominal. Types that fall under the nominal category are similar to class types in Java and C++, while non-nominal types include structural or arbitrary types. Compared to non-nominal types, nominal types are more amenable to optimizations. Therefore, the approach in this thesis uses nominal type information for optimizations through the application of special heuristics.

3.2 Type Classification

Nominal types represent types consistently utilized across all usage points. In contrast, non-nominal types conform to the usage expected by the TypeScript compiler, representing the default behaviour. Additionally, non-nominal types are associated with dynamic and potentially unsafe type usage. These distinct patterns of type usage are illustrated in Figures 3.8a, 3.8b and 3.8c.

Nominal types contribute to typing consistency and concreteness, unlike non-nominal types. In TypeScript, type casting implicitly or explicitly presents a significant soundness issue, and is often used to bypass the type checker and perform operations not permitted prior to the cast. A common example is casting to `any`, as discussed in Section 2.5.2.

While nominal types may also be subject to unsafe type casting, such as downcasting an object's field or property or the object's type itself, nominality checks (discussed in Section 3.6.1) ensure that optimizations are only applicable in cases where nominal type usage is guaranteed and enforced. In the optimizations developed in this thesis, I prefer

nominal types over non-nominal types. This preference stems from their ability to facilitate the application of optimizations to dynamic JavaScript, akin to statically typed languages such as C++ and Java.

```
let fby: any = {};
for (const p of ["c", "am"]) {
    x[p + "ount"] = 12;
}
fby.deductions = 14;
compute(fby.amount - fby.deductions);
```

(a) Non-nominal with arbitrary types.

```
type Fooby = {amount: number};
let fby2: Fooby = {
    unit: "dollar",
    fox: 23,
} as Fooby;
compute(fby.amount);
```

(b) Non-nominal with structural types.

```
class Fooby {
    constructor(public readonly amount: number){}
}
let fby = new Fooby(23);
compute(fby.amount);
```

(c) Nominal with classes.

Figure 3.8: Classification of types according to usage.

Property existence and layout is ensured by optimizing property accesses when they *fully* comply with nominal type usage. A type satisfies nominal type usage when it meets the heuristics discussed in Section 3.3. Types resembling nominal types can undergo shape mutation through the deletion of properties or fields. During type checking and nominal type analysis, `mtsc` flags nominal types with deleted properties and eliminates all optimizations on such types.

3.3 Heuristics

Structure and access optimizations are applicable when objects experience nominal type usage through classes at all access points. Nominal type usage involves the consistent use of fields or properties as originally typed from the class constructor.

Object layout information is crucial for the optimizations presented in this thesis. This information, derived from an object's fields or properties, is employed to construct the object during runtime. However, utilizing layout information for optimization of access operations can present challenges, particularly when deleting a field or property from the object. In such cases, the JavaScript VM may store new properties in the slot previously occupied by the deleted property. This issue becomes evident when attempting to access

the deleted property again. Instead of retrieving `undefined` as specified in the JavaScript specification [62], the value of the new property occupying the deleted property's slot is returned, thus violating the expected semantics. Hence, objects created from such classes must not undergo mutation concerning their fields, such as through field deletion.

An additional requirement imposed by my approach for optimization is that access operations involve base class objects, i.e., objects from classes not included in an inheritance chain. This requirement is due to the increased complexity in object layout introduced by inheritance, which affects the object model along an inheritance chain. Such objects could be optimizable using similar techniques in the future.

3.4 Case Study: Nominal Types

A type satisfies nominal type usage if objects created from it are used nominally according to the heuristics discussed in Section 3.3. During nominal type analysis, types undergo scrutiny against both explicit and implicit type casts within assignment statements and cast expressions. In TypeScript, any type can be cast to the type `any`. Such casting serves as a means to circumvent the type checker, allowing operations might be disallowed by the checker. For instance, casting to type `any` may enable the deletion of properties that are not initially optional or did not contain the `undefined` type.

Tracking these types and the operations performed on them becomes challenging, making it difficult to define corresponding object layouts for access optimizations. Apart from casting to `any`, types may also be cast to structural types that precisely match their shape. While such casts are considered safe in TypeScript, they pose the same tracking challenge mentioned earlier.

Furthermore, these casts may subsequently be upcast to `any` or a broader structural type to execute unsupported operations on the original type. As this chain of possibilities expands with each cast, I have adopted a conservative approach for nominal type usage. Namely, for a type to satisfy nominal type usage, it must refrain from any form of casting, whether implicit or explicit, except for casting to itself (an operation that is of little utility).

3.4.1 Explicit Type Casts

In Figure 3.9, the variable `p` has the type `Point`. Before line 13, `Point` may be classified as a nominal type because `p` remains consistent from its construction. However, at line 13, `p` is cast to `any`, and its property `x` is then deleted. It is impossible to delete any of the properties of `p` without casting to `any`, as the types of the properties `x` and `y` do not include `undefined`.

Similarly, in Figure 3.10, the variable `ab` is first cast to `any` and then assigned to a new variable. This variable is subsequently used to delete the property `x`.

Both of these cases fail nominal type checks. The code in Figure 3.10 fails nominal type checks even if line 13 is removed. This failure stems from the cast to `any` at line 12.

```
1  class Point {
2      public x: number;
3      public y: number;
4
5      constructor(a: number, b: number) {
6          this.x = a;
7          this.y = a;
8      }
9  }
10
11 let p = new Point(1, 2);
12 // below line fails nominal type checks
13 delete (p as any).x;
14 p.x + p.y;
```

Figure 3.9: Direct property deletion through cast to `any`.

```
1  class Point {
2      public x: number;
3      public y: number;
4
5      constructor(a: number, b: number) {
6          this.x = a;
7          this.y = a;
8      }
9  }
10 let ab = new Point(12, 13);
11 // below line fails nominal type checks
12 let ab2 = ab as any;
13 delete ab2.x;
14 ab2.x + ab2.y
```

Figure 3.10: Indirect property deletion through cast to `any`.

In Figure 3.12, a program features a class `Point` and two interfaces, `PointLike` and `PointLike2`, with similar shapes to `Point`. In Figure 3.11a, the `Point` type is cast to the type `PointLike`. Furthermore, `PointLike` is cast to `any`, and the property `y` of `Point` is deleted.

Similarly, in Figure 3.11b, the same operations are executed. However, unlike the `PointLike` interface in Figure 3.12, the `PointLike2` type contains optional fields that can be directly deleted. Hence, casting to `any` is unnecessary. In both cases, the type `Point` fails nominal type usage due to the cast and delete operations.

```
1 let cd = new Point(3, 5);
2 // below line fails nominal type checks
3 let px = <PointLike> cd;
4 delete (px as any).y;
5 cd.x + cd.y
```

(a) Cast to identical structural type.

```
1 let cd1 = new Point(3, 5);
2 // below line fails nominal type checks
3 let px1 = <PointLike2> cd1;
4 delete px1.x;
5 cd1.x + cd1.y
```

(b) Cast to type with optional fields.

Figure 3.11: Structural type casts in TypeScript.

```
class Point {
    public x: number;
    public y: number;

    constructor(a: number, b: number) {
        this.x = a;
        this.y = b;
    }
}

interface PointLike {
    x: number;
    y: number;
}

interface PointLike2 {
    x?: number;
    y?: number;
}
```

Figure 3.12: Class and interfaces with similar types.

Nominal type checks not only inspect cast expressions but also assignment statements. In Figure 3.13, the `Point` variable `ef` is assigned to a variable `gt` typed `any`. This assignment allows operations on `gt` that would be flagged by the type checker if performed on `ef`. `mtsc` detects this by inspecting types of the source (RHS) and target (LHS) of an assignment during nominal type checking. As with previous cases, this example also fails nominal type usage.

```
class Point {
    public x: number;
    public y: number;

    constructor(a: number, b: number) {
        this.x = a;
        this.y = a;
    }
}
let ef = new Point(5, 7);
// below line fails nominal type checks
let gt: any = ef; // type `Point` coerced to `any`
delete gt.x; // now delete becomes possible here
ef.x + ef.y
```

Figure 3.13: Type coercion to `any` through assignment.

3.4.2 Implicit Type Casts

The program in Figure 3.14 almost satisfies nominal type usage. However, the `Point` variable `qx` is passed as an argument to a function with a single parameter typed `any`.

In this case, the `Point` type is implicitly coerced to type `any`, permitting any kind of operation on the object. If line 17 is removed, then the `Point` type satisfies nominal type usage at all points in the program.

```
1  class Point {
2      public x: number;
3      public y: number;
4
5      constructor(a: number, b: number) {
6          this.x = a;
7          this.y = b;
8      }
9  }
10
11 function breakThings(p: any) {
12     delete p.x;
13 }
14
15 let qx = new Point(12, 13);
16 // below line fails nominal type checks
17 breakThings(qx);
18 qx.x + qx.y
```

Figure 3.14: Type coercion in function parameters.

In JavaScript and TypeScript, objects are assigned by reference. Figure 3.15 presents a subtle case of implicit type coercion. In line 11, the variable `qj` is implicitly typed as `any` because it is not yet assigned to any type.

However, assigning the `Point` variable `qx` to `qj` does not change the type of `qj`. TypeScript simply coerces the type of `qx` to `any`. Since `qj` is typed as `any`, it can be used to perform any kind of operation permitted on the `any` type. Thus, the `Point` type fails nominal usage in this case.

```
class Point {
    public x: number;
    public y: number;

    constructor(a: number, b: number) {
        this.x = a;
        this.y = b;
    }
}

let qj; // implicitly typed `any`
let qx = new Point(12, 13);
// below line fails nominal type checks
qj = qx;  // qx assigned to type `any`
qx.x + qx.y;
```

Figure 3.15: Type coercion in assignment.

3.4.3 Delete Operations

In some cases, a nominal type may fail to satisfy nominal type usage even without explicit and implicit casts. In Figure 3.16, the `Point` object almost passes nominality checks but fails at line 15 when the property `y` of the `Point` object `pt` is deleted.

```
class Point {
    public x?: number;  // x has an optional type
    public y?: number;  // y has an optional type

    constructor(a: number, b: number) {
        this.x = a;
        this.y = b;
    }
}

let pt = new Point(12, 13);
// ! is the non-null assertion operator which tells tsc that pt.x is non-null
console.log(pt.x! + pt.y!);
// below line fails nominal type checks
delete pt.y;
console.log(pt.x, pt.y);
```

Figure 3.16: Deleting the property `y` of the `Point` object `pt`.

While some of these examples may appear restrictive, this conservative approach to nominality checks is implemented to ensure that optimizations are only applied to operations that are guaranteed to be safe.

3.5 Modifications

This work explains modifications made to both the TypeScript compiler and the JavaScriptCore engine in WebKit to support this thesis. The TypeScript compiler is adjusted to produce optimized JavaScript code for TypeScript programs that meet previously mentioned heuristics. This optimization involves restructuring the object layout to unlock further possibilities. The restructured layout enables optimized property access operations by leveraging intrinsics developed for the underlying JavaScript engine, JavaScriptCore. The implementation of the concepts presented in this thesis in the TypeScript compiler and the JavaScriptCore engine, excluding configuration, build, and installation scripts, totaled approximately 350 lines of code: 200 lines in the TypeScript compiler and 150 in JavaScriptCore. The remainder of this chapter examines these modifications in detail.

3.6 TypeScript

This section discusses the concepts and implementation details of the extensions to `tsc`. The extended TypeScript compiler is called `mtsc`.

3.6.1 Checking Nominality

Nominality checks inspect various expressions and statements within a TypeScript program to determine if types are used nominally based on previously developed heuristics. Nominality checks scrutinize the types used in expressions such as casts, function calls, property deletions, assignments, and property accesses. If a type violates nominal type usage in any of these expressions, it is flagged. In such cases, if layout information has already been derived for the type, the compiler discards the information and does not apply optimizations to the type.

Several examples demonstrating violations of nominal type usage are discussed in Section 3.4. However, if a type satisfies nominality checks by not violating nominal type usage, the compiler computes layout information for the type at property access points. This information is not only utilized in creating the object's layout at runtime but also employed to optimize the access expression. This optimization involves determining the slot in the computed layout where the property is located and fetching the property directly using runtime intrinsics.

Nominal type detection is implemented within TypeScript's type checking pass by identifying nominal type patterns using an indirect whole-program analysis approach [45]. To accomplish this, the type checker is modified to examine nominal type usage in specific expressions and statements. This strategy eliminates the complexity of an additional analysis pass, which would need to consider all TypeScript constructs, a task exacerbated by TypeScript's lack of a compact intermediate representation. Additionally, it ensures low overhead and lazy computation.

Figure 3.17 presents pseudocode for the function used by the TypeScript compiler to check expressions and statements. This function acts as a dispatcher, by inspecting the type of an AST node, and invoking the corresponding type checking function on such nodes.

```
function check(node: AST) -> Type {
    const kind = node.kind;
    if kind is Identifier:
        return checkIdentifier(node)
    else if kind is DeleteExpression:
        return checkDeleteExpression(node)
    else if kind is QualifiedName:
        return checkQualifiedName(node)
    else if kind is ElementAccessExpression:
        return checkElementAccess(node)
    else if kind is PropertyAccessExpression:
        return checkPropertyAccessExpression(node)
    // several other checks...
    else:
        FAIL
}
```

Figure 3.17: Pseudocode for type checking expressions.

The type checker examines property access expressions and determines their type using a utility function. This function not only handles the type checking of property access expressions, but also that of expressions resembling property access (see Appendix A) such as a qualified name (e.g., namespace `Foo.Bar`). Nominal type detection is performed only after the property access expression has been successfully type-checked.

If an access expression has been successfully type checked, the type checker populates a locally defined `prop` variable, which is a `Symbol` (see Appendix A), with the resolved and linked property name.

Additionally, assignment targets, such as `foo.bar = 3`, are excluded from the analysis, as optimization efforts focus on access operations within expressions (or RHS sources) that are not being assigned to (`isAssignmentTarget(..)`). Furthermore, the expression

is verified to ensure it does not involve property access on a `this` variable within a constructor by utilizing the function `isThisPropertyAccessInConstructorStrict` depicted in Figure 3.18. This precaution prevents the optimization of field access on an object that may not have been fully constructed or initialized. Figure 3.19 illustrates property access expressions in a class constructor that would not be optimized.

```
function isThisPropertyAccessInConstructorStrict(node: PropertyAccessExpression) {
    return isThisProperty(node) and getThisContainerKind(node) is Constructor;
}
```

Figure 3.18: Pseudocode for checking if a property access is within a constructor.

```
class Show {
    margin: number;
    constructor(public item: string) {
        this.margin = this.item.length / 4;
    }
}
```

Figure 3.19: `this` property access in a class constructor.

Given the type of the object involved in a property access operation, denoted as `typ`, the operation satisfies nominal type usage and becomes optimizable when three criteria are met:

1. `typ` is class-like or satisfies class-like properties (`satisfiesClsTy`).
2. `typ` is also an eligible class type (`isEligibleClsTy`).
3. `typ` is not flagged for shape mutation resulting from property deletion (`isNotFlaggedTy`).

```
const satisfiesClsTy = (
    !isBuiltinTy && !isAbstract && isClassLikeTy &&
    !(["name", "length", "caller", "arguments", "prototype"].includes(_name))
);
```

Figure 3.20: Derivation of `satisfiesClsTy`.

A check is performed to ensure that the apparent type is not a built-in type (`isBuiltinTy`), as shown in Figure 3.20. This check involves verifying whether the type's name is present in the built-in globals symbol table.

TypeScript supports abstract classes, which are classes that cannot be instantiated directly. They are intended to be extended by other classes, which then provide implementations for the abstract methods declared in the abstract class [74]. Since the heuristics defined in Section 3.3 aim to provide optimization for concrete nominal classes, properties found on abstract classes are automatically deemed ineligible and filtered out. Determining whether a class is abstract (`isAbstract`) is achieved by checking if the declaration associated with `typ` includes an `Abstract` syntactic modifier flag.

In Figure 3.20, `isClassLikeTy` verifies that the type of the left-hand side of the property access expression, `typ`, genuinely represents a class type or resembles one. This check is achieved by verifying that the type's instantiations, base constructors, and target properties are defined, indicating that they are not `undefined`, as is typically the case in nominal classes.

In TypeScript, all properties of an object are available in the type of the object itself, including built-in properties. If a collision occurs, where a property name clashes with a built-in name, the nominality check is aborted. For instance, `arguments` is a built-in property associated with functions in JavaScript, and mistaking it for a user-defined property name may lead to unintended behaviour. To mitigate this risk, a conservative approach is taken by ensuring that any collision aborts the check entirely.

Hence, an object's type (`typ`) is considered class-like or satisfies class-like properties under the following conditions:

1. The apparent type `typ` is not a built-in type.
2. The declaration containing the field or property is not abstract.
3. The type is nominal or class-like.
4. The property name found does not conflict with very common names associated with built-in types.

If `typ` satisfies the first condition (`satisfiesClsTy`), it undergoes further inspection to determine its eligibility for optimization. `typ` qualifies as an eligible class type if it is either a base class or a class not included in an inheritance chain. Figure 3.21 shows the derivation of class eligibility.

The third criterion is verified by ensuring that the type has not been flagged. In nominality checks, flagged types are those that may satisfy the first two nominal type usage criteria but undergo shape mutation due to property deletion or are used in a type cast, whether implicit or explicit. These types are monitored during the type checking of function calls, cast expressions, delete expressions, as well as assignment expressions and

```
const isEligibleClsTy = (
    (typ.hasResolvedBaseTypes() or (isThis && typ.hasPropertyNodes())) // valid class
    and typ.hasNoHeritageClauses() // inheritance
);
```

Figure 3.21: Pseudocode for deriving `isEligibleClsTy`.

statements. In property access expressions, the type of the object whose property is being accessed is checked to ensure it has not been flagged during these other checks. Figure 3.22 illustrates the verification process for flagged types.

Additionally, AST nodes for which layout metadata has been computed during the check are also monitored. If a type is flagged, any optimization metadata applied to nodes of that type is erased. Satisfying these criteria implies meeting nominality check requirements.

```
const isNotFlaggedTy = !flaggedTypes.has(typ.symbol.id);
```

Figure 3.22: Checking for flagged types.

3.6.2 Optimization Metadata

If the three criteria (`satisfiesClsTy`, `isEligibleClsTy`, and `isNotFlaggedTy`) are met, the next step is to compute layout metadata for the class type (`typ`), if it has not already been computed, and to determine the offset of the property being accessed relative to the computed layout.

The layout metadata is obtained in two steps. First, the properties of `typ` are retrieved using its `declaredProperties` or `properties` fields, which are arrays containing all properties (fields and methods) defined in the type as `Symbol`s. In certain cases, such as during type checking a `this` property access expression, `declaredProperties` may be empty or `undefined`, in which case `properties` serves as a fallback.

Since `declaredProperties` or `properties` may also include class methods, the array is filtered to include only properties or fields (see Appendix B). As fields in TypeScript can be declared and initialized in various places, such as outside a class constructor or directly within the constructor parameter list, the ordering of initialization statements in a class constructor emitted as JavaScript can be somewhat arbitrary or non-deterministic. To ensure that the object layout is deterministic and computable for optimization purposes, the filtered properties are sorted according to their order of declaration, i.e., in the order

they appear in the original TypeScript source. This is crucial because the order of field initialization in a class constructor determines its layout in JavaScriptCore.

The obtained properties, represented as `Symbol`s, are mapped to their textual names using their `escapedName`. The resulting array, `propertyNames`, serves as layout information during the JavaScript emitting phase. This array is attached as metadata to the `constructor` node associated with the class obtained from `typ`. The number of properties in this array is limited to ensure it does not surpass the maximum number of inline slots for an object in the version of JavaScriptCore utilized, which is 64 (`maxInlineSlots` in Figure 3.23). Therefore, layout information is only emitted for properties that do not exceed this limit. Additionally, access optimizations are only applied to properties that fall within this limit. Consequently, properties that fall outside this limit are stored out-of-line in JavaScriptCore.

```
// First Node is the property access expression, second is the constructor
type OptInfo = [Node, Node];
// store types flagged against nomality checks - key is the type id
const flaggedTypes = new Map<number, Type>();
// store all nodes for which layout metadata has been computed - key is the type id
const computedLayouts = new Map<number, OptInfo[]>();
// max inline slots
const maxInlineSlots = 0x40;
```

Figure 3.23: Layout, flagged types, and maximum inline slots definitions.

In certain scenarios, some classes do not define their constructors, either having all fields statically initialized or having no fields at all. In the former case, the metadata is attached to the node corresponding to `typ` itself. The layout metadata computed during nominality checks is stored in `computedLayouts` (Figure 3.23) for effective tracking and erasure casting and during shape mutation.

However, for optimizing the property access expression itself, the offset of the property name is obtained from the resulting array (`propertyNames`). This offset corresponds to the offset in the inline slots where the value associated with the property in the access expression would be found, if laid out accordingly. The offset is computed using the helper function `getOffset`, shown in Figure 3.24.

```
const getOffset = (props: string[], ppty: string) => {
    const index = props.indexOf(ppty);
    return index !== -1 ? index : undefined;
};
```

Figure 3.24: Implementation of `getOffset`.

If the offset returned by the function is not `undefined`, the property (`Symbol`) is retrieved from the modified properties array and inspected for getters and setters. Getters and setters in JavaScript and TypeScript are special methods for encapsulating object properties and are invoked using property access syntax instead of method call syntax. If the expression is indeed not a getter or setter access, the obtained offset, along with the property name, is attached as metadata to the AST node of the property access expression in preparation for the emitter. Figure B.1 presents the implementation of this metadata extraction process.

```
// get the property offset
const offset = getOffset(propertyNames, name);
if (offset is not undefined) {
    // ensure that the property is not a setter / getter
    const sym = properties[offset];
    const getter = obtainGetAccessor(sym);
    const setter = obtainSetAccessor(sym);
    if (is not getter and is not setter) {
        // attach access optimization metadata
        node.accessMetadata = { property: name, offset };
        const typeID = getTypeID(typ);
        const constructor = getConstructorOfType(typ);
        const target = constructor or getTypeDeclaration(typ);
        if (target has no layout) {
            target.layoutMetada = propertyNames;
        }
        if (typ not in computedLayouts) {
            computedLayouts.set(typeID, [[node, target]]);
        } else {
            const saved = computedLayouts.get(typeID);
            saved.push([node, target]);
        }
    }
}
```

Figure 3.25: Pseudocode for attaching layout and offset metadata to a property access node.

The nominality checks, layout, and property offset computation aim to transform the code illustrated in Figure 3.26a into the structure depicted in Figure 3.26b. By ensuring that the ordering of constructor field initialization statements emitted by the emitter is statically determinable and aligns with the offset metadata generated during type checking, property optimization using offsets becomes feasible.

```
class Point {
   constructor(x, y) {
      this.x = x;
      this.y = y;
   }
}
const obj = new Point(5, 2);
obj.x;
obj.y;
```

(a) Class with field initialization statements in constructor.

```
// effectively an index access
obj[0]; // x
obj[1]; // y
```

(b) Optimization denotation.

Figure 3.26: Decomposition of property access expressions.

3.6.3 Shape Mutation

Shape mutation is detected by inspecting delete expressions during type checking. To accommodate this behaviour, I extended the function responsible for type-checking delete expressions.

```
function checkDeleteExpression(node: DeleteExpression): Type {
    // ..
    const expr = getTheExpression(node);
    if (isPropertyAccessExpression(expr)) {
        const type = check(expr);
        const typeID = getTypeID(type);
        const layouts = computedLayouts.get(typeID);
        if (layouts is not undefined) {
            for (const data of layouts) {
                removeAccessMetadata(data);
                removelayoutMetada(data);
            }
        }
        if (not flaggedTypes.has(typeID)) {
            flaggedTypes.set(typeID, type);
        }
    }
}
```

Figure 3.27: Pseudocode for detecting shape mutation through deletion.

When encountering a `delete` expression, the function examines the expression itself. If the expression represents a property access, the type on the left-hand side is obtained

by type checking the expression and added to the flagged types tracker (`flaggedTypes`) using its `symbol` ID, if not already present. Additionally, if layout and offset optimization metadata have been computed for such a type, the metadata is erased entirely. This process is part of ensuring strict adherence to nominal type usage during class optimization.

Property access optimization specifically targets properties guaranteed to be in inline slots—those laid out by the TypeScript compiler itself. This approach not only avoids disruption from foreign properties, which are not subject to optimization, but also ensures that properties are exactly where the TypeScript compiler expects them to be at runtime when invoking optimizations on access operations. Consequently, shape mutation that adversely affects access optimization only occurs through deletion. Figure 3.27 presents the pseudocode for detecting shape mutation and erasing optimizations.

3.6.4 Emitting JavaScript

Constructors

The function responsible for emitting class constructors is extended to emit layout code for the constructor of a class that has layout metadata. The layout metadata, attached during type checking, is retrieved from the constructor node. Each field in the array is initialized as follows: for every field or property specified in the array, an intrinsic function call, `$__putByIdDirect`, is emitted to store the field in inline-slots by assigning it the value `undefined`. The value `undefined` is obtained by mapping a field to its corresponding offset that is subsequently cast to `void`. A `void` cast in JavaScript is semantically equivalent to `undefined`. Assigning `undefined` using this method is especially helpful for debugging purposes.

```
1  class Point {
2     x: number;
3     y: number;
4     // computed layout: ['x', 'y']
5     constructor(ax: number, ay: number) {
6        this.x = ax;
7        this.y = ay;
8     }
9  }
10 const obj = new Point(5, 2);
```

Figure 3.28: class `Point` with field initializations matching computed layout information.

Generating layout instructions ensures that an object's shape in JavaScriptCore conforms to the layout expected for the application of property access optimizations. However,

in some cases, certain objects already conform to the desired layout. For instance, in TypeScript, the constructor of an object may contain field initialization statements in the order in which they are defined outside the constructor, as seen in Figure 3.28. In such scenarios, layout instructions are not generated for the corresponding objects.

```
function emitConstructor(node: ConstructorDeclaration) {
    emitModifiers(node, node.modifiers);
    writeKeyword("constructor");
    {
        const body = node.body;
        const layoutMetadata = getLayoutMetadataFromNode(node);
        if (layoutMetadata and body and isBlock(body)) {
            if (not alreadyHasLayout(layoutMetadata)) {
                const statements = emitLayoutMetadata(layoutMetadata);
                updateStatements(body.statements, statements);
            }
        }
    }
    emitSignatureAndBody(node);
}
```

Figure 3.29: Pseudocode for emitting a constructor with extension for layout metadata.

In Figure 3.30a, the class `Point` has the layout shown in Figure 3.30b. Although the layout information may appear redundant for simple and straightforward code like in Figure 3.30a, it proves valuable for non-trivial or complex constructors. In such cases, where fields are declared and initialized outside the class constructor, or where field initialization involves control flow leading to non-deterministic order of field initializations, having the layout information guarantees determinism.

```
class Point {
   x: number;
   y: number = 2;
   constructor(ax: number) {
      this.x = ax;
   }
}

const obj = new Point(5);

```

(a) Class in TypeScript.

→

```
class Point {
   constructor(x, y) {
      $__putByIdDirect(this, "x", void 0);
      $__putByIdDirect(this, "y", void 1);
      this.y = 2;
      this.x = x;
   }
}
```

(b) Layout in JavaScript.

Figure 3.30: Transformation of class fields to object layout instructions.

Intrinsics

`getByIdOffset`

Figure 3.31 presents pseudocode that extends the function used by the compiler for emitting access expressions. Property access expression nodes with a metadata payload attached by the type checker during nominality detection are emitted differently from those without the access payload. A new function is defined to emit optimized property access expressions. This function emits a special JavaScriptCore intrinsic called `getByIdOffset`, which is discussed in Section 3.7.4. Intrinsics emitted are prepended with `$__` to disambiguate them from regular function calls in the JavaScript engine.

The `getByIdOffset` intrinsic accepts three arguments: the object on which the property is being accessed, the property name for debugging purposes, and the offset of the property computed earlier by the type checker. It retrieves the property directly from the inline slot of the object, bypassing all shape checks. Unlike the inline caches employed in JavaScriptCore, which are updated when the structure or shape of an object changes (that is, becoming increasingly polymorphic), shape changes or modifications (resulting from adding extra properties to an object far from its allocation) do not affect the intrinsic.

```
function emitPropertyAccessExpression(node: PropertyAccessExpression) {
    // if access optimization metadata is set, emit optimized property access
    if (node has accessMetaData) {
        return emitOptimizedPropertyAccessExpression(node);
    }
    // emit regular property access expression
    emitExpression(node.expression);
}
```

Figure 3.31: Pseudocode for emitting property access expressions.

In Figure 3.32, the expression `obj.x` from Figure 3.1 is transformed into an intrinsic call. The property `x` corresponds to index 0 in the underlying object layout in the engine, enabling direct retrieval of the property value.

```
let j = obj.x;
```

→

```
let j = $__getByIdOffset(obj, "x", 0);
```

Figure 3.32: Transformation of a property access expression to an intrinsic function call.

`putByIdDirect`

The `putByIdDirect` instruction is a preexisting intrinsic function in JavaScriptCore, and may be used to directly assign a value to an object's property. It has the following format:

```
put_by_id_direct <base> <property> <value>
```

`base` is a virtual register that represents the target object for the property access value, `property` is the name of the property being accessed, and `value` is the right-hand side of the assignment. In the emitter, the function presented in Figure 3.29 utilizes a utility function to emit the layout metadata in class constructors. Particularly, the `putByIdDirect` intrinsic for specifying the layout in such constructors.

3.6.5 `tsconfig.json` Configuration

The modifications to the TypeScript compiler are managed through a new configuration option introduced in `tsconfig.json` named `optimizeWithTypes` (presented in Appendix C). When this option is set to `true`, the optimization changes implemented in `mtsc` are executed whenever applicable.

3.7 JavaScriptCore

This section discusses the implementation details of the extensions to JavaScriptCore. In both the interpreter (LLInt) and JIT (Baseline) tiers, the implemented optimization focuses on enabling efficient object property access. This optimization is achieved by retrieving the accessed property directly from the object using the known offset, which maps to its storage location in the object.

3.7.1 `get_by_id_offset` Bytecode Instruction

Object property access is implemented in LLInt using the `get_by_id` bytecode instruction. To handle optimized property accesses, a new instruction called `get_by_id_offset`, similar to `get_by_id`, has been developed. The `get_by_id_offset` instruction has the format:

```
get_by_id_offset <dst> <base> <property> <propertyOffset>
```

The instruction behaves almost identically to `get_by_id` (Section 2.3.8): `dst` represents the destination register for the property access value, and `property` is the name of the property being accessed. However, unlike `get_by_id`, it also includes a property offset,

which corresponds to the position where the property value can be found in the object's inline slots, as described earlier in Section 2.3.8.

An advantage of `get_by_id_offset` over `get_by_id` is its ability to directly fetch an object's property without traversing the object access chain/link or using a cache. This optimization is achieved by encoding the offset of the property to be accessed directly into the instruction itself. Furthermore, structure validation and checks around property access, present in `get_by_id`, are completely eliminated in `get_by_id_offset`. Changes in an object's shape/structure that require cache updates (when inline caches are employed) and validation (to ensure the correct object structure is fetched) in `get_by_id` do not affect `get_by_id_offset`.

3.7.2 LLInt

At a high level, object property access optimization in LLInt is achieved by utilizing the offset of the accessed property to retrieve the corresponding value from the object. The `get_by_id_offset` opcode fetches the property value using the offset specified in the bytecode instruction and stores it in a destination register, skipping all shape-related checks as they are redundant.

In LLInt, bytecode instructions are defined in a template file that is used in generating implementation files containing utility functions and instruction encoding schemes. The `get_by_id_offset` opcode is implemented in LLInt by initially defining the opcode in the template file and then implementing the actual opcode behaviour in Offlineasm.

Figure 3.33 defines the `get_by_id_offset` opcode. The `args` field defines the arguments accepted by the opcode, while `metadata` contains additional information following the format of `get_by_id`. Additionally, `valueProfile` is an extra metadata within `args` used for value profiling [15], although it remains unused in LLInt.

The `get_by_id_offset` instruction is designed to maintain compatibility with `get_by_id` and can easily be cast to `get_by_id`. This compatibility is the reason why it retains the `metadata` field, despite not being particularly useful to the instruction itself. Furthermore, compatibility with `get_by_id` is necessary to support tiers that are not directly supported by the `get_by_id_offset` instruction.

The implementation of `get_by_id_offset` in Offlineasm is straightforward. First, the object is loaded into a temporary register using a custom macro (`loadConstantOr VariableCellUnchecked`), which skips object structure ID tests. Then, the property offset is loaded into another temporary register and used to compute the actual offset in the storage slots. Subsequently, the corresponding property value is loaded into a temporary register using the computed offset, which is then returned. Figure 3.34 contains a snippet of the core implementation in Offlineasm.

```
op :get_by_id_offset,
    args: {
        dst: VirtualRegister,
        base: VirtualRegister,
        property: unsigned,
        propertyOffset: unsigned,
        valueProfile: unsigned, # not used in llint
    },
    metadata: {
        structureID: StructureID,
        offset: unsigned,
    }
```

Figure 3.33: Definition of `get_by_id_offset` opcode format.

```
llintOpWithMetadata(op_get_by_id_offset, OpGetByIdOffset,
  macro (size, get, dispatch, metadata, return)
    # metadata(OpcodeID opcodeID, unsigned metadataID)
    metadata(t2, t0)
    # m_metadata->get<Metadata>()[metadataID]
    get(m_base, t0)
    # load object into t3 using size, and index t0 -- skip checks
    loadConstantOrVariableCellUnchecked(size, t0, t3)
    loadi OpGetByIdOffset::Metadata::m_offset[t2], t1
    get(m_propertyOffset, t5)
    # add the appropriate/direct offset to t1
    addp t5, t1, t1
    # propertyOffsetAsInt=t1, objectAndStorage=t3, value/propdst=t0
    loadPropertyAtVariableOffset(t1, t3, t0)
    valueProfile(size, OpGetByIdOffset, m_valueProfile, t0, t2)
    return(t0)
end)
```

Figure 3.34: Implementation of `get_by_id_offset` in LLInt.

3.7.3 Baseline

In the Baseline, property access optimization is accomplished by emitting native code that utilizes the offset of the accessed property to retrieve the corresponding value from the object during the access operation. Similar to the LLInt implementation, the `get_by_id_offset` opcode fetches the property value using the offset specified in the bytecode instruction and stores it in a destination register.

JavaScriptCore contains a template file used by the Baseline JIT for defining opcodes, similar to the template file used by LLInt. Opcodes are defined using C++ macros. These macros are employed for native code emission dispatch within a switch block, specifically within a function handling the compilation pass. They serve as wrappers for generating C++ case statements that map an opcode name to a call to the opcode's fast and slow path code generation functions. `get_by_id_offset` is defined using both macros.

Template JIT

The `get_by_id_offset` instruction is implemented in the Baseline JIT as both a bytecode-to-native code transformation and an intrinsic function.

```
1  void JIT::emit_op_get_by_id_offset(const JSInstruction* currentInstruction){
2      auto bytecode = currentInstruction->as<OpGetByIdOffset>();
3      VirtualRegister resultVReg = bytecode.m_dst;
4      VirtualRegister baseVReg = bytecode.m_base;
5      const Identifier* ident = &(m_unlinkedCodeBlock->identifier(bytecode.m_property));
6
7      const auto propOffset = bytecode.m_propertyOffset;
8      using BaselineJITRegisters::GetById::baseJSR;
9      using BaselineJITRegisters::GetById::resultJSR;
10     using BaselineJITRegisters::GetById::stubInfoGPR;
11
12     emitGetVirtualRegister(baseVReg, baseJSR);
13     auto [ stubInfo, stubInfoIndex ] = addUnlinkedStructureStubInfo();
14     loadConstant(stubInfoIndex, stubInfoGPR);
15
16     JITGetByIdGenerator gen(...);
17
18     gen.generateBaselineDataICFastPath(*this, propOffset);
19
20     setFastPathResumePoint();
21     emitValueProfilingSite(bytecode, resultJSR);
22     emitPutVirtualRegister(resultVReg, resultJSR);
23 }
```

Figure 3.35: Pseudocode for `emit_op_get_by_id_offset` in Baseline.

The bytecode-to-native code transformation involves generating code using the fast path/slow path pattern. Optimistic inline operations are implemented in the fast path, while other operations are delegated to the slow path. Therefore, the transformation is handled by two separate JIT methods: `emit_op_get_by_id_offset` for the fast path and `emitSlow_op_get_by_id_offset` for the slow path.

The fast path implementation closely resembles that of LLInt, as illustrated in Figure 3.35. The bytecode arguments for `get_by_id_offset` are fetched and used to generate template native code using a utility function of the code generator. However, in the Baseline implementation, the property offset is directly loaded into a scratch register.

Subsequently, the value corresponding to the offset in the scratch register is loaded into a general-purpose register, ensuring that property access is completely inline. This implementation not only eliminates structure checks and associated jumps but also guarantees that no slow path exit occurs. In other words, execution of `get_by_id_offset` in the Baseline JIT never jumps to the slow path implementation.

Why, then, is there a slow path implementation? The slow path implementation is required by JavaScriptCore for all opcode implementations. However, for the `get_by_id_offset` instruction, the slow path is merely a formality and is never executed. Since `get_by_id_offset` is compatible with `get_by_id`, the slow path logic of `get_by_id` is reused.

```
1  void JIT::emitSlow_op_get_by_id_offset(
2      const JSInstruction*,
3      Vector<SlowCaseEntry>::iterator& iter)
4  {
5      ASSERT(BytecodeIndex(m_bytecodeIndex.offset()) == m_bytecodeIndex);
6      JITGetByIdGenerator& gen = m_getByIds[m_getByIdIndex++];
7      linkAllSlowCases(iter);
8      gen.reportBaselineDataICSlowPathBegin(label());
9      emitNakedNearCall(
10         InlineCacheCompiler::generateSlowPathCode(vm(), gen.accessType())
11             .retaggedCode<NoPtrTag>()
12     );
13 }
```

Figure 3.36: Slow path implementation of `op_get_by_id_offset` in Baseline.

3.7.4 Intrinsic Instructions

JavaScriptCore allows functions that directly map to an operational implementation in the engine to be used within certain contexts, known as *built-in mode*. These functions are referred to as intrinsics. In built-in mode, the JavaScript parser embedded in JavaScriptCore permits the usage of specific code syntax that would normally result in a syntax error. This syntax allows identifiers to begin with an `@` symbol and enables direct access to intrinsic functions in JavaScriptCore. For example, the property `foo` of an object `obj` may be accessed directly using `@getByIdDirect(obj, "foo")` instead of the dot access operation `obj.foo`.

Intrinsics are implemented during the code generation pass of the bytecode compiler. JavaScriptCore parses JavaScript source into an AST based on the *parse mode*, which

may be either `Builtin` or `NotBuiltin` mode. When parsing in `Builtin` mode, function calls resembling intrinsics are recognized and validated. Validation involves checking if the intrinsic name is a well-known symbol or a built-in name; otherwise, a syntax error is generated. Well-known symbols typically begin with double `@`, for example, `@@iterator`, `@@isConcatSpreadable`, etc. Built-in names, on the other hand, begin with a single `@`, such as `@getByIdDirect` mentioned earlier. `Builtin` mode in JavaScriptCore is used for parsing and compiling built-in JavaScript modules, which extensively utilize well-known symbols and built-in names (see Appendix D).

AST nodes are transformed into bytecode by the bytecode compiler. During this transformation, an intrinsic function call is directly translated into bytecode, utilizing the arguments provided to the function in the underlying implementation. The `get_by_id_offset` bytecode can be accessed in `Builtin` mode using the syntax `@getByIdOffset(obj, "prop", offset)`. Here, `obj` corresponds to the JavaScript object on which the property access is being performed, `"prop"` is a `string` denoting the name of the property, and `offset` is an unsigned integer representing the property offset. While the property name is included in the intrinsic function call, it is only necessary for the slow path implementation in the Baseline JIT and for debugging purposes, such as in a bytecode dump of a program.

Allowing Intrinsic Access

While intrinsics are extremely useful, they are inaccessible and trigger a syntax error in user code, as JavaScript does not support function calls beginning with an `@` symbol. To address this syntax limitation, the JavaScript parser in JavaScriptCore is modified to recognize function calls beginning with a `$__` as a built-in name or intrinsic. Additionally, the parser is adjusted to parse code only in `Builtin` mode, enabling direct access to intrinsics.

The drawback of these alterations is that functions can no longer be declared with names starting with `$__` in user code, as such declarations could potentially lead to crashes in the engine. Additionally, parsing code in Builtin mode raises security considerations and is not suitable for production. Nonetheless, for the objectives and applications outlined in this thesis, I believe these modifications are entirely reasonable. The `get_by_id_offset` intrinsic may be accessed as follows:

```
$__getByIdOffset(object, "property", offset)
```

This approach prevents syntax error squiggles and highlighting in code editors such as VS Code [40]. Furthermore, intrinsic function calls tend to stand out from ordinary function calls. Since Bun leverages JavaScriptCore, the modifications made to JavaScriptCore become directly accessible in Bun. However, the modified WebKit project within Bun must be rebuilt for changes to take effect.

Implementation

The intrinsic for `get_by_id_offset` is implemented by utilizing the arguments of the function-call AST node corresponding to an intrinsic function call. This involves generating the `get_by_id_offset` bytecode using a bytecode generator, with operands corresponding to the argument nodes of the function call expression. Figure 3.37 presents the implementation.

```
RegisterID* BytecodeIntrinsicNode::emit_intrinsic_getByIdOffset(
    BytecodeGenerator& generator,
    RegisterID* dst)
{
    ArgumentListNode* node = m_args->m_listNode;
    RefPtr<RegisterID> base = generator.emitNode(node);
    node = node->m_next;
    ASSERT(node->m_expr->isString());
    const Identifier& ident = static_cast<StringNode*>(node->m_expr)->value();
    ASSERT(node->m_next);
    node = node->m_next;
    ASSERT(node->m_expr->isNumber());
    unsigned offset = (unsigned)(static_cast<NumberNode*>(node->m_expr)->value());
    return generator.emitGetByIdOffset(
        generator.finalDestination(dst), base.get(), ident, offset
    );
}
```

Figure 3.37: Implementation of `emit_intrinsic_getByIdOffset` by walking the function call AST node.

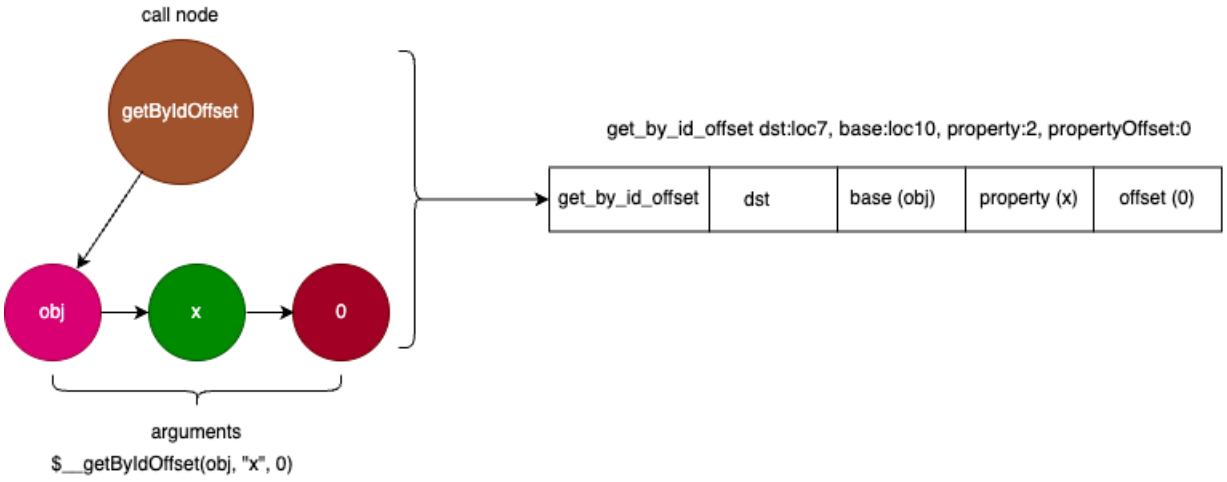

Figure 3.38: Intrinsic function call to bytecode.

3.8 Bringing it All Together

To implement the concepts of this thesis, I have enhanced the TypeScript compiler to optimize property access operations in the resulting JavaScript code. This optimization strategy capitalizes on nominal type information from TypeScript code, while integrating optimization intrinsics provided by JavaScriptCore. Furthermore, modifications are made to JavaScriptCore to support optimized property accesses at both the LLInt and Baseline tiers. Additionally, an intrinsic function is provided, which can be directly utilized for optimizing these operations. The following section presents an example tying together the concepts discussed.

3.8.1 A Complete Example

Figure 3.39 contains an extract of the n-body system simulation ported to TypeScript from the JetStream 2.1 benchmark suite [19], modified for use as an example. In this example, the type `Body` satisfies nominal type usage throughout the program.

When this code is compiled using `mtsc` with optimizations enabled in the `tsconfig.json` configuration file, the result is the JavaScript code presented in Figure 3.40. In Figure 3.40, `mtsc` has optimized property access expressions to calls to the `get_by_id_offset` intrinsic. Additionally, layout specification using `put_by_id` is omitted because the constructor of the class `Body` has field initializations matching the layout metadata computed during nominality checks in `mtsc`.

In each call to `get_by_id_offset`, the object and the offset of the property being accessed is utilized. For example, the property `vx` has an offset of 3, `vy` has an offset of 4, and so on. The property `mass`, being the last property in the layout, has an offset of 6. In this example, `mtsc` creates code that signals optimizations to be performed by JavaScriptCore.

```
const PI = 3.141592653589793;
const SOLAR_MASS = 4 * PI * PI;
const DAYS_PER_YEAR = 365.24;

class Body {
    constructor(
        public x: number, public y: number,
        public z: number, public vx: number, public vy: number,
        public vz: number,
        public readonly mass: number
    ) {
    }

    offsetMomentum(px: number, py: number, pz: number) {
        this.vx = -px / SOLAR_MASS;
        this.vy = -py / SOLAR_MASS;
        this.vz = -pz / SOLAR_MASS;
        return this;
    }
}

const saturn = new Body(
        8.34336671824457987e+00,
        4.12479856412430479e+00,
        -4.03523417114321381e-01,
        -2.76742510726862411e-03 * DAYS_PER_YEAR,
        4.99852801234917238e-03 * DAYS_PER_YEAR,
        2.30417297573763929e-05 * DAYS_PER_YEAR,
        2.85885980666130812e-04 * SOLAR_MASS);
let px = 0.0;
let py = 0.0;
let pz = 0.0;
for (let i = 0; i < 10; i++) {
    px += saturn.vx * saturn.mass;
    py += saturn.vy * saturn.mass;
    pz += saturn.vz * saturn.mass;
}
saturn.offsetMomentum(px, py, pz);
console.log(saturn);
```

Figure 3.39: Extract from the n-body system implementation used in benchmarks.

```
const PI = 3.141592653589793;
const SOLAR_MASS = 4 * PI * PI;
const DAYS_PER_YEAR = 365.24;
class Body {
    constructor(x, y, z, vx, vy, vz, mass) {
    // consistent field initializations, no layout metadata emitted
        this.x = x;
        this.y = y;
        this.z = z;
        this.vx = vx;
        this.vy = vy;
        this.vz = vz;
        this.mass = mass;
    }
    offsetMomentum(px, py, pz) {
        this.vx = -px / SOLAR_MASS;
        this.vy = -py / SOLAR_MASS;
        this.vz = -pz / SOLAR_MASS;
        return this;
    }
}
const saturn = new Body(
    8.34336671824457987e+00,
    4.12479856412430479e+00,
    -4.03523417114321381e-01,
    -2.76742510726862411e-03 * DAYS_PER_YEAR,
    4.99852801234917238e-03 * DAYS_PER_YEAR,
    2.30417297573763929e-05 * DAYS_PER_YEAR,
    2.85885980666130812e-04 * SOLAR_MASS
);
let px = 0.0;
let py = 0.0;
let pz = 0.0;
for (let i = 0; i < 10; i++) {
    // access expressions are transformed into intrinsic calls
    px += $__getByIdOffset(saturn, "vx", 3) * $__getByIdOffset(saturn, "mass", 6);
    py += $__getByIdOffset(saturn, "vy", 4) * $__getByIdOffset(saturn, "mass", 6);
    pz += $__getByIdOffset(saturn, "vz", 5) * $__getByIdOffset(saturn, "mass", 6);
}
saturn.offsetMomentum(px, py, pz);
console.log(saturn);
```

Figure 3.40: Optimized JavaScript for the n-body simulation.

Chapter 4

Evaluation

This chapter evaluates the performance optimizations implemented in this thesis on a series of benchmark programs. It discusses the benchmark programs utilized, as well as the evaluation methodology. It presents the results obtained from the benchmark process, and discusses in detail the implications of said results.

4.1 Benchmarks

The benchmark programs are adapted from the JetStream 2.1 [20] JavaScript and WebAssembly benchmark suite, which is designed for assessing advanced web application performance [19]. Developed by the WebKit team, JetStream 2.1 is a browser benchmark that can be executed on most web browsers. It has been used to evaluate the performance of Chrome [30] (V8 JavaScript engine) and Safari [36] (WebKit) [18].

To evaluate the performance optimizations presented in this thesis, I have selected six benchmark programs from JetStream 2.1, as listed in Table 4.1. As JetStream 2.1 primarily comprises programs written in JavaScript, I have carefully ported each benchmark program to TypeScript. The ports use TypeScript-specific syntax and semantics, such as namespaces and typed classes, as needed, while also prioritizing nominal type usage wherever possible to enable better chances for optimizations. The ported programs are compiled using `mtsc` compiler discussed in Chapter 3. The columns $\texttt{O}_{\texttt{loc}}$ and $\texttt{P}_{\texttt{loc}}$ in Table 4.1 correspond to the number of lines of code in the original JavaScript programs from the test suite and the ported TypeScript programs, respectively.

Most of the ported programs have fewer lines of code compared to their original versions. This is because the original JavaScript programs did not utilize modern features such as classes to concisely model object behaviour. Instead, they frequently relied on constructor functions extended by direct prototype manipulation, resulting in a significant amount of boilerplate code. Additionally, the choice of formatting in the original sources also affects the number of lines of code. For instance, in the program `hash-map`, when defining blocks,

Benchmark B_{jet}	Description	O_{loc}	P_{loc}
`hash-map`	A JavaScript implementation of Apache Harmony's `java.util.HashMap`, designed to perform hash table insertions, queries, and iteration over the associated `entrySet`. This program serves to test object-oriented JavaScript idioms and object construction [20].	521	377
`n-body`	A classic solar system simulation benchmark from The Great Computer Language Shootout [39]. It models the dynamics of the Sun and the four gas giants for a specified number of iterations, evaluating both mathematical computation and object access performance.	155	156
`navier-stokes`	A fluid simulation focusing on floating-point array performance [20].	381	354
`raytrace`	An implementation of a ray tracer written in JavaScript which tests object construction performance and floating point math.	739	633
`richards`	An implementation of an operating system's task dispatcher ported to JavaScript. It tests object property access performance.	479	337
`splay`	The original JavaScript implementation of splay tests the manipulation of splay trees represented using plain JavaScript objects [20]. However, in the port of splay to TypeScript utilized in this benchmark, classes are employed judiciously. This benchmark particularly emphasizes the performance of the garbage collector.	373	264

Table 4.1: Benchmark programs and their descriptions.

opening braces are placed on a new line; however, this is not the case in my port, as opening braces always start at the current line where they are being defined.

For each JavaScript program, the corresponding TypeScript port is compiled with `mtsc` in two modes: normal and patched. In normal mode, `mtsc` produces JavaScript as if it were a regular TypeScript compiler, without invoking any of the optimizations discussed in Chapter 3. The resulting JavaScript code is referred to as `nts`, representing JavaScript from *normal TypeScript*. In patched mode, the same TypeScript program is compiled to JavaScript using `mtsc`, with optimizations enabled and invoked whenever possible. The resulting JavaScript code is referred to as `pts`, representing JavaScript from *patched TypeScript*. The original JavaScript program from JetStream 2.1 is identified as `ojs`.

Compilation in these two distinctive modes is set up as follows: For each JavaScript program, two separate TypeScript projects are created, each containing a root `tsconfig.json` file. This means that each ported JavaScript program is embedded in a directory and

adapted into two separate TypeScript subdirectories corresponding to `nts` and `pts` within the same program directory.

Both `nts` and `pts` contain the exact same TypeScript code; however, their `tsconfig.json` files differ by a single line of configuration. In `pts`, the configuration `optimizeWithTypes` is set to `true`, indicating that the TypeScript compiler should apply optimizations discussed in Chapter 3, whenever possible. However, `nts` does not include such configuration, implying that the TypeScript code should be compiled like a regular, unmodified TypeScript compiler would.

Subsequently, `mtsc` generates JavaScript code for each program, applying optimizations where possible (`pts`) and producing JavaScript code without optimizations (`nts`). The details of the benchmark setup are presented in Table 4.2.

	Setup	*tsc* Comp. Mode	Optimization
Program	`nts`	*normal*	disabled
	`pts`	*patched*	enabled
	`ojs`	*original*	unavailable

Table 4.2: Benchmark setup.

Although `nts` and `pts` are carefully adapted from `ojs`, each benchmark program includes a checksum. These checksums are inherent to the original JavaScript programs from JetStream and are designed to validate the computed results of each program at runtime after a successful run. The checksum also helps to ensure that the ported programs maintain semantic equivalence with the original JavaScript programs and exhibit the same observable behaviour. While the benchmarks are executed for `ojs`, `nts`, and `pts`, the evaluation in this chapter primarily focuses on the performance of JavaScript in `nts` and `pts`. This emphasis is because they are much more syntactically and semantically equivalent compared to the original JavaScript programs [4].

It is worth noting that the tests selected from JetStream 2.1 are complete working programs that implement specific features. Due to the diverse nature of these programs, isolating the effect of optimizations developed in this thesis is challenging. Therefore, an additional set of microbenchmark programs has been developed in TypeScript to properly evaluate the effectiveness of discussed optimizations in isolation. These programs do not implement specific features, but perform computations utilizing access expressions that are optimizable.

For the purpose of distinguishing benchmarks, the benchmark programs from JetStream 2.1 are referred to as B_{jet}, while the microbenchmark programs are referred to as B_{μ}.

4.1.1 Microbenchmarks

Table 4.3 presents the list of programs used in the microbenchmarks, along with their descriptions and number of lines of code ($\texttt{O}_{\texttt{loc}}$). These programs have been developed from scratch in TypeScript and compiled with `mtsc` in two modes, similar to the JetStream benchmark programs: `nts` (optimizations disabled) and `pts` (optimizations enabled).

To evaluate execution performance in the microbenchmarks, the performance of `nts` is compared with `pts`. Similar to B_{jet}, each microbenchmark program incorporates a checksum that validates the computation performed and ensures that semantics are preserved across `nts` and `pts`.

The microbenchmark programs execute computations over a long-running loop, utilizing property access expressions within each computation. These programs perform computations at specific intervals within each iteration. The programs listed in Table 4.3 are variations of each other.

Benchmark B_μ	Description	$\texttt{O}_{\texttt{loc}}$
`micro-k`	Performs computations over a number of iterations at even and multiples of three intervals.	97
`micro-x`	Performs computation over a number of iterations at multiples of two, three, and four intervals.	100
`micro-y`	Similar to `micro-x`, this program performs computation over a number of iterations at multiples of two and three intervals.	104
`micro-z`	Combines computation in `micro-x` and `micro-y` with a slight variation on intervals and additional object method calls during each computation.	96

Table 4.3: Microbenchmark programs and their descriptions.

4.2 Methodology

The benchmarks were conducted on a 64-bit Ubuntu 22.04.4 LTS machine featuring an AMD EPYC 9754 processor with 256 cores, 768 GB of RAM, a clock speed of 2.25 GHz, and 4 TB of storage. To assess the impact of optimizations applied to both the LLInt and Baseline tiers of JavaScriptCore, the benchmark runs for both B_{jet} and B_μ are divided into three categories of JavaScriptCore configurations across all test suites:

- LLInt only (T_L)
- Baseline only (T_B)
- LLInt and Baseline only (T_{LB})

These configuration modes are used to isolate the impact of the optimizations in the LLInt and Baseline tiers of the JavaScript engine. For the T_L category, JavaScriptCore is modified to run only on LLInt and never JIT compile to any of the JIT tiers. Running code solely in the LLInt tier of JavaScriptCore is known as executing in no-JIT or "mini mode" [15]. In the T_B category, JavaScriptCore is modified to always JIT-compile and execute code in the Baseline, without utilizing LLInt or higher JIT tiers. In the T_{LB} category, JavaScriptCore is modified to run both in LLInt and JIT-compile to the Baseline when applicable; however, higher JIT tiers are not utilized. In each test category, other configuration options in JavaScriptCore are kept at their default values.

While JavaScriptCore serves as the engine modified in this thesis, the Bun JavaScript runtime [29] is employed to evaluate these modifications by executing the programs in the test suite. This decision is driven by the fact that the programs heavily rely on runtime APIs, most of which are not provided by the JavaScriptCore engine but are instead delegated to a runtime environment. The Bun runtime utilizes JavaScriptCore under the hood. Therefore, modifications made to JavaScriptCore are applied to Bun, which is then recompiled in release mode for test execution. For the evaluation process, a recent version of Bun, 1.0.14, is utilized. Additionally, the TypeScript compiler extended is also recent (4.9.5).

The benchmarks B_{jet} and B_{μ} are conducted separately. In both benchmarks, each program listed in Tables 4.1 and 4.3 is executed for 10 iterations. For each iteration, 10 warm-up runs are conducted, followed by the actual benchmark runs consisting of 50 executions.

The main focus of B_{jet} is to compare tests between `nts` and `pts`, prioritizing them over `ojs` due to their similarity. Hence, the benchmark B_{jet} is conducted in two phases: firstly with `nts` and `pts`, and secondly with `nts`, `pts`, and `ojs`. Both benchmark processes measure the execution time in seconds of each running program and are executed using the command-line benchmarking tool, hyperfine [80]. Additionally, iterations are conducted separately for each JavaScriptCore configuration (T_L, T_B, and T_{LB}), utilizing a Python script [34] that automates hyperfine.

4.3 Results

This section presents results from benchmarks B_{jet} and B_{μ}.

4.3.1 B_{jet}

The benchmark results for `nts` and `pts` are shown in Table 4.4. These results represent the average runtime across all runs of each iteration for each test program. Additionally, the results are obtained for each configuration mode of JavaScriptCore. The column labeled `rel` indicates the relative time of each test compared to the fastest runtime.

B_{jet}		T_L					T_B				T_{LB}		
Program	kind	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time
n-body	nts	0.231	0.216	0.237	1.000	0.089	0.083	0.092	1.044	0.080	0.075	0.083	1.047
	pts	0.231	0.213	0.237	1.000	0.085	0.080	0.088	1.000	0.077	0.071	0.079	1.000
richards	nts	0.307	0.293	0.313	1.035	0.111	0.105	0.115	1.128	0.119	0.108	0.123	1.088
	pts	0.297	0.286	0.303	1.000	0.098	0.093	0.102	1.000	0.109	0.100	0.114	1.000
splay	nts	0.202	0.194	0.206	1.007	0.125	0.117	0.130	1.017	0.126	0.118	0.132	1.017
	pts	0.201	0.193	0.204	1.000	0.123	0.116	0.125	1.000	0.124	0.116	0.129	1.000
navier-stokes	nts	1.310	1.240	1.343	1.000	0.733	0.719	0.746	1.000	0.596	0.587	0.600	1.000
	pts	1.311	1.249	1.339	1.000	0.733	0.719	0.744	1.000	0.597	0.588	0.604	1.000
hash-map	nts	0.231	0.221	0.237	1.024	0.733	0.117	0.108	1.034	0.129	0.117	0.135	1.018
	pts	0.226	0.216	0.232	1.000	0.113	0.104	0.117	1.000	0.127	0.114	0.132	1.000
raytrace	nts	0.363	0.352	0.368	1.024	0.165	0.157	0.170	1.050	0.166	0.156	0.172	1.031
	pts	0.355	0.344	0.361	1.000	0.158	0.150	0.161	1.000	0.161	0.153	0.165	1.000

Table 4.4: Performance comparison of `nts` and `pts` in B_{jet} across three configurations of JavaScriptCore. Lower is better.

The results obtained from performing the same benchmarks using `nts`, `pts`, and `ojs` for T_L, T_B, and T_{LB} configurations are also shown in Table 4.5. Although the performance of `nts` and `pts` compared to `ojs` is reported, the results of `ojs` are not central to the succeeding discussion.

B_{jet}		T_L					T_B				T_{LB}		
Program	kind	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time
n-body	nts	0.231	0.218	0.236	1.013	0.089	0.083	0.092	1.044	0.080	0.075	0.083	1.046
	pts	0.230	0.215	0.236	1.011	0.085	0.080	0.088	1.000	0.076	0.071	0.079	1.000
	ojs	0.228	0.212	0.235	1.000	0.090	0.084	0.094	1.057	0.080	0.074	0.083	1.045
richards	nts	0.307	0.294	0.312	1.087	0.111	0.105	0.114	1.124	0.119	0.109	0.125	1.088
	pts	0.296	0.286	0.301	1.049	0.099	0.092	0.102	1.000	0.109	0.099	0.114	1.000
	ojs	0.282	0.268	0.287	1.000	0.111	0.104	0.114	1.121	0.127	0.111	0.136	1.165
splay	nts	0.202	0.193	0.206	1.009	0.126	0.117	0.130	1.021	0.126	0.118	0.130	1.016
	pts	0.200	0.192	0.204	1.000	0.123	0.116	0.127	1.000	0.124	0.117	0.127	1.000
	ojs	0.276	0.268	0.281	1.376	0.134	0.126	0.138	1.091	0.136	0.128	0.139	1.094
navier-stokes	nts	1.306	1.248	1.354	1.000	0.733	0.719	0.743	1.002	0.596	0.587	0.601	1.000
	pts	1.308	1.236	1.338	1.001	0.733	0.718	0.742	1.001	0.597	0.588	0.603	1.001
	ojs	1.307	1.229	1.350	1.001	0.732	0.721	0.743	1.000	0.596	0.586	0.603	1.000
hash-map	nts	0.231	0.220	0.237	1.022	0.117	0.108	0.122	1.035	0.129	0.117	0.135	1.014
	pts	0.226	0.217	0.231	1.000	0.113	0.105	0.117	1.000	0.127	0.112	0.134	1.001
	ojs	0.236	0.225	0.240	1.042	0.114	0.106	0.118	1.012	0.127	0.115	0.132	1.000
raytrace	nts	0.363	0.350	0.369	1.024	0.165	0.158	0.171	1.051	0.166	0.157	0.172	1.031
	pts	0.355	0.345	0.361	1.000	0.157	0.149	0.162	1.000	0.161	0.153	0.164	1.000
	ojs	0.549	0.538	0.555	1.548	0.263	0.254	0.268	1.672	0.265	0.254	0.274	1.649

Table 4.5: Performance comparison of `nts`, `pts` and `ojs` in B_{jet} across three configurations of JavaScriptCore. Lower is better.

Figure 4.1 compares the performance of `nts` and `pts` using the data obtained from Table 4.4. Specifically, the results of each test in `nts` and `pts` are compared for each configuration mode. Furthermore, the performance of `nts`, `pts`, and `ojs` is compared across tiers for each test program, as shown in Figure 4.2.

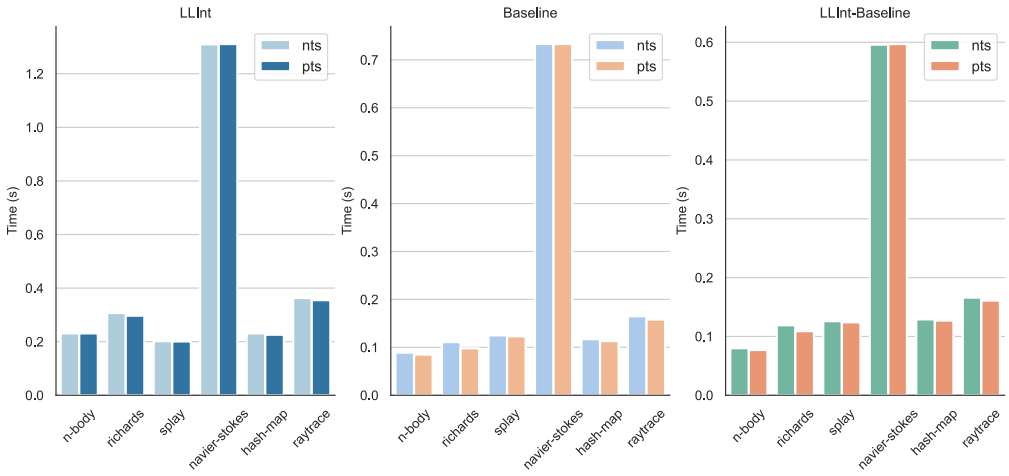

Figure 4.1: Performance comparison of `nts` and `pts` in B_{jet} across T_L, T_B and T_{LB} configurations. Lower is better.

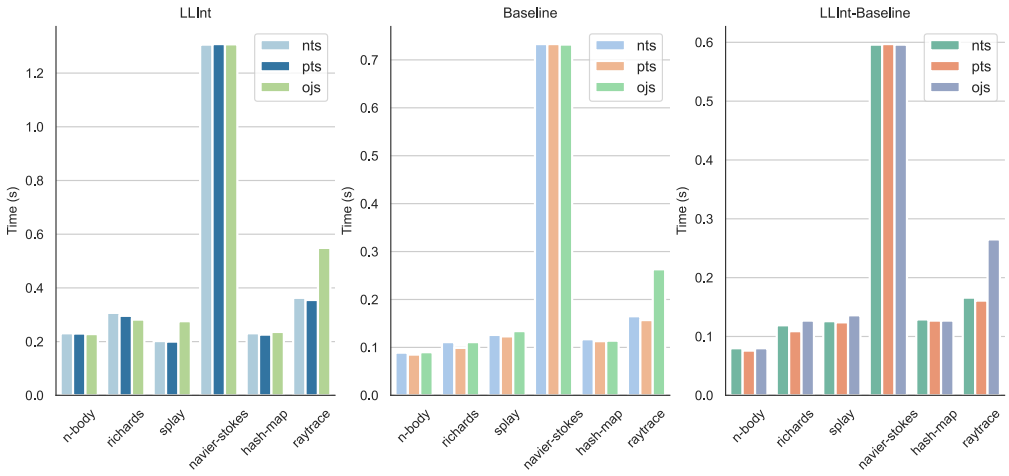

Figure 4.2: Performance comparison of `nts`, `pts` and `ojs` in B_{jet} across T_L, T_B and T_{LB} configurations. Lower is better.

The data from Table 4.4 is also utilized in the computation of the geometric mean [72] of all tests conducted within each configuration, as shown in Table 4.6. This method is akin to the computation of browser benchmark performance scores across multiple workloads conducted by Speedometer 3 [37, 38]. Finally, the percentage speedup of `pts` over `nts` for each test and in each benchmark mode for all iterations and across all configurations in B_{jet} are presented in Table 4.7.

When comparing the results of `pts` and `nts` across each configuration (T_L, T_B, and T_{LB}) using the geometric mean score, it is observed that T_{LB} exhibits the best performance

B_{jet}	nts	pts
T_L	0.341	0.336
T_B	0.161	0.154
T_{LB}	0.158	0.153

Table 4.6: Geometric mean of nts and pts in B_{jet}. Lower is better.

compared to the other two modes, with pts outperforming nts. However, the configuration T_{LB} is only slightly more performant than T_B. Additionally, pts outperforms nts in every configuration and exhibits significantly better performance than nts in T_B. Table 4.7 shows richards demonstrates the best performance speedups across all configurations. In T_L, a speedup of 3.5% is observed, 12.8% in T_B, and 8.8% in T_{LB}. The increase in speedup in the Baseline tier might be attributed to the absence of warm-up time in T_B, as execution directly starts natively. This is further reflected in raytrace by the decline in speedup from 12.8% to 8.7%. In contrast, n-body improves across each mode, with no speedup in T_L, 4.4% speedup in T_B, and 4.7% speedup in T_{LB}.

B_{jet}	T_L		T_B		T_{LB}	
Program	time	% speedup	time	% speedup	time	% speedup
n-body	0.231	0.0	0.089	4.4	0.080	4.7
richards	0.307	3.5	0.111	12.8	0.119	8.8
splay	0.202	0.7	0.125	1.7	0.126	1.7
navier-stokes	1.310	0.0	0.733	0.0	0.596	0.0
hash-map	0.231	2.4	0.117	3.4	0.129	1.8
raytrace	0.363	2.4	0.165	5.0	0.166	3.1

Table 4.7: Speedup of pts over nts in B_{jet}. For speedup, higher is better.

4.3.2 B_{μ}

In Table 4.8, the results of the microbenchmarks are presented, detailing the performance comparison between nts and pts across the T_L, T_B, and T_{LB} settings. This performance comparison is further illustrated in Figure 4.3, utilizing the data from the aforementioned table. Additionally, Table 4.9 presents the geometric mean of performance across all tests for each JavaScriptCore configuration mode. Furthermore, Table 4.10 displays the percentage speedup of pts over nts.

Comparing the performance of pts and nts across each configuration (T_L, T_B, and T_{LB}) using the geometric mean score reveals consistent trends. Like B_{jet}, T_{LB} consistently demonstrates better performance compared to the other two modes, with pts consistently outperforming nts. Moreover, pts surpasses nts in every configuration, nearly doubling

B_μ		T_L					T_B				T_{LB}		
Program	kind	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time	time	min-time	max-time	rel-time
micro-y	nts	0.362	0.335	0.380	1.137	0.197	0.187	0.202	1.335	0.154	0.146	0.157	1.601
	pts	0.319	0.300	0.325	1.000	0.147	0.140	0.150	1.000	0.096	0.089	0.099	1.000
micro-x	nts	0.379	0.352	0.393	1.127	0.208	0.197	0.213	1.267	0.167	0.158	0.170	1.650
	pts	0.336	0.325	0.343	1.000	0.164	0.155	0.170	1.000	0.101	0.094	0.104	1.000
micro-k	nts	0.318	0.291	0.348	1.119	0.170	0.161	0.176	1.320	0.133	0.125	0.137	1.642
	pts	0.284	0.268	0.296	1.000	0.129	0.121	0.133	1.000	0.081	0.074	0.083	1.000
micro-z	nts	0.273	0.250	0.278	1.144	0.157	0.146	0.163	1.461	0.117	0.110	0.120	1.685
	pts	0.238	0.227	0.248	1.000	0.107	0.100	0.111	1.000	0.069	0.063	0.073	1.000

Table 4.8: Performance comparison of `nts` and `pts` in B_μ across three configurations of JavaScriptCore. Lower is better.

its speedup with each transition from T_L to T_{LB}. `micro-z` showcases the most significant performance improvements across all configurations, with a speedup of 14.4% in T_L, 46.1% in T_B, and 68.5% in T_{LB}.

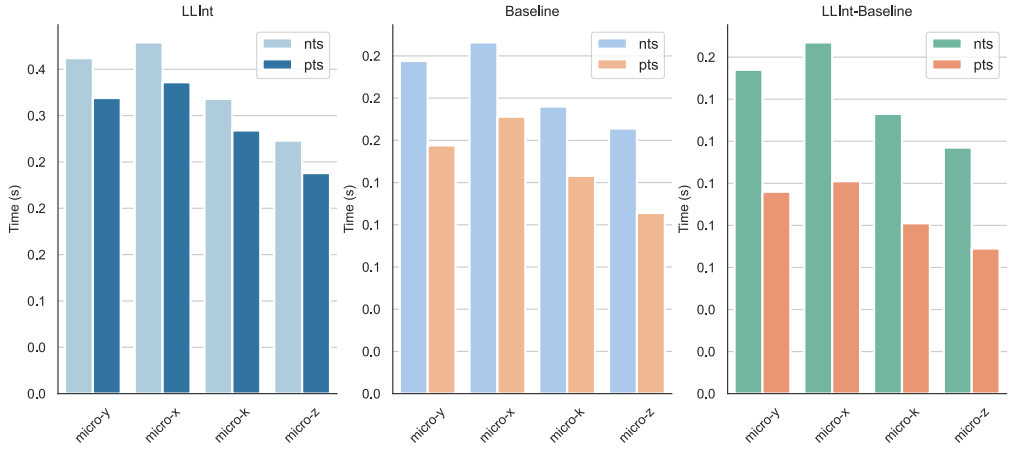

Figure 4.3: Performance comparison of `nts` and `pts` in B_μ across T_L, T_B and T_{LB} configurations. Lower is better.

B_μ	`nts`	`pts`
T_L	0.330	0.292
T_B	0.182	0.135
T_{LB}	0.141	0.086

Table 4.9: Geometric mean of `nts` and `pts` in B_μ. Lower is better.

B_{jet}	T_L		T_B		T_{LB}	
Program	time	% speedup	time	% speedup	time	% speedup
`micro-y`	0.362	13.7	0.197	33.5	0.154	60.1
`micro-x`	0.379	12.7	0.208	26.7	0.167	65.0
`micro-k`	0.318	11.9	0.170	32.0	0.133	64.2
`micro-z`	0.273	14.4	0.157	46.1	0.117	68.5

Table 4.10: Speedup of `pts` over `nts` in B_μ. For speedup, higher is better.

4.3.3 Intrinsics

Tables 4.11 and 4.12 present the test programs along with the number of classes defined and the optimization intrinsics emitted per program (`pts`) in B_{jet} and B_μ respectively. The optimization intrinsics are divided into layout intrinsics and access intrinsics for object layout metadata and property access optimizations, respectively. In cases where there are no layout intrinsics (i.e., 0), it implies field layout instructions were omitted as the class constructor field initializations already conform to the layout metadata computed by `mtsc`, as discussed in Section 3.6.4.

Benchmark B_{jet}	classes	layout intrinsics	access intrinsics	total intrinsics emitted
`hash-map`	9	6	108	114
`n-body`	2	1	41	42
`navier-stokes`	2	0	6	6
`raytrace`	14	14	219	233
`richards`	7	20	87	107
`splay`	2	0	49	49

Table 4.11: Benchmark programs and emitted intrinsics.

Benchmark B_μ	classes	layout intrinsics	access intrinsics	total intrinsics emitted
`micro-k`	2	0	209	209
`micro-x`	2	0	239	239
`micro-y`	2	0	226	226
`micro-z`	2	0	206	206

Table 4.12: Benchmark programs and emitted intrinsics.

In both B_{jet} and B_μ, the number of layout intrinsics emitted does not directly correlate with the amount of performance speedup gained by `pts` over `nts`. This lack of correlation is attributed to the fact that the runtime behaviour of the program is not solely dependent

on property accesses. Additionally, the static number of intrinsics is not reflective of the number of times the intrinsic is executed dynamically, for instance, in a loop.

4.4 Discussion

4.4.1 B_{jet}

The results of the benchmarks reveal that in T_L (LLInt only), `pts` is faster than `nts`. Particularly for tests such as `richards`, `hash-map`, and `raytrace`, a speedup of 3.5%, 2.4%, and 2.4% is attained, respectively. However, a speedup of less than 1% is observed in the other test cases. In fact, there is no speedup in `n-body` and `navier-stokes`. Predictably, the performance in T_L is the slowest in all three configurations, as LLInt executes bytecode only in a virtual machine. Perhaps due to the structure of some of the programs, it is difficult to achieve better speedups by relying entirely on LLInt.

Also, in T_B (Baseline only), a much higher speedup is recorded for tests: `richards` (12.8%), `raytrace` (5.0%), and `n-body` (4.4%). Additionally, `splay` has a higher speedup of 1.7% an improvement compared to its speedup in T_L. However, `navier-stokes` remains the same with no speedup.

`richards` is a program that simulates the task dispatcher of an operating system and tests object property access performance [20]. Given that the optimizations in this thesis target object property access operations, the highest speedup observed in this test is expected. Additionally, the program demonstrates good usage of nominal types, primarily employing nominal classes. Consequently, access optimizations are applied across a wide range of code sections in the generated JavaScript.

`n-body` also performs better in T_B compared to T_L, and the same is true for `raytrace` and `hash-map`. Interestingly, while `n-body` showed no speedup in T_L, it exhibits a dramatic speedup in T_B compared to T_L. Additionally, the speedup of `raytrace` in T_B doubles, while `richards` quadruples. It is reasonable to suspect that some opcode implementations in LLInt are subpar compared to their counterparts in the Baseline. This suspicion arises from the fact that the interpreter is designed with a focus on portability, while performance optimization is primarily delegated to the JIT tiers [9, 15]. For instance, the Baseline implementation of the opcode `get_by_id`, responsible for object property access in JavaScriptCore, is extensively optimized through the use of sophisticated inline caching techniques, unlike LLInt. In fact, the Baseline and higher JIT tiers boast a superior inline caching implementation compared to LLInt [15].

In T_{LB} (LLInt-Baseline only), there is a decline in speedups compared to T_B. Specifically, `richards` exhibits a speedup of 8.8%, `raytrace` 3.1%, and `hash-map` 1.8%. However, `n-body` experiences a speedup of 4.7%, while `navier-stokes` shows no speedup. The decline in performance might be due to the warm-up time required to move execution from LLInt to the Baseline JIT. Code execution would have persisted in LLInt for a while before

being transferred to the Baseline, resulting in more time spent executing in LLInt than in the Baseline.

Additionally, the lack of speedup of `navier-stokes` across all configurations might be due to the program's heavy emphasis on floating-point array performance, which may limit the effectiveness of the optimizations implemented. Floating-point arithmetic and array accesses are prevalent throughout the entire source code. In fact, `pts` in `navier-stokes` has the lowest number of intrinsic calls compared to other test programs. Therefore, access optimizations do not prove effective because more time is spent performing floating-point arithmetic using arrays.

Figure 4.2 compares the benchmark results of `nts`, `pts`, and `ojs`. Although `pts` and `nts` were ported to TypeScript from `ojs`, they are semantically equivalent to `ojs` in terms of JavaScript output, but differ in the language constructs utilized. `nts` and `pts` have been ported using TypeScript constructs where suitable, but TypeScript constructs may not always translate to performant JavaScript. For instance, the use of namespaces in TypeScript translates to immediately invoked function expressions (IIFE) [23], which may not be as performant as using classes or objects for encapsulation. That being said, in T_L, `ojs` outperforms `nts` and `pts` in `n-body` by 1.3% and 1.1%, respectively, and in `richards` by 8.7% and 4.9%, respectively. However, `pts` exhibits speedups over `ojs` in `splay` by 37.6%, `hash-map` by 4.2%, and `raytrace` by 54.8%.

In T_{LB}, the performance difference between `pts` and `ojs` is more pronounced. `pts` exhibits a speedup of 5.7% over `ojs` in `n-body`, 12.1% in `richards`, 9.1% in `splay`, and 67.2% in `raytrace`. The significant performance gap between `pts`/`nts` and `ojs` in `raytrace` might be attributed to differences in implementation constructs. In `ojs`, direct manipulation of a function object's prototype is distributed across several code sites, whereas in `pts` and `nts`, the implementation relies on classes as much as possible and avoids direct prototype manipulation where possible. This observation aligns with the findings of Ahn et al. [43], which highlights object prototype mutation as the primary cause of type unpredictability, impacting performance in V8.

4.4.2 B_μ

Table 4.3 presents the results of the microbenchmarks, including the relative speedup of `pts` over `nts`. In these results, the effects of the applied optimizations are isolated and much more observable in the LLInt and Baseline tiers. In LLInt, access optimizations are shown to be beneficial and provide up to a 14% speedup. This improvement is especially useful for JavaScriptCore when running in mini-mode, that is, LLInt only, as preferred by some users for security and portability reasons [15].

Furthermore, the results in Table 4.10 show that performance improvements and speedups nearly double across each tier, from T_L to T_B, and from T_L to T_{LB}, aligning with established performance metrics of the tiers in JavaScriptCore on the JetStream 2 benchmarks

[15]. In T_L, a performance speedup of 14.4% (`micro-z`) of `pts` over `nts` is recorded. Moreover, in T_B, a speedup of 46.1% (`micro-z`) is observed, and in T_{LB}, a speedup of 68.5% (`micro-z`) is observed.

The results in Figure 4.3 indicate that T_B surpasses T_L in performance. This performance difference stems from various factors. Firstly, code compiled by the Baseline runs natively, eliminating the need for interpretation encountered in LLInt. This alone reduces the overhead associated with interpreter dispatch for bytecode instruction decoding and execution [13]. Interpreter dispatch is particularly costly due to the difficulty the CPUs face in predicting the indirect branches used for selecting the implementation of an opcode [47].

T_{LB} represents the combination of LLInt and Baseline, where LLInt is permitted to transition execution to the Baseline. From Table 4.10, the speedups of T_{LB} over T_L are largely similar to those of T_B over T_L. It is not surprising that the Baseline performs slightly better because T_{LB} initially executes code in LLInt and transitions to the Baseline only after frequent execution, unlike T_B, where JavaScript code is directly compiled to machine code before execution.

Similar to B_{jet}, the geometric mean of the performance of `nts` and `pts` in each configuration indicates that `pts` outperforms `nts` in all configurations, and T_{LB} is the best configuration in terms of performance. However, unlike B_{jet}, T_{LB} appears to be significantly more performant than T_B.

4.4.3 Heuristics

One limitation of the optimization heuristics (Section 3.3) and consequently `mtsc` is that during nominality checks, if a type is identified as non-nominal, its usages are considered unoptimizable. This means that other objects created from the type will not be optimized, even if they are used nominally.

Consider the program in Figure 4.4. The `Point` type does not satisfy nominal type usage because the `Point` variable `p1` is cast to `any` at line 11. However, the `Point` variable `p2` at line 15 is used nominally. `mtsc` fails to optimize the program because nominal type usage for the type `Point` has already been violated by variable `p1`.

In benchmark B_{jet}, when porting the programs to TypeScript, I took advantage of classes to satisfy nominal type usage requirements when possible. This contrasts with the original programs where constructor functions are used with prototype extensions.

My inheritance constraint for nominal types excludes types that satisfy nominal type usage but are part of an inheritance chain. This requirement impacts the `hash-map` and `raytrace` test programs in B_{jet}. In `hash-map`, three classes were created using inheritance, and in `raytrace`, two classes were created using inheritance. However, the original JavaScript test programs never utilized class inheritance; instead, they extended objects by modifying their prototypes.

(a) Code in TypeScript.

```
class Point {
    public x: number;
    public y: number;

    constructor(a: number, b: number) {
        this.x = a;
        this.y = a;
    }
}
let p1 = new Point(12, 13);
let k = p1 as any;
p1.x + p1.y;

// used nominally but not optimized
let p2 = new Point(5, 6);
p2.x * p2.y;
```

→

(b) Generated JavaScript.

```
class Point {

    constructor(a, b) {
        this.x = a;
        this.y = a;
    }
}


let p1 = new Point(12, 13);
let k = p1;
p1.x + p1.y;


let p2 = new Point(5, 6);
p2.x * p2.y;
```

Figure 4.4: Nominal type violation and missed access optimizations.

4.5 Closing Remarks

The results from B_{jet} and B_{μ} exhibit interesting similarities and differences. In both benchmarks, `pts` outperforms `nts` across all configurations. Additionally, according to the geometric mean score, T_{LB} emerges as the optimal configuration. However, notable differences arise between the two benchmarks. In B_{jet}, performance increases from T_L to T_B, but decreases in T_{LB}, unlike in B_{μ} where performance nearly doubles with each configuration from T_L to T_{LB}. It is quite plausible that in B_{μ}, the programs run for longer, leading to the amortization of the effect of execution transitioning from LLInt to the Baseline in T_{LB}, as more time is spent in the Baseline compared to LLInt. Additionally, most of the test programs used in B_{jet} are general implementations of complete working algorithms, with emphasis on specific areas of the JavaScript engine. In contrast, test programs in B_{μ} specifically target access operations.

In both B_{jet} and B_{μ}, the benchmarks are conducted on the LLInt and Baseline tiers, with the DFG and FTL tiers excluded. The DFG and FTL JITs are intricate pieces of software with numerous optimizations implemented over time [9, 15], and they are actively under development [31]. The cost-to-benefit ratio in these tiers is likely to be high. Additional optimization support in these tiers may not yield substantial performance improvements, considering that the JITs already perform numerous optimizations in those areas. Therefore, it might be more beneficial to focus on improving speculation to reduce deoptimization in these tiers, which could lead to overall performance enhancements. Additionally, this work is a research prototype with limited implementation time, making it

impractical to support the higher JIT tiers.

Chapter 5

Related Work

This chapter examines previous academic and industrial endeavours aimed at optimizing the runtime performance of dynamic languages, with a particular emphasis on JavaScript engines and JavaScript programs. It is broadly divided into two sections: The first section provides an overview of performance considerations, including optimization techniques and associated challenges. The second section delves into relevant research and developments pertinent to this thesis.

5.1 JavaScript Performance and Optimization Challenges

Modern JavaScript engines employ various optimization strategies to enhance the performance of JavaScript programs. One prevalent method is the utilization of just-in-time (JIT) compilers. These compilers utilize several optimization techniques, including runtime type inference [41] and type feedback [69], to generate efficient code.

Type-feedback JIT is a speculative technique that utilizes runtime information to generate efficient code, reusable in future executions if types remain unchanged [69, 93]. This approach enables type specialization by instrumenting the code at runtime to collect and store observed types [71]. Type inference, on the other hand, deduces value types that must be correct and specializes code based on these deductions [71]. Unlike type feedback, type inference does not require runtime code instrumentation. In dynamic language engines, type inference is typically performed on a per-function basis for hot functions, which are adaptively detected during execution [71].

Despite advancements in optimization techniques, JavaScript's dynamic nature continues to pose a challenge for achieving optimal performance. This inherent characteristic permits the creation of code that undermines assumptions made by JIT compilers for aggressive optimizations, resulting in performance regressions [87, 93].

An empirical study examining the dynamic behaviour of a corpus of widely-used JavaScript programs found on popular websites, focusing on the utilization of dynamic features and their rationales, reveals that several assumptions made by optimizing JavaScript engines do not directly apply to real-world code [84]. For instance, the assumption of prototype hierarchy invariance suggests that the hierarchy remains unchanged after an object's creation. However, the study indicates that libraries often modify JavaScript's built-in prototypes to introduce behaviours to types (such as `Object` and `Array`), which would typically be immutable in a stricter language. Additionally, the assumption that properties are added during object initialization implies that most changes to an object's fields and methods occur at initialization, making it reasonable to assign an almost complete type to objects upon creation, with only a small number of properties marked as *potential*. However, this was found to be true for only a subset of sites. Furthermore, the assumption that properties are rarely deleted seldom holds true.

One of the most common sources of performance bottlenecks in modern engines is type mutation [93]. At the JIT level, type mutations can lead to deoptimization, fallback to inline caching, weakening of optimizations, and increased garbage collection pressure [93]. Ahn et al. [43] identified type unpredictability as a significant factor contributing to poor performance in JavaScript websites. Type predictability, crucial for performance optimization, is assessed based on the compiler's ability to anticipate object types at access sites (*type-hit-rate*) and the variability of observed object types at those sites (*polymorphism*) [43].

In their study focusing on V8 [5] as a case example, Ahn et al. found that changes in (object and function) prototypes and method bindings are the primary causes of type unpredictability. V8, like JavaScriptCore and Self [55], utilizes *hidden classes* to represent types. Objects created in a similar manner are grouped into the same hidden class, allowing V8 to generate efficient code by recording the offset where a property for a particular object type is located [43]. Although popular JavaScript engine benchmarks often assume that object prototypes and method bindings remain static, Ahn et al. observed a more dynamic behaviour in website code, where prototypes and method bindings frequently change, consistent with the findings of Richards et al. [84].

The immutability of prototypes in hidden classes allows for automatic prototype checks when a hidden class is examined in inline caches [43]. However, a drawback of this approach is that, to maintain prototype immutability, a new hidden class needs to be created for every change in the prototype [43]. This issue is exacerbated by the absence of limits on the number of hidden classes that can be created due to prototype changes. By restructuring the V8 compiler to decouple prototypes and method bindings from an object's type, a 36% reduction in the execution time of `JSBench` [83] and a 49% decrease in the dynamic instruction count were achieved [43].

Identifying and understanding performance issues related to type mutation can often be daunting and laborious. In large codebases, type mutation may be challenging to comprehend and thus difficult to address. Xiao et al. [93] developed `JSweeter`, a tool for

detecting performance issues resulting from type mutations based on type evolution graphs extracted from program executions. These graphs are generated from operational logs containing type update operations for objects and deoptimization information, obtained through instrumentation of a JavaScript engine [93]. The application of refactoring hints generated by `JSweeter` increases JavaScript performance on average by 5.3% and up to 23% [93].

Another empirical study, examining 98 resolved performance issues across 16 JavaScript open-source projects, encompassing both client-side and server-side code, popular libraries, and widely used application frameworks, revealed inefficient API usage as the most prevalent root cause [88]. Common examples include runtime type checks, invocations of function objects, and object property checks [88]. TypeScript, being a superset of JavaScript, can be employed to mitigate API misuse issues. Furthermore, many of these issues can be addressed through optimizations that involve only a few lines of code, without significantly affecting the complexity of the source code [88]. For instance, replacing a `for-in` loop, which iterates over object properties, with code that first computes the object's properties using the built-in `Object.keys()` function and then iterates through them using a traditional `for` loop [88], often allows for optimization by the JIT compiler in V8 [88].

Although type feedback and type inference optimization strategies have individually demonstrated effectiveness [42], there may still be potential performance gains from combining both approaches, as demonstrated in the work of Hackett et al. [64]. The key insight of their study involves a hybrid type inference algorithm obtained through the fusion of unsound constraint-based static inference of expression and heap value types with targeted dynamic type updates and runtime checks. In this approach, type feedback information is utilized to enhance the effectiveness of type inference.

On a similar note, Kedlaya et al. [71] take a slightly different approach by investigating the interplay between these two methods. They propose two novel methods to combine type feedback and type inference, aiming to amplify their collective benefits while reducing their aggregate overhead. In their approach, type feedback supports type inference by categorizing function invocations based on the types of function arguments at the time of invocation. Conversely, type inference supports type feedback by leveraging inferred type information to strategically place type profiling hooks, thereby significantly minimizing profiling overhead.

5.2 Performance Optimization Techniques

Although numerous efforts have been dedicated to the performance optimization of JavaScript engines and dynamically typed programming languages in general [58, 56, 89, 52, 60, 53], the following section specifically highlights those efforts that are closely related to the work conducted in this thesis.

5.2.1 Concrete Types for TypeScript

StrongScript [85] is an extension of TypeScript featuring a new type system. It introduces a single type constructor for concrete types that offers enhanced semantic guarantees. The type system provides developers with the flexibility to choose among untyped code, where all variables are of type `any`; optionally typed code, which does not affect the semantics of dynamic programs; and concretely typed code, offering traditional correctness guarantees but influencing the semantics of dynamic code.

One of the primary objectives of StrongScript is to enable typing for common programming idioms in JavaScript. As such, all JavaScript programs are considered valid well-typed StrongScript programs. Additionally, the majority of TypeScript programs are compatible with StrongScript.

Concrete types are denoted by `!C` and represent objects that are instances of the corresponding class or its subclasses, exhibiting behaviour akin to types in nominally-statically-typed languages. Static type checking is conducted on these types, and no dynamic checks are necessary in the absence of downcasts. In addition to providing correctness guarantees, concrete types in StrongScript are utilized for optimizing property access.

To effectively utilize concrete types, StrongScript employs a sound type system that necessitates certain modifications to TypeScript's (overly permissive) type rules and the underlying implementation. At runtime, a distinction is made between dynamic objects created using JavaScript syntax (`{ x:y }`) and objects that are instances of a class, created with the `new` keyword. Casts are explicit, and in many cases, they require runtime checks. StrongScript employs nominal subtyping for classes to ensure that the memory layout of parent classes is a prefix of child classes, facilitating fast property access.

The implementation of StrongScript comprises two components: an extension of the TypeScript 0.9.1 compiler and the Truffle JavaScript engine (TruffleJS). While the compiler generates portable JavaScript code capable of running on any JavaScript virtual machine, performance enhancements stem from type-related extensions to the compiler. These extensions include support for concrete types and dynamic contracts at explicit downcasts, checked downcasts (implicit and unsound in TypeScript), and function code suitable for both typed and untyped invocation. The compiler ensures the sound use of concrete types by inserting dynamic contracts wherever unsafe downcasts occur, whether explicit or implicit. This safety is achieved through the `$$check` function, which verifies that a value conforms to a specified type. Additionally, the compiler emits intrinsics that describe the layout of concretely-typed objects, similar to intrinsics emitted by `mtsc` for object layout and access optimizations. Furthermore, TruffleJS is extended to comprehend and utilize these intrinsics for efficient property access in concrete types, similar to the extensions I implemented in JavaScriptCore.

Concrete types allow for object layout to be determined at compile time, enabling direct access to fields and methods by their known locations within the object. This optimization eliminates the need for hash-table lookups. However, JavaScript lacks a

mechanism to explicitly define object layouts. To leverage known concrete objects, the JavaScript code generated by StrongScript includes calls to intrinsic operations. These operations access fields using explicit offsets within objects. While these intrinsics act as no-ops on non-supporting engines, they are implemented as direct accesses on TruffleJS, the only supporting engine. The intrinsics, known as `direct` and `directWrite`, facilitate direct reading and writing to offsets within an object, respectively.

The results from StrongScript and its integration with TruffleJS demonstrate speed-ups ranging between 2% and 32% across a limited number of benchmarks. These performance enhancements stem from type specialization intrinsics and direct access to fields in class instances. Additionally, property access intrinsics remain unaffected by subclass polymorphism, resulting in consistently faster performance. However, when StrongScript-generated code is executed on Node.js (V8), the presence of runtime checks incurs a performance cost, presumably due to V8's lack of support for the developed intrinsics and the additional time spent constructing checked classes.

While StrongScript offers several benefits, its performance optimizations come at the expense of a constrained type system with eager type checking [82]. Furthermore, it introduces new syntax to the TypeScript language, and its implementation is now considered dated. Finally, the argument against hash-table lookups may be obsolete, as most modern JavaScript engines utilize shapes (structures, hidden classes, etc.) for property lookups.

5.2.2 Leveraging Property Access Optimization in V8

Lindroth's work [73] aimed to enhance JavaScript performance in the V8 engine by identifying and eliminating JavaScript code patterns that frequently resulted in deoptimization events [68]. The study specifically focused on the impact of property access optimizations on runtime performance within the V8 JavaScript engine, analyzing real-world application code implementations.

In particular, the research delved into the consequences of deoptimization events caused by incorrect type speculation, which often lead to the removal of optimized machine code and a context switch in the underlying runtime environment.

To assess the performance implications of deoptimization events and the state of inline caches, Lindroth constructed a benchmarking environment to isolate the behaviour of these effects. The first suite (Suite 1) analyzes the impact of inline cache state when functions are optimized and running optimized machine code. A second suite (Suite 2) measures the performance penalty associated with triggering the wrong map deoptimization event. Suite 2 aimed to identify correlations between performance loss due to deoptimizations and inline cache state, as well as the performance cost of modifying inline cache state. Finally, a third suite (Suite 3) investigates how the size of inline caches affects property access performance when executing bytecode through Ignition, the JavaScript interpreter in V8. This setup isolated property lookup operations and facilitated the examination of performance variations by manipulating the size of the inline caches.

Optimized property lookup using a single monomorphic inline cache is significantly faster compared to polymorphic and megamorphic caches within the optimized category (Suite 1). Interestingly, neither the occurrence of a deoptimization event nor an increase in the degree of polymorphism had a significant impact on runtime performance (Suite 2). However, performance regressions from monomorphic property access to polymorphic property access is observed when an additional hidden class is introduced to a monomorphic access. In the interpreted category (Suite 3), there is a much smaller relative difference among the various inline cache states. The results indicate inconclusively a potential linear relationship between the number of introduced hidden classes and runtime performance, despite the experiment primarily focusing on the state of the inline caches rather than the number of hidden classes.

To enhance the performance of JavaScript programs based on the experimental findings, the author proposed five actionable steps:

1. Determining the level of dynamism in the program.
2. Identifying problematic property access sites.
3. Probing objects with a checker function.
4. Implementing changes to the original code.

The checker function in step 3 is designed to identify potential deoptimizations caused by incorrect maps based on a set of rules. The four-step strategy is applied in the analysis of applications and libraries included in the Octane 2.0 Benchmarks [54]. While the optimization proved effective for some application code (e.g., GameBoy, with up to a 24.7% improvement), it performed poorly on others due to the limited capability of the checker function in detecting the cause of deoptimizations [73].

One drawback of this optimization strategy is that it requires manual modification of the code, which increases the risk of introducing bugs, particularly in complex software. Additionally, steps 1 and 2 may be laborious in non-trivial application code. While the author argues that this strategy is broadly applicable to other VMs, a potential downside is that it may lead to conflicts and performance impairment when targeting multiple VMs with the same application code. This impairment is because what may be optimizable in one JavaScript engine may be deoptimized in another.

5.2.3 Typed JavaScript

Typed JavaScript (Typed JS) [57] is a subset of JavaScript specifically designed for running web applications on mobile devices. Its primary focus is on reducing memory footprint and binary size while achieving high performance. Typed JS employs type-decorated syntax (type annotations) and is transpiled to C++ 11. Subsequently, it is compiled ahead-of-time

(AOT) into a target-specific, optimized binary executable. Despite this transformation, Typed JS retains several core JavaScript semantics, including its object model, prototype, functions, closures, and garbage collection.

Furthermore, Typed JS offers type checking and portability benefits to the JavaScript mobile ecosystem. One of its key design principles is strict typing, which mandates that the types of objects be specified during declaration and prohibits runtime type mutation. While Typed JS supports the traditional JavaScript prototypal object model, it also introduces sealed classes for enhanced runtime performance. A sealed class prevents the runtime addition or deletion of properties and is specifically designed to directly map a JavaScript object to a C++ class, further optimizing performance.

Typed JS utilizes a fixed object layout, allowing for object access using memory offsets. It employs property access optimizations using hidden classes and inline caching techniques similar to those found in V8 and JavaScriptCore. Frequently accessed property offsets are cached, enabling direct access to property values using the cached offset. Moreover, Typed JS achieves a smaller memory footprint and binary size through AOT compilation, replacing the need for a VM with a compact native runtime library.

Benchmark experiments conducted using the SunSpider JavaScript Benchmark suite [8] demonstrate that Typed JS is memory-efficient and achieves better performance compared to industry-leading JavaScript engines on the Tizen mobile platform, an open-source mobile operating system developed by Samsung Software R&D Center. Typed JS outperforms V8 and JavaScriptCore by up to 3.5× while consuming up to 20× less memory. However, Typed JS is surpassed by JavaScriptCore (and V8) on recursive and mathematical test suites, reportedly due to inefficient lambda functions in C++11 and the absence of JIT compilation. Furthermore, when the same tests are conducted in a Linux desktop environment, similar results are observed, albeit with a reduced magnitude of difference. Test suites are observed to run up to 10× faster on a desktop platform.

Although Typed JS achieves impressive performance and aims to become the default language for mobile applications, its compilation of JavaScript code is platform-specific, requiring the same code to be compiled for multiple platforms. Additionally, it suffers from poor string performance due to suboptimal string implementation. Furthermore, the requirement for type annotations, stemming from a lack of mature type inference implementation changes the language and may hinder code reuse within a large JavaScript application codebase that targets multiple platforms.

Chapter 6

Future Work

An extension that could enhance this work is the integration of the developed intrinsic into the optimizing tiers, namely the DFG and FTL JIT compilers. However, it is reasonable to suspect that any performance improvement from this addition would be minimal, given that both tiers already implement numerous optimizations, particularly related to property access.

Presently, optimizations are erased when an object's shape is mutated through property or field deletion, even though the object's type may otherwise undergo nominal usage. Developing new heuristics based on usage patterns to address this limitation could enhance performance when such scenarios arise.

Alternatively, some analysis may be performed to determine if such delete operations may have an adverse result on an object's shape, or have no effect at all. For instance, an object whose property is deleted but never dynamically assigned a new property may still maintain a stable shape, albeit with extra unused storage, and may benefit from nominal optimizations.

Furthermore, current optimizations do not extend to inheritance. An important future direction for this work is to include support for property access expressions in inherited classes, ensuring that classes adhere to a layout where property access in any object, whether superclass or subclass, can be directly mapped to an offset at compile time. This expansion would broaden the optimization scope to larger application codebases. Additional intrinsics could be developed to further optimize performance by targeting frequently used patterns, similar to how superinstructions [78] expedite bytecode execution by combining multiple instructions into one. Such extensions have the potential to double current performance speedups.

The implementation currently parses JavaScript code exclusively in `Builtin` mode in JavaScriptCore. This approach may raise security concerns, as it grants user code access not only to intrinsics relevant to this work but also to all preexisting intrinsics in JavaScriptCore. Misuse of intrinsics can lead to undefined behaviour, segmentation faults,

and memory-related issues. A potential solution is to modularize intrinsics in JavaScriptCore by exporting only the required intrinsics in a non-builtin (`NotBuiltin`) mode, which the TypeScript compiler can target.

A possible future direction is integrating `mtsc` into the upstream TypeScript compiler for production. However, considering that TypeScript is a fast-moving language, and the core developers do not prioritize optimizations over industrial usage issues, this may not be feasible.

Furthermore, in a scenario where `mtsc` is integrated into production, there is still a need to migrate the implemented optimizations in JavaScriptCore to other JavaScript engines, as `mtsc` currently only targets JavaScriptCore. Ultimately, this could potentially become more of a burden than a benefit.

Chapter 7

Conclusions

JavaScript is a programming language that is widely used in web and application development. Its dynamic nature presents challenges for performance optimizations. State-of-the-art engines employ various optimization strategies, including the utilization of sophisticated just-in-time compilers (JITs) to compile JavaScript code at runtime for performance enhancements.

This thesis proposes a method to enhance the performance of JavaScript using TypeScript. TypeScript's type system is classified into nominal and non-nominal types, with nominal types (linked to consistent use of classes) being particularly relevant and applicable for optimization. Considering that a significant portion of JavaScript's performance hinges on optimizing objects and operations performed on them, the implemented optimizations primarily target property access operations on objects. These optimizations are integrated into both the TypeScript compiler and JavaScriptCore, the JavaScript engine utilized in the WebKit project, which powers the Safari browser.

The effectiveness of these optimizations is empirically evaluated using two sets of benchmarks. The first benchmark (B_{jet}) comprises six test programs from the JetStream 2.1 JavaScript benchmark suite, widely utilized by several browser engines. The second is a microbenchmark (B_{μ}) consisting of four test programs. The results from both benchmarks indicate an improvement in runtime performance when a nominal typing style is employed in TypeScript.

The empirical evaluation of the optimizations reveals that for JavaScript code heavily utilizing objects and property access operations, such operations may significantly impact performance. While the optimizations are implemented in JavaScriptCore, similar performance improvements can be expected in modern JavaScript engines, as they model objects in a similar manner and apply comparable optimizations.

Although property access optimizations offer performance improvements, the speedups in LLInt (T_L) in the B_{jet} benchmark were slightly lower than anticipated. This outcome is surprising considering that LLInt performs no specific optimizations apart from the use of inline caches.

An especially intriguing idea is to compare the results obtained in this thesis with the application of these optimizations to another multi-tiered JavaScript engine, such as V8. Such a comparison may help answer the question of whether optimizations provide benefits in the interpreter tier beyond the security considerations imposed by a JIT.

A particular challenge in implementing the optimizations is developing reliable heuristics with applicable optimization techniques. Despite TypeScript being statically typed, it exhibits unsoundness in its type system. Hence, relying on the type system for guarantees required by several optimization strategies is challenging, making optimizations difficult to apply. This thesis implements optimizations for access operations on nominal types. Extending the heuristics developed by streamlining them can enable more optimization possibilities.

While this thesis primarily focuses on structure-related optimizations in JavaScript, there is potential to further extend the idea of leveraging types in TypeScript for non-intrusive JavaScript optimization by identifying optimization opportunities within the architecture of JavaScript engines that are suitable for the type system. The main goal of the thesis is to demonstrate the feasibility of this approach and to serve as inspiration for future research.

References

[1] JavaScript Interpreter. https://firefox-source-docs.mozilla.org/js/index.html#javascript-interpreter.

[2] Spidermonkey. https://firefox-source-docs.mozilla.org/js/index.html.

[3] The WebKit Open Source Project. https://webkit.org/project/.

[4] *Measuring program similarity for efficient benchmarking and performance analysis of computer systems*. PhD thesis, 2006.

[5] What is V8? https://v8.dev/, 2008.

[6] Announcing TypeScript 0.8.1. https://devblogs.microsoft.com/typescript/announcing-typescript-0-8-1/, 2012.

[7] Typescript: JavaScript Development at Application Scale. 2012.

[8] Announcing SunSpider 1.0. https://www.webkit.org/blog/2364/announcing-sunspider-1-0/, 2013.

[9] JavaScriptCore. https://trac.webkit.org/wiki/JavaScriptCore, 2014.

[10] Sources | JavaScriptCore. https://github.com/WebKit/WebKit, 2014.

[11] Firing up the Ignition Interpreter. https://v8.dev/blog/ignition-interpreter, 2016.

[12] Ignition. https://v8.dev/docs/ignition, 2016.

[13] A New Bytecode Format for JavaScriptCore. https://www.webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/, 2019.

[14] JavaScriptCore Internals Part 1: Tracing JavaScript Source to Bytecode. https://zon8.re/posts/jsc-internals-part1-tracing-js-source-to-bytecode, 2020.

[15] Speculation in JavaScriptCore. https://webkit.org/blog/10308/speculation-in-javascriptcore/, 2020.

[16] TypeScript Design Goals | Non-goals. https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals#non-goals, 2020.

[17] TypeScript Compiler Internals. https://basarat.gitbook.io/typescript/overview, 2021.

[18] Jetstream 2. https://webkit.org/blog/13146/introducing-jetstream-2-1/, 2022.

[19] JetStream 2. https://browserbench.org/JetStream/, 2022.

[20] JetStream 2 In-Depth Analysis. https://browserbench.org/JetStream/in-depth.html, 2022.

[21] Automatic semicolon insertion. 2023.

[22] Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS. https://www.electronjs.org/, 2023.

[23] IIFE. https://developer.mozilla.org/en-US/docs/Glossary/IIFE, 2023.

[24] Inheritance and the prototype chain. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain, 2023.

[25] JetBrains Dev Report: TypeScript Is Fastest-Growing Programming Language. https://visualstudiomagazine.com/articles/2023/02/02/jetbrains-survey.aspx, 2023.

[26] Object prototypes. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes, 2023.

[27] What is type erasure? https://github.com/microsoft/TypeScript/wiki/FAQ#what-is-type-erasure, 2023.

[28] Angular | The web development framework for building the future. https://angular.io/, 2024.

[29] Bun is a JavaScript runtime. https://bun.sh/, 2024.

[30] Chrome | The browser built to be yours. https://www.google.com/intl/en_ca/chrome/, 2024.

[31] Commits | JavaScriptCore. https://github.com/WebKit/WebKit/commits, 2024.

[32] Hello, nest! *https://nestjs.com/*, 2024.

[33] Introduction to Node.js. https://nodejs.org/en/learn/getting-started/introduction-to-nodejs, 2024.

[34] Python. https://www.python.org/, 2024.

[35] React Native | Learn once, write anywhere. https://reactnative.dev/, 2024.

[36] Safari. Blazing fast. Incredibly private. https://www.apple.com/ca/safari/, 2024.

[37] Speedometer 3. https://www.browserbench.org/Speedometer3.0/about.html, 2024.

[38] Speedometer 3.0: The Best Way Yet to Measure Browser Performance. https://webkit.org/blog/15131/speedometer-3-0-the-best-way-yet-to-measure-browser-performance/, 2024.

[39] The Computer Language 23.03 Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html, 2024.

[40] Visual Studio Code | Code editing. Redefined. https://code.visualstudio.com/, 2024.

[41] Ole Agesen. *Concrete type inference: delivering object-oriented applications*. PhD thesis, Stanford, CA, USA, 1996. UMI Order No. GAX96-20452.

[42] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, page 91–107, New York, NY, USA, 1995. Association for Computing Machinery.

[43] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. Improving JavaScript Performance by Deconstructing the Type System. PLDI '14, page 496–507, New York, NY, USA, 2014. Association for Computing Machinery.

[44] Marcus Arnström, M. G. Christiansen, and Daniel Sehlberg. Prototype-based programming. 2003.

[45] D.C. Atkinson and W.G. Griswold. The design of whole-program analysis tools. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 16–27, 1996.

[46] John Aycock. A Brief History of Just-in-Time. volume 35, page 97–113, New York, NY, USA, jun 2003. Association for Computing Machinery.

[47] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. volume 2005, pages 15–26, 01 2005.

[48] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 351–364, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, page 257–281, Berlin, Heidelberg, 2014. Springer-Verlag.

[50] Camillo Bruni. A closer look at crankshaft, v8's optimizing compiler. https://wingolog.org/archives/2011/08/02/a-closer-look-at-crankshaft-v8s-optimizing-compiler, 2017.

[51] Camillo Bruni. Fast properties in V8. https://v8.dev/blog/fast-properties, 2017.

[52] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. Compiler-assisted object inlining with value fields. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 128–141, New York, NY, USA, 2021. Association for Computing Machinery.

[53] John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. Type-Based Gradual Typing Performance Optimization. volume 8, New York, NY, USA, jan 2024. Association for Computing Machinery.

[54] Stefano Cazzulani. Octane: the JavaScript benchmark suite for the modern web. https://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html, 2012.

[55] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 49–70, New York, NY, USA, 1989. Association for Computing Machinery.

[56] Jiho Choi, Thomas Shull, and Josep Torrellas. Reusable inline caching for JavaScript performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 889–901, New York, NY, USA, 2019. Association for Computing Machinery.

[57] Ryan H. Choi and Youngil Choi. A lightweight JavaScript engine for mobile devices. In *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*, MobileDeLi 2015, page 3–4, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023, page 34–46, New York, NY, USA, 2023. Association for Computing Machinery.

[59] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, page 297–302, New York, NY, USA, 1984. Association for Computing Machinery.

[60] Gem Dot, Alejandro Martínez, and Antonio González. Removing checks in dynamically typed languages through efficient profiling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 257–268. IEEE Press, 2017.

[61] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. volume 54, New York, NY, USA, may 2021. Association for Computing Machinery.

[62] ECMA International. *ECMAScript® 2022 Language Specification*. ecma international, 13 edition, June 2022.

[63] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 239–250, New York, NY, USA, 2012. Association for Computing Machinery.

[64] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 239–250, New York, NY, USA, 2012. Association for Computing Machinery.

[65] John L. Hennessy and David A. Patterson. *The Ruby Programming Language*. O'Reilly Media, Inc, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st edition, 2008.

[66] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[67] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, page 21–38, Berlin, Heidelberg, 1991. Springer-Verlag.

[68] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 32–43, New York, NY, USA, 1992. Association for Computing Machinery.

[69] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. PLDI '94, page 326–336, New York, NY, USA, 1994. Association for Computing Machinery.

[70] Md Saiful Islam. JavaScript alternative (TypeScript) and its effectiveness in web development. diplomathesis, Tampere University of Applied Sciences, 2023.

[71] Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, page 37–48, New York, NY, USA, 2013. Association for Computing Machinery.

[72] Gary King. How Not to Lie With Statistics: Avoiding Common Mistakes in Quantitative Political Science. volume 30, page 666–687, August 1986.

[73] Jonathan Lindroth. Leveraging property access optimization in the V8 JavaScript engine for improved runtime performance. diplomathesis, University of Uppsala, 2021.

[74] Microsoft. abstract Classes and Members. [77].

[75] Microsoft. Contextual Typing. [77].

[76] Microsoft. Narrowing. [77].

[77] Microsoft. *The TypeScript Handbook*. Microsoft, 2024.

[78] Lukas Miedema. QuickInterp - Improving interpreter performance with superinstructions. June 2020.

[79] David J. Pearce. Sound and Complete Flow Typing with Unions, Intersections and Negations. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 335–354, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[80] David Peter. hyperfine, March 2023.

[81] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 167–180, New York, NY, USA, 2015. Association for Computing Machinery.

[82] Gregor Richards, Ellen Arteca, and Alexi Turcotte. The VM already knew that: leveraging compile-time knowledge to optimize gradual typing. volume 1, New York, NY, USA, oct 2017. Association for Computing Machinery.

[83] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 677–694, New York, NY, USA, 2011. Association for Computing Machinery.

[84] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[85] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete Types for TypeScript. In *European Conference on Object-Oriented Programming*, 2015.

[86] Theodore H. Romer, Dennis Lee, Geoffrey M. Voelker, Alec Wolman, Wayne A. Wong, Jean-Loup Baer, Brian N. Bershad, and Henry M. Levy. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 150–159, New York, NY, USA, 1996. Association for Computing Machinery.

[87] Marija Selakovic and Michael Pradel. Automatically Fixing Real-World JavaScript Performance Bugs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, page 811–812. IEEE Press, 2015.

[88] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: An empirical study. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 61–72, 2016.

[89] Manuel Serrano. Of JavaScript AOT compilation performance. volume 5, New York, NY, USA, aug 2021. Association for Computing Machinery.

[90] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[91] Jeremy G. Siek. Gradual Typing for Functional Languages. 2006.

[92] Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4–27, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[93] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. Uncovering JavaScript Performance Code Smells Relevant to Type Mutations. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS 2015)*, pages 335–355, November 2015.

APPENDICES

Appendix A

`Symbol` and `PropertyAccessExpression`

A property access expression in TypeScript is represented as an AST node, defined by the interface `PropertyAccessExpression`. This node contains four members: `kind`, `expression`, `name`, and an optional `questionDotToken`. The `kind` member is an `enum` type called `SyntaxKind`, which represents both the TypeScript token and AST node types. Specifically, the `kind` of a property access expression is set to `SyntaxKind.PropertyAccessExpression`. The `expression` member represents the left-hand side expression, while the `name` member represents the property name found in the access expression. Figure A.1 presents the definition of `PropertyAccessExpression` in TypeScript.

```
interface PropertyAccessExpression extends MemberExpression, NamedDeclaration {
    readonly kind: SyntaxKind.PropertyAccessExpression;
    readonly expression: LeftHandSideExpression;
    readonly questionDotToken?: QuestionDotToken;
    readonly name: MemberName;
}
```

Figure A.1: Definition of `PropertyAccessExpression` AST node.

A `Symbol` connects declaration AST nodes to other declarations contributing to the same entity [17]. `Symbols` are produced by the Binder in `tsc`. Figure A.2 presents the definition of `Symbol` in TypeScript.

```
interface Symbol {
    flags: SymbolFlags;                 // Symbol flags
    escapedName: __String;              // Name of symbol
    declarations?: Declaration[];       // Declarations associated with this symbol
    valueDeclaration?: Declaration;     // First value declaration of the symbol
    members?: SymbolTable;              // Class, interface or object literal instance members
    exports?: SymbolTable;              // Module exports
    id?: SymbolId;                      // Unique id (used to look up SymbolLinks)
    mergeId?: number;                   // Merge id (used to look up merged symbol)
    parent?: Symbol;                    // Parent symbol
}
```

Figure A.2: Representation of `Symbol` in TypeScript.

Appendix B

Nominal Type Properties for Object Layout

In some cases, symbols corresponding to multiple types may be merged by the binder, for example, classes declared in separate sources but with the same name. To prevent the utilization of properties not defined in the actual class for which the check is being performed, the sorted properties are filtered again to include only properties whose parent nodes have the same ID as that of the type being inspected (`typ`). Figure B.1 presents a pseudocode for computing object layout information.

```
const isNotMethodOrFunction = (nd: Node) =>
    !(isMethodDeclaration(nd) || isFunctionDeclaration(nd) || isFunctionExpression(nd));
let properties = (typ.declaredProperties || typ.properties).filter(
    (sym: Symbol) => isNotMethodOrFunction(sym.valueDeclaration!)
);
properties.sort((a: Symbol, b: Symbol) =>
    a.valueDeclaration!.pos - b.valueDeclaration!.pos);
properties = properties.filter((propSym: Symbol) => (
    propSym.parent && propSym.parent.valueDeclaration &&
    propSym.parent.valueDeclaration.id === typ.symbol.valueDeclaration.id
));
// layout metadata
const namedProperties = properties.map((x: Symbol) => x.escapedName)
                            .slice(0, maxInlineSlots);
```

Figure B.1: Computing layout metadata.

Appendix C

`tsconfig.json` support for `optimizeWithTypes`

The integration of `optimizeWithTypes` involves the addition of a new parameter to the `CompilerOptions` interface, as well as adjustments to the command-line options utilized in `tsc`. Figures C.1 and C.2 present the `optimizeWithTypes` configuration in the compiler and command-line options, respectively.

```
interface CompilerOptions {
    allowJs?: boolean;
    allowUnreachableCode?: boolean;
    allowUnusedLabels?: boolean;
    alwaysStrict?: boolean;
    checkJs?: boolean;
    noEmitOnError?: boolean;
    noErrorTruncation?: boolean;
    noFallthroughCasesInSwitch?: boolean;
    noImplicitAny?: boolean;
    noImplicitReturns?: boolean;
    noImplicitThis?: boolean;
    optimizeWithTypes?: boolean;
}
```

Figure C.1: `optimizeWithTypes` configuration in `CompilerOptions` type definition.

```
const commandOptionsWithoutBuild: CommandLineOption[] = [
    // CommandLine only options
    {
        name: "version",
        shortName: "v",
        type: "boolean",
        showInSimplifiedHelpView: true,
        category: Diagnostics.Command_line_Options,
        description: Diagnostics.Print_the_compiler_s_version,
        defaultValueDescription: false,
    },
    // ...
    {
        name: "optimizeWithTypes",
        type: "boolean",
        category: Diagnostics.Language_and_Environment,
        description: Diagnostics.Allow_optimizations_with_types_when_possible,
        defaultValueDescription: false,
    },
]
```

Figure C.2: `optimizeWithTypes` configuration in `CommandLineOption` array.

Appendix D

`Builtin` mode in JavaScriptCore

`Builtin` mode in JSC is used for parsing and compiling built-in JavaScript modules, which extensively utilize well-known symbols and built-in names as seen in Figure D.1.

```
1  function concat(first)
2  {
3      "use strict";
4
5      if (@argumentCount() === 1
6          && @isJSArray(this)
7          && @tryGetByIdWithWellKnownSymbol(this, "isConcatSpreadable") === @undefined
8          && (
9              !@isObject(first)
10             || @tryGetByIdWithWellKnownSymbol(first, "isConcatSpreadable") === @undefined)
11         ) {
12
13         var result = @concatMemcpy(this, first);
14         if (result !== null)
15             return result;
16     }
17
18     return @tailCallForwardArguments(@concatSlowPath, this);
19 }
```

Figure D.1: Implementation of `concat` for `ArrayPrototype.js` in JSC [10].