

Author
David Leopoldseder, BSc.

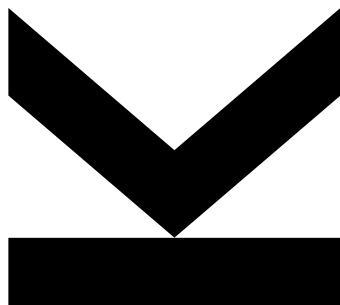
Submission
Institute for System Software

Thesis Supervisor
**o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c.
Hanspeter Mössenböck**

Assistant Thesis Supervisor
**Dipl.-Ing. Dr. Lukas Stadler
Dipl.-Ing. Dr. Matthias Grimmer**

Linz, December 2015

Graal AOT JS: A Java to JavaScript Compiler



Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

Abstract

Executing Java code inside the browser has been targeted by researchers for many years. Therefore, different approaches proposed solutions adding plug-ins to the browser or extending it with additional runtimes, to enable the execution of Java code in the Browser. However, those approaches neglected to perceive that users typically do not want to install additional software enabling the execution of Java in the browser. The community realized that only the translation of Java to JavaScript code can permanently satisfy the execution of Java in the browser. However, no approaches existed during writing this thesis that supported the compilation of the JDK to the browser, while simultaneously featuring high runtime performance and reasonable code size. The Java programming language has a large and well tested API and comes with an enormous code base. Reusing this code in the browser would significantly reduce the development effort of JavaScript. JavaScript is the assembly language of the web and offers a suitable target platform for cross compilation from Java. This thesis presents a novel Java bytecode to JavaScript compiler implemented in Java. Compilation happens ahead of time. It performs a static analysis to reduce code size of the generated JavaScript code. The compiler applies several optimizations to increase runtime performance of the generated JavaScript code. The generated code runs with an average slowdown from $2\times$ to $20\times$ compared to the HotSpot Server compiler. The compiler allows to reuse existing general purpose Java code, including the JDK, in the browser. The generated JavaScript code runs with high performance and is reasonable in size.

Kurzfassung

Die Ausführung von Java Code im Browser ist seit Jahren ein Forschungsziel. Verschiedene Ansätze schlugen die Entwicklung von Browser-Plugins, oder die Erweiterung des Browsers mit zusätzlichen Ausführungsumgebungen, vor, um Java im Browser ausführbar zu machen. Allerdings vernachlässigten diese Ansätze die Tatsache, dass User normalerweise keine zusätzlichen Browser Plugins installieren wollen. Die Community hat realisiert, dass nur die Umwandlung von Java Code zu JavaScript die Ausführung von Java Code im Browser dauerhaft garantieren kann. Allerdings existierten zur Zeit des Schreibens dieser These keine Ansätze, die die Kompilierung des JDK für den Browser unterstützen, und gleichzeitig hohe Laufzeit Performanz und akzeptable Codegröße boten. Die Programmiersprache Java hat eine große und gut getestete API und umfasst eine enorme Codebasis. Die Wiederverwendung dieser Codebasis im Browser würde den Entwicklungsaufwand von JavaScript deutlich reduzieren. JavaScript ist die Assemblersprache des Web und bietet eine passende Zielplatform für die Übersetzung von Java zu JavaScript. Diese These präsentiert einen neuen, in Java implementierten, Java Bytecode zu JavaScript Übersetzer. Die Übersetzung passiert bevor der Code ausgeführt wird und führt eine statische Analyse auf dem Code aus um ihn später zu Verkleinern. Der Übersetzer führt verschiedene Optimierungen auf dem generierten JavaScript Code aus, um die Laufzeit Performanz zu erhöhen. Der generierte Code braucht bei der Ausführung 2x bis 20x länger als eine Ausführung des Java Codes mit dem HotSpot Server Übersetzer. Der Übersetzer erlaubt die Wiederverwendung von existierendem Java Code, unter Verwendung des JDK, im Browser. Der generierte Code hat eine hohe Laufzeit Performanz und eine annehmbare Größe.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	Challenges	3
1.4	Structure of the Thesis	4
2	System Overview	5
2.1	JVM - Java Virtual Machine	5
2.1.1	HotSpot Java VM	6
2.2	Graal	8
2.2.1	Graal IR	8
2.2.2	Graal Tiers	9
2.3	Substrate Virtual Machine	10
2.3.1	Static Analysis	10
3	Graal AOT JS Compiler	12
3.1	Frontend	13
3.2	Backend	14
3.3	Architecture	14
3.3.1	Components	15
3.3.2	Node Lowering	16
3.4	Limitations	16
4	Control Flow Reconstruction	18
4.1	Structured Control Flow	21
4.2	Graph Tagging	30
4.2.1	Algorithm	30
4.2.2	Block Interpreter	34
4.3	Reconstruction Pipeline	35

5 JavaScript Code Generation	43
5.1 Compilation of the Java Type System to JavaScript	43
5.1.1 Primitives	43
5.1.2 Objects	44
5.2 Constant Data	47
5.3 Exception Handling	48
5.4 JavaScriptify - Compiler Intrinsics & Native Calls	50
5.5 Unsafe Memory Access	51
6 Ahead-of-Time Optimizations	52
6.1 Graal Optimizations	52
6.2 Exception Handler Optimizations	53
6.3 SSA Node Inlining	53
7 Evaluation	58
7.1 Benchmarks	59
7.2 SSA Node Inlining Optimization	60
7.3 Control Flow Reconstruction	60
7.4 Code Size	61
7.5 Run-time Performance	63
8 Related Work	66
8.1 Structured Control Flow Reconstruction & Decompilation Existing Approaches	66
8.2 Java to JavaScript Translation	68
8.2.1 Offline Transformation	69
8.2.2 Online Transformation	71
8.2.3 Online Bytecode Interpreters	72
9 Future Work	74
10 Conclusion	76
Acknowledgments	77
Bibliography	82

Chapter 1

Introduction

This chapter presents an approach to cross-compile Java bytecodes to JavaScript henceforth called Graal AOT JS. Graal AOT JS is an ahead-of-time Java bytecode to JavaScript compiler building on the Graal VM. It presents the advantages of cross-compiling Java bytecode to JavaScript but also the challenges for such a compiler. At the end of the chapter a short outline of the entire thesis is given.

1.1 Motivation

JavaScript has become the major language of today's web development. It is the assembly language of the 21st century offering a wide set of development tools, APIs and supported platforms [30]. However, it still suffers from several disadvantages compared to a language like Java. It is untyped, which makes it potentially more error prone than typed languages. It has no language intrinsic modularization paradigms and no elaborated mechanism for information hiding. In contrast, Java is the number one managed language for server applications [61] and offers run-time performance comparable to one of unmanaged languages [7]. Java is typed, has an elaborated exception handling mechanism and there are Java implementations for most industry use cases available.

We propose that merging the advantages of both languages into one system is a desirable research goal. Such a system would enable safe development and the reuse of existing Java code in the browser. Clients could execute the code without additional run-time dependencies and profit from competitive performance.

A possible solution for this problem is the compilation from Java to JavaScript code. Java libraries, including the Java Development Kit (JDK) could be compiled to JavaScript and executed inside the browser without additional dependencies. For the feasibility of such a tool, the resulting code must be easy to deploy and offer competitive performance.

This thesis proposes one possible solution for the task of executing arbitrary Java code inside the browser. It presents an ahead-of-time Java bytecode to JavaScript compiler that runs on-top of the Graal just-in-time compiler. Graal is a novel dynamic Java JIT compiler. Graal AOT JS enables the compilation of a large set of the JDK to the browser. The compilation pipeline of Graal AOT JS is easy to use. The generated JavaScript code executes with competitive performance on modern JavaScript VMs. The compilation process happens offline, thus ahead-of-time. It collects all dependencies for a given target application and outputs a standalone independent JavaScript file. The compiled code is easy to deploy and does not need additional libraries, network communications or runtime compilations. Compilation of Graal AOT JS uses Graal's high level intermediate representation as an input and generates JavaScript code. Graal AOT JS applies optimizations on the intermediate representation (IR) to increase the performance of the generated JavaScript code. Graal offers such high level optimizations including global value numbering, constant folding, conditional elimination, strength reduction and a partial escape analysis. Additionally, this thesis proposes optimizations on the generated code that are unique to the code generation of JavaScript. We demonstrate the feasibility of the approach with a set of standard Java and JavaScript benchmarks.

1.2 Contributions

This thesis contributes a novel approach for high level language (HLL) code generation from a just-in-time compiler's IR. Additionally, it presents a novel control flow reconstruction algorithm to generate structured HLL code from possibly unstructured bytecode. Other concepts presented by this thesis include:

- The support for compilation of arbitrary Java bytecode to JavaScript code
- The usage of preexisting static analysis technique to decrease code size of the resulting JavaScript code
- AOT optimizations on Graal's IR during compilation to increase the performance of the generated JavaScript code

This thesis presents all aspects of Graal AOT JS. It describes the steps during compilation and the theory behind them.

To evaluate the presented approach and to verify the feasibility, we conducted several experiments evaluating different aspects of the compiler including optimizations, code size and run-time performance. Experiments for the different concepts of interest were all conducted using a set of industry standard Java benchmarks of several sizes, each of which targeting different concepts of the tested language like, e.g., floating point arithmetic, integer arithmetic or heavy usage of control flow statements.

The Graal AOT JS Java bytecode to JavaScript compiler has already been published in a conference paper called "*Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR*" [34]. This thesis is an extended version of this paper. It also presents the surrounding eco system of the compiler.

1.3 Challenges

To compile Java bytecode to JavaScript, several challenges arise from the different paradigms of the source and target language. These differences are:

- **Control Flow Reconstruction:** Bytecode expresses control flow differently than Java source-code. Java bytecode [35] is not defined to be structured nor reducible (see Chapter 4). Java high level language control flow instructions are compiled to conditional and unconditional jumps in bytecode. To produce efficient JavaScript code, it is necessary to analyze the bytecode and reconstruct structured high level language control flow instructions.
- **Constant & Static Data:** Certain high level optimizations on the IR, like e.g. constant folding or conditional elimination, require the static and constant data of Java classes to be known at compile time. To achieve this, Graal AOT JS loads classes prior to compilation and uses constant reflection information for optimizations. This requires a heap compilation strategy to map constant and static data to JavaScript.
- **Type System:** The entire Java type-system including inheritance, interfaces and type checks must be mapped to semantically equivalent concepts in JavaScript.

- Intrinsics, Native Calls, etc: There are several concepts used in the JDK and the Graal compiler to optimize machine code generated by they dynamic just-in-time compiler. Not all of these concepts can be mapped to JavaScript. E.g., native calls cannot be compiled to JavaScript, as this would require to parse and compile C and C++ code.
- Exception Handling: Although JavaScript and Java share a common semantic for exception handlers, JavaScript is less strict with runtime exceptions than Java. The mapping of the Java exception semantic to JavaScript also requires a special treatment to achieve high run-time performance. JavaScript programs do not use exceptions as excessive as Java programs, thus JavaScript VMs typically do not optimize exception handlers properly.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows: Chapter 2 presents the foundation this thesis is based on. It describes Java, the Java virtual machine, Graal and the Substrate Virtual Machine.

Then, Chapter 3 describes the architecture of Graal AOT JS and the surrounding eco system with Graal and the Substrate VM.

Chapter 4 presents the parts of Graal AOT JS that compile Java bytecode to JavaScript and how it reconstructs control flow.

Chapter 5 presents the mapping of relevant Java language paradigms to JavaScript including the type system mapping, handling of static and constant data and how exception semantic is presented in the generated JavaScript code.

Furthermore, Chapter 6 presents optimizations to increase the run-time performance of the generated JavaScript code.

Chapter 7 contains a description of the experiments we conducted to show the feasibility of the presented compilation approach.

A comprehensible comparison of Graal AOT JS with related work is given in Chapter 8.

The thesis concludes with the future work in Chapter 9 and a conclusion in Chapter 10.

Chapter 2

System Overview

This chapter explains the context of Graal AOT JS which is built on-top of the Graal VM, a modified version of the HotSpot JVM. For this purpose this chapter introduces the Java platform, the concepts of a JVM and it provides an overview of the HotSpot JVM. In the second part of the chapter more advanced concepts of Graal are explained covering Graal's IR, compilation tiers and optimizations. The chapter ends with a short overview of Substrate VM's static analysis.

2.1 JVM - Java Virtual Machine

Java [23] is a managed, general-purpose programming language developed by Sun Micro Systems, which was bought by Oracle in 2010 [49]. Currently Java is the most popular programming language with a relative distribution of around 20% [61]. Java's popularity can be attributed to a multitude of reasons, the most important ones being security, platform independence and performance.

Java programs are executed in a separate environment called the *Java Runtime Environment (JRE)*. A Java runtime environment contains the major component of the runtime system called a *Java Virtual Machine (JVM)* which underlies the Java Virtual Machine Specification [35]. The JVM is an abstract computing machine that handles hardware and operating system independence. It is also the reason for the small size of compiled Java code and it acts as a *sandbox* to protect the operating system and the host in general from malicious programs. A JVM does not execute Java source code, but a JVM specific machine-code like instruction set called *Java bytecode*. This bytecode is stored in so-called *class-files*. Java classes are compiled by the Java compiler to class files which can be executed on the JVM. Java bytecode is platform independent and can be easily deployed to any platform that is supported by the JVM. The JVM is the platform dependent part. Typical JVM implementations feature support for several operating systems and CPU architectures. Java's approach of being platform independent on the bytecode level is known as "*Write once, run everywhere*".

2.1.1 HotSpot Java VM

The HotSpot JVM [48] is the current leading Java virtual machine on the market. It had proven to be the fastest and most reliable JVM implementation over nearly two decades. HotSpot combines several compilers, interpreters and GCs that are widely known as state-of-the-art virtual machine techniques.

Graal AOT JS builds ontop of Graal. The Graal compiler [47] initially was a port of the HotSpot client compiler from C++ to Java. There are imports parts of Graal, like, e.g., the IR, that are heavily influenced by HotSpot. Therefore, this section describes the HotSpot JVM in more detail and provides an overview of the execution paradigms of HotSpot as well as the two compilers and their IRs.

Mixed Mode Execution

HotSpot features a mixed mode execution model with an interpreter and two different just-in-time (JIT) compilers. The *client compiler* [32] and the *server compiler* [50].

HotSpot's aggressive optimizations are feedback driven. The two dynamic JITs generate code based on profiling data recorded during execution of the code in one of the two interpreters. There are two atomic counters that are interesting for each method: the number of invocations and the number of back edges taken in a loop (which actually is the number of times the loop header was executed in a consecutive run of a loop). Based on those counters, a method (or loop) can get *hot*, if a counter hits a threshold. Such a method is then called a *hot spot*. Based on the configuration of the VM and the configured compilation strategy one of the two JITs compiles the hot method or the hot loop. The JIT compilers perform feedback directed optimizations. This means they use profiling information to further optimize the machine code. Optimized code have been compiled based on assumptions that can be invalidated later during execution. E.g., an inlining decision of a dynamic method was based on the assumption that the callsite is monomorphic as only one possible dynamic receiver was loaded. Class loading may now invalidate this assumptions as another receiver type gets loaded. Therefore, the callsite may no longer be monomorphic and the optimized code gets incorrect. Such invalidations are handled via the concept of deoptimization [27]. Deoptimization is the process of transferring execution from an optimized frame to a more general, un-optimized frame. Normally, this means to transfer execution from JIT compiled native code to one of the two interpreters. For this process, there needs to be a mapping from locals, stack and registers of native code to the interpreter.

Client Compiler

The client compiler is the younger JIT compiler of HotSpot. It is written in C++ and is known for low compilation time and reasonable good code quality.

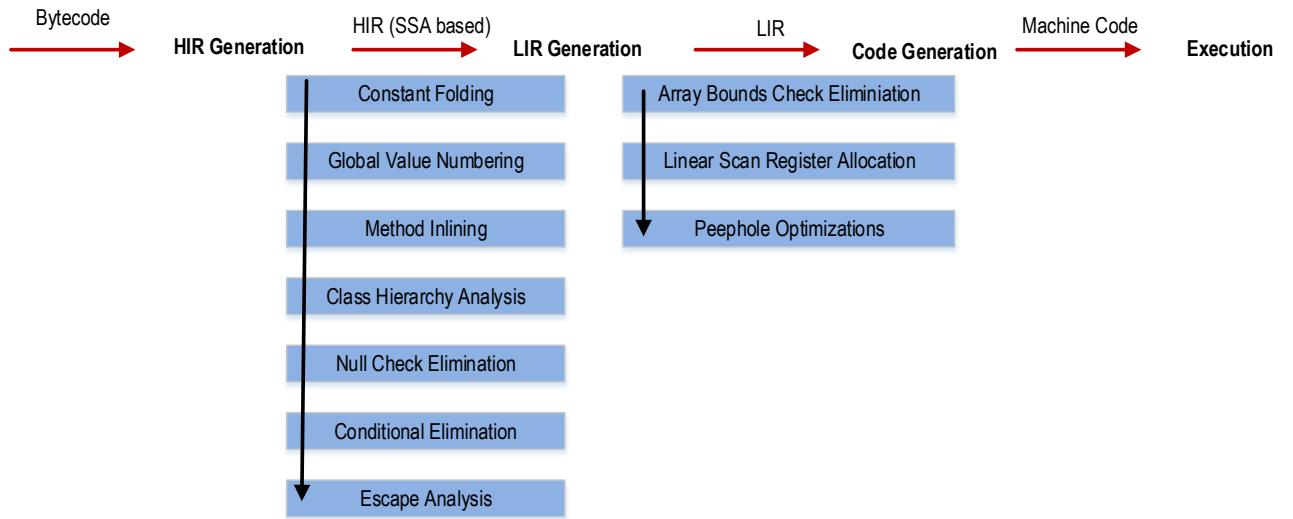


Figure 2.1: HotSpot Client Compiler IR

Figure 2.1 shows the compilation pipeline of the client compiler. The bytecode parser builds the client compiler's high level intermediate representation (HIR). HIR is a static-single-assignment (SSA) [11] based intermediate representation that expresses control flow with a control flow graph (CFG). The nodes of the CFG are called basic blocks. A basic block is the longest possible sequence of instructions with incoming branches on the first instruction and outgoing branches on the last instruction. Therefore a basic block has at least one entry instruction, the first instruction, and one exit instruction, the last instruction. The client compiler applies high level Java optimizations like constant folding and inlining on the HIR. After those optimizations the compiler constructs the low level intermediate representation (LIR) from the HIR. While HIR is still platform independent LIR is not. LIR is a three-operand (like x86 assembler) machine code that still contains some high level instructions for object allocation and locking. LIR already uses operands that represent a certain memory location like a register, stack slot or constant. The last step of the compilation pipeline marks the register allocation before machine code is emitted. The register allocator of the client compiler is a linear scan register allocator [67].

Server Compiler

The server compiler uses a different intermediate representation than the client compiler. Where the client compiler uses a CFG with basic blocks and instruction nodes in the HIR and three-operand instructions in the LIR, the server compiler is based on the *sea of nodes* IR [9, 10] approach. The approach's IR is designed for easier optimizations than a traditional IR like HIR [8]. Sea of nodes IR is in SSA form and requires an additional instruction scheduling. The server compiler IR is based on the concept of *floating* nodes. While more conventional compilers like, e.g., the client compiler use the concept of a CFG with a list of instructions in each basic block, in the sea of nodes approach control and data dependencies have the same structure and implementation. Where CFG IR refers to control flow and instructions inside a basic block to data flow, the sea of nodes IR represents them both as nodes. Nodes "float" through the IR. A node has no fixed position in the control flow of a method until a last *scheduling* step. The scheduling assigns each node to a fixed position in a basic block. Instead of basic blocks, sea of nodes IR uses so called *region* nodes that are generated where control flow of several predecessors joins (see *Merge* nodes later in Graal IR).

In Section 2.2.1 we describe Graal IR in more detail. Graal IR merges the concepts of the client and the server compiler. Graal uses CFG nodes and floating nodes in the IR.

2.2 Graal

For the implementation of Graal AOT JS we use the Graal compiler [47], an aggressively optimizing Java JIT compiler written in Java itself. Graal performs all standard compiler optimizations such as method inlining, global value numbering, constant folding, conditional elimination, strength reduction, and partial escape analysis. The high-level optimizations improve the quality of the generated JavaScript source code.

2.2.1 Graal IR

Graal AOT JS generates JavaScript code from Graal's high level intermediate representation. The intermediate representation of Graal [13, 14] is a directed graph in static single assignment (SSA) form [11]. Each IR node produces at most one value. To represent data flow, a node has *input* edges pointing to the nodes that produce its operands. To represent control flow, a node has *successor* edges pointing to its successors. In summary, the IR graph is a superposition of two directed graphs: the data-flow graph and the control-flow graph. Note that the two kinds of edges point in opposite

directions. Input edges go up, from the user of a value to the definition of a value. Control flow edges go down, from one control flow node to the next one. Nodes that are not necessarily fixed to a certain point in control flow, as they, e.g., do not have a side effect, may be floating. A floating node is a node which "floats" through the IR during compilation until a final scheduling step determines its position in the resulting program. A floating nodes position in the scheduling list of instructions is only dependent of its usages. For detailed information about scheduling we refer to [8] and for Graal IR to [13].

2.2.2 Graal Tiers

During compilation from Java to JavaScript Graal AOT JS re-uses large parts of Graal's compilation pipeline. Graal has several tiers each being more platform dependent than the previous one. The IR after the high-tier, after the mid-tier and after the low-tier. Figure 2.2 shows the different tiers of the Graal compiler in context of the compilation pipeline.

High Tier: The high tier is responsible for applying standard Java optimizations like constant folding, global value numbering, etc.

IR during the high tier is completely platform independent and represents bytecode semantics.

Mid Tier: The mid tier applies memory optimizations and prepares the graph for the runtime by adding meta data needed by the GC and the deoptimizer.

Low Tier: The low tier prepares the IR for LIR generation.

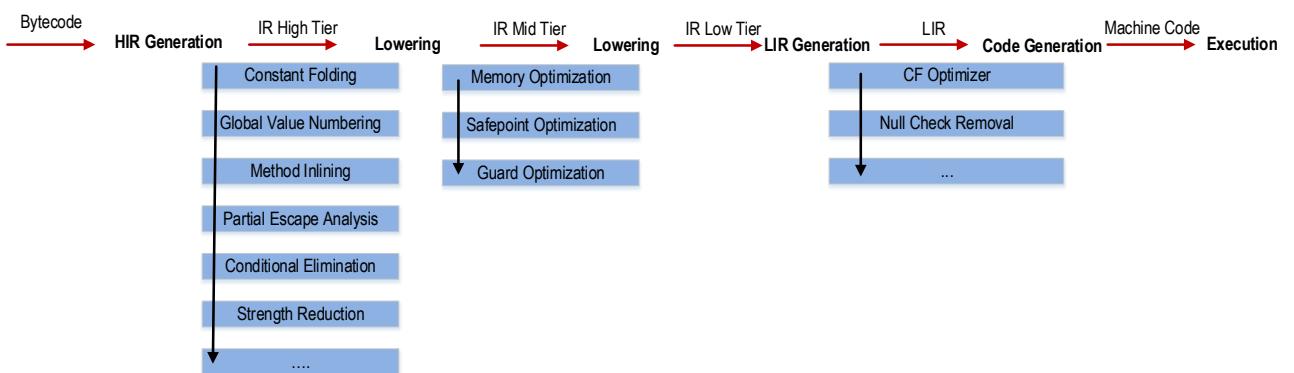


Figure 2.2: Graal Compilation Tiers

After the three phases the backend constructs the low-level intermediate representation (LIR) from HIR. LIR is a low-level intermediate representation that is similar to the client compiler’s LIR [32]. A linear scan register allocator [67] performs register allocation on the LIR. Then the compiler applies peephole optimizations and generates machine code.

Graal AOT JS cannot compile IR nodes that are dependent on a specific memory layout or target platform. Because JavaScript is a high level language that does not offer access to native memory. However, after the high-tier Graal IR mirrors the Java bytecodes parsed during graph building. Graal AOT JS can use this high-level IR for compilation as it is still platform independent and does not introduce concepts that have no high-level representation in JavaScript. The disadvantage of using a higher-level IR is that certain optimizations are not possible yet, e.g., array bounds-check elimination as presented in [68]. In the high-level IR an array load is represented as a single operation, not yet modeling the array bounds check.

2.3 Substrate Virtual Machine

Graal AOT JS builds on a specific extension of the Graal VM, the *Substrate VM*. The Substrate Virtual machine (SVM) is an embeddable Java VM built on-top of the Graal VM capable of compiling Graal AOT to native code. Graal AOT JS re-uses SVM as it provides elaborated static analysis technique as well as an optimization pipeline specific to AOT compilation.

2.3.1 Static Analysis

Compiling Java bytecodes ahead-of-time has one major drawback: code size. Java programs heavily use elaborate class libraries of the JDK. Thus, Java applications have numerous dependencies into the JDK, which produce a large call tree even for simple applications such as a trivial HelloWorld program. AOT compilation of a Java program with all its dependencies is therefore not feasible. As Java applications only use portions of the imported classes, we need to remove unused methods and types. This can be achieved by using a closed world assumption. For the AOT compilation to native code, SVM features static analysis technique based on a closed world assumption. Internally, this static analysis is an extended context-sensitive points-to analysis [55]. The reachable world is deduced to be the call graph and all potentially read and written fields [59]. For a given *entry point method* SVM’s static analysis iteratively processes all transitively reachable types, methods and fields that are necessary to execute the code of the entry point method. In this process all required types and their fields and methods are identified. The built call graph enables SVM to apply certain optimizations like, e.g., to treat classes without subclasses as final.

Graal AOT JS leverages the static analysis provided by SVM to reduce the size of the generated JavaScript code by removing unused elements.

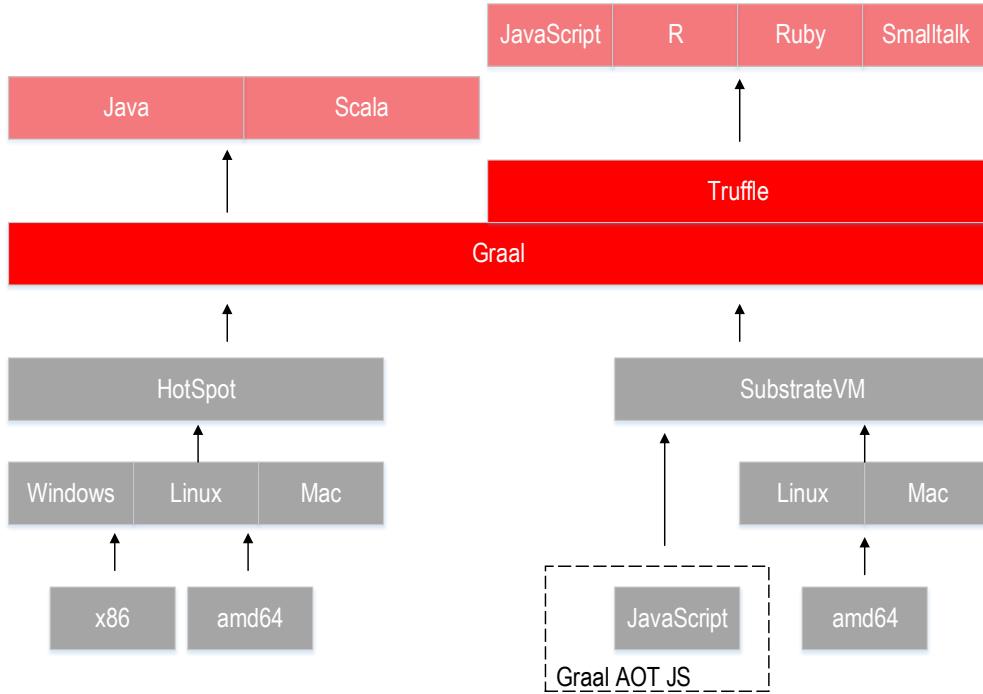


Figure 2.3: Graal Eco System

Figure 2.3 shows the big picture of the Graal eco system including the Graal compiler, with the self optimizing AST interpreter framework Truffle [69, 70] on-top. Truffle uses Graal for runtime compilation. On-top of Truffle there are several language implementations. Graal itself is a JIT compiler for all languages compiling to bytecode like, e.g., Java and Scala. Graal builds ontop the HotSpot VM. HotSpot runs on all standard platforms and operating systems. SVM extends Graal with static analysis technique to AOT compile Graal and Truffle to native code. SVM currently only supports Linux and Darwin (MAC). Graal AOT JS extends SVM with a novel backend generating JavaScript code instead of native code. Graal AOT JS is not only a backend, it performs several additional optimizations and also adds new interfaces to Graal and SVM. Chapter 3 goes more into detail about the interfaces between Graal, SVM and AOT JS.

Chapter 3

Graal AOT JS Compiler

This chapter explains the compilation pipeline that transforms Java to JavaScript. It gives an overview of the entire system including Graal, SVM and Graal AOT JS. In the first part the interfaces of the compiler with SVM and Graal are explained. The interfaces to SVM and Graal form the frontend. Then, the Graal AOT JS backend is explained. The main system steps introduced in this chapter being, control flow reconstruction and code generation, are explained in full detail in subsequent chapters.

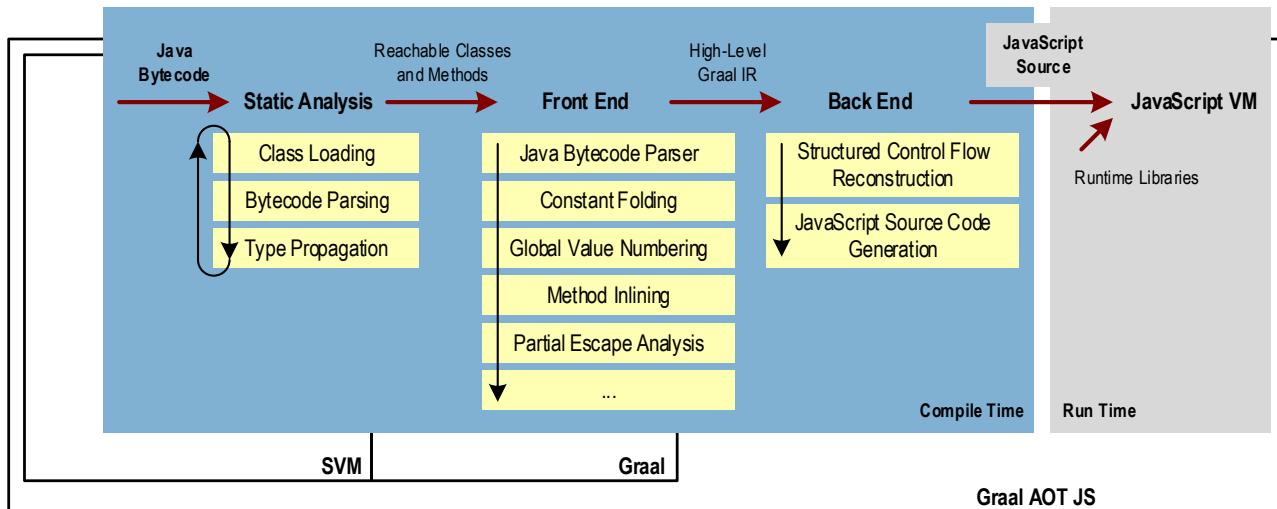


Figure 3.1: Graal AOT JS System Overview

Figure 3.1 shows the compilation pipeline of Graal AOT JS as well as which step of the pipeline belongs to which part of the system. The figure is divided into the two major parts of the compiler. The left hand side shows the AOT compilation of Java to JavaScript and the generation of JavaScript code. The first part happens at compile time. The right hand side denotes the final standalone JavaScript application for deployment. The compiler combines the generated code with a set of

runtime libraries that are needed for the execution of the code at runtime. Graal AOT JS generates a standalone JavaScript file that can be executed on any JavaScript VM without additional dependencies at runtime.

The class loading and the static analysis steps of the pipeline are provided by SVM. Subsequently Graal AOT JS leverages the high level optimizations of Graal and applies them on the IR. The final JavaScript code generation step is a novel implementation and runs ontop of SVM and Graal.

3.1 Frontend

Graal AOT JS can be divided into the frontend and the backend. The frontend does not contain novel implementations it reuses the static analysis technique provided by SVM and applies all optimizations on the IR Graal offers. The following list describes the steps of the compilation pipeline in the frontend:

Class loading Graal AOT JS loads the classes of the source application. It collects meta-information on them and invokes all static initializers.

Static analysis The static analysis phase uses Graal to build the IR for the methods encountered during analysis. The analysis parses every transitively reachable method and builds the IR for it. Those parts of the class path that were not discovered to be reachable can be excluded from compilation. Discovering classes leads to class loading, i.e., the first two steps are executed until a fixpoint is reached and no more new classes are discovered.

Graal optimizations Graal AOT JS applies standard compiler optimizations offered by Graal to the IR. Optimizations include global value Numbering [8], constant folding, strength reduction [3], conditional elimination [57], method inlining, and partial escape analysis and scalar replacement [58]. Graal AOT JS can leverage these sophisticated compiler optimizations without any additional implementation effort. The performance impact of some of these optimizations is discussed in Chapter 7.

3.2 Backend

The backend of Graal AOT JS, as seen in Figure 3.1, is a novel implementation. The Graal JIT compiler targets several existing CPU architectures. There exist several backends in the compiler including a mature *amd64* backend, a *Sparc* backend and an experimental *x86* backend. Graal contains an API for the implementation of native compiler backends for different architectures. However it is not possible to implement a backend generating high level language code with interfaces defined for machine code generation. Because Graal is no longer platform independent after the high tier. JavaScript does not allow for native memory manipulation, therefore all operations must be high-level language statements. This prohibits Graal AOT JS to use a lower level of representation of Graal IR. Therefore Graal AOT JS defines a new generic type of a compiler backend suitable for the generation of high level language code. Graal AOT JS implements this generic high level language backend for the code generation of JavaScript code. The following list describes the steps of the compiler in the generic high level language backend:

Graph canonicalization to structured control flow A set of control flow transformations rewrites certain structures in the IR to produce structured control flow.

Control flow reconstruction optimization The compiler performs a special analysis phase called *graph tagging*. Graph tagging is a novel control flow reconstruction approach. Graph tagging analyses the CFG in order to find structured control flow. It applies, in-place, structural rewritings to the IR, to generate structured control flow if possible.

Code generation The code generator of Graal AOT JS iteratively processes all types. For each type, it iterates all methods and generates code for them. Code generation for methods iterates the CFG in a semi depth-first fashion, using control flow information from the graph tagging phase for code generation of high level language control flow instructions.

3.3 Architecture

Figure 3.2 shows the most important components and modules of the Graal AOT JS compiler. Graal AOT JS extends the SVM modules for loading the classes of the entry point methods, and uses Graal to parse their bytecodes and start building the type tree. The `CompileQueue` component is responsible for compiling all Java types and methods to native code. Graal AOT JS extends this module and overwrites the compilation process. Compilation is performed by a component called *JSImageGeneration*. It will generate for each type the code for the type definition, then iterate

each method and for each method reconstruct control flow and traverse the CFG and generate code. Constant and static objects are recursively inspected and registered for compilation. Code generation instantiates lowerables for the IR nodes which then effectively generate the JavaScript code for each operation.

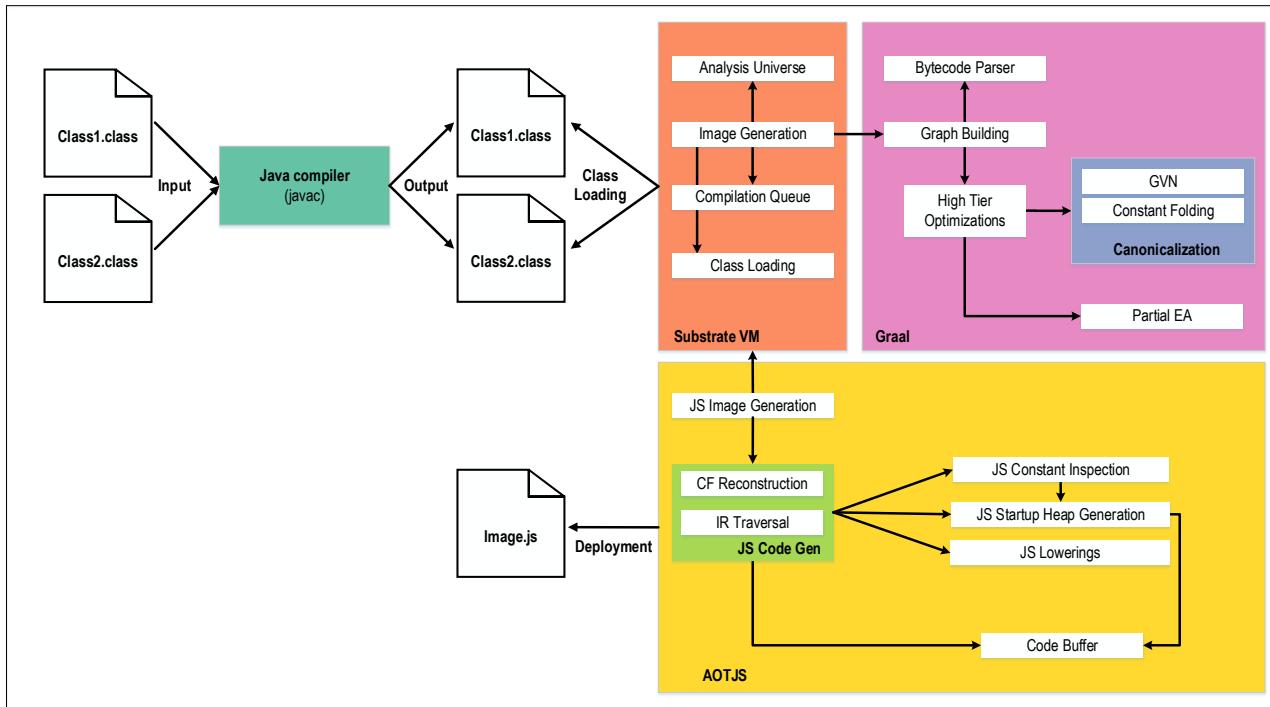


Figure 3.2: Graal AOT JS System Architecture Overview

3.3.1 Components

The following list describes the most important components of the Graal AOT JS compiler as seen in Figure 3.2:

JSImageGeneration This component is the main entry point to the compiler. It is the high level component that combines SVM with Graal AOT JS. It invokes the static analysis of SVM and then the code generation of Graal AOT JS.

JSCodeGen The JSCodeGen component drives the entire compilation process. It invokes compilation for types and methods, and combines the generated JavaScript code with the runtime support libraries. For the compilation of methods it invokes control flow reconstruction and constant resolve. Code generation is invoked by invoking the *IR traversal* component which effectively performs code generation.

IR Traversal The *IR Traversal* component performs the IR traversal for code generation. It uses the control flow reconstruction information to traverse the IR recursively and generate code for each basic block. Code generation for each basic block iterates all instructions of a basic block and invokes code generation for each Graal IR node. The code generation mechanism based on the class of a node is explained in Section 3.3.2.

Constant Inspection Constant and static data [35] is used during high tier optimizations that are based on values like, e.g., conditional elimination. This data must be analyzed and compiled to JavaScript, as referencing code uses it at runtime. Graal AOT JS writes constant and static data to a pre-initialized JavaScript heap, for details we refer to Chapter 5.

Code Buffer The code buffer stores the bytestream of JavaScript code during compilation.

3.3.2 Node Lowering

The *node lowering* mechanism of Graal AOT JS determines which code is generated for which IR node. To avoid changes upstream in Graal and to have a modular and extensible architecture of the code generation for a high level language, Graal AOT JS defines a class called `LowerableNode`. A `LowerableNode` wraps a Graal IR Node. Every Graal IR node, for which code must be generated, is directly represented by Graal AOT JS as a subclass of the `LowerableNode` class. Each subclass must implement the functionality for code generation. During code generation Graal AOT JS traverses the CFG of each method. Scheduling gives a distinct order of all nodes inside a basic block. Code generation sequentially iterates over all instructions of a basic block and generates for each Graal IR node, wrapped in a `LowerableNode`, JavaScript code.

3.4 Limitations

AOT compilation of Java bytecode to JavaScript comes with a set of limitations. There are certain concepts of Java that cannot be mapped to JavaScript with AOT compilation:

- **Dynamic Class Loading:** Graal AOT JS cannot support dynamic class loading as this would require runtime compilation. Runtime compilation would require a compiler like, e.g., Graal itself to be compiled to JavaScript. Currently, runtime compilation is not supported, but this might change in the future in order to support the self-optimizing AST interpreter framework *Truffle* [69], on-top of Graal.

- **Reflection:** Java has an elaborate reflection mechanism enabling programmers to examine or modify the behavior of applications at runtime. AOT support for the Java reflection API is limited for the same reasons that dynamic class loading is not supported.
- **Multithreading:** Multithreaded applications introduce two problems with the presented approach. Our current static analysis is not capable of analyzing multithreaded applications appropriately. Another problem is JavaScript's inherent lack of a real concurrency model. JavaScript has WebWorkers [44] which offer the capability to execute code in a different thread. However this model does not feature shared memory. Data is exchanged via message passing and callbacks. Java and JavaScript do not share a common semantic notion of concurrency, thus this feature of Java is not supported by Graal AOT JS.
- **Synchronous APIs:** JavaScript does not support synchronous I/O, whereas Java does. It is not possible to map synchronous Java APIs to JavaScript with AOT compilation. Such APIs like, e.g., the Java Socket API, must be re-written to work asynchronously. Currently, this is not supported by Graal AOT JS.

Chapter 4

Control Flow Reconstruction

This chapter presents the structured control flow reconstruction that Graal AOT JS applies during Java bytecode to JavaScript translation. This reconstruction happens after Graal AOT JS applied the high level optimizations on the IR. It is done in the compiler backend. Control flow reconstruction is not only crucial for runtime performance of the generated JavaScript code, it also improves readability and code size. Therefore this chapter specifies the properties of structured control flow and shows how unstructured control flow emerges from the compilation of Java bytecode. It presents Graal AOT JS's approach for the reconstruction of structured control flow.

Graal IR uses a directed graph for modeling control flow. Each node in the CFG has *predecessor* and *successor* nodes. For compilation to a high level language Graal AOT JS must establish a mapping from Graal IR to the high level language control flow constructs of JavaScript. Figure 4.1 shows an example of a simple Java function summing up values until an upper bound compiled with Graal AOT JS to JavaScript. The sub figure in the middle shows simplified Graal IR for the function. The IR shows the loop (`LoopBeginNode`), the end of the loop, the exit and the return. As Graal IR is in SSA, we see that the variables `i` and `sum` from the Java source code are represented as *Phi-Nodes*. For each Graal IR node that has more than one predecessor or successor, e.g., in Figure 4.1 the `IfNode`, it is necessary to find a corresponding JavaScript high level language control flow instruction. Figure 4.2 illustrates the problem. There is one node with more than one successor, the `IfNode`, and one node with more than one predecessor, the `MergeNode`. Every node having more than one CFG successor must be represented with an open lexical block in JavaScript (denoted by an opening brace). For every node that has more than one CFG predecessor it is, in a structured CF case, necessary to close a block. Graal AOT JS needs to be able to establish such a mapping. However finding the mapping between

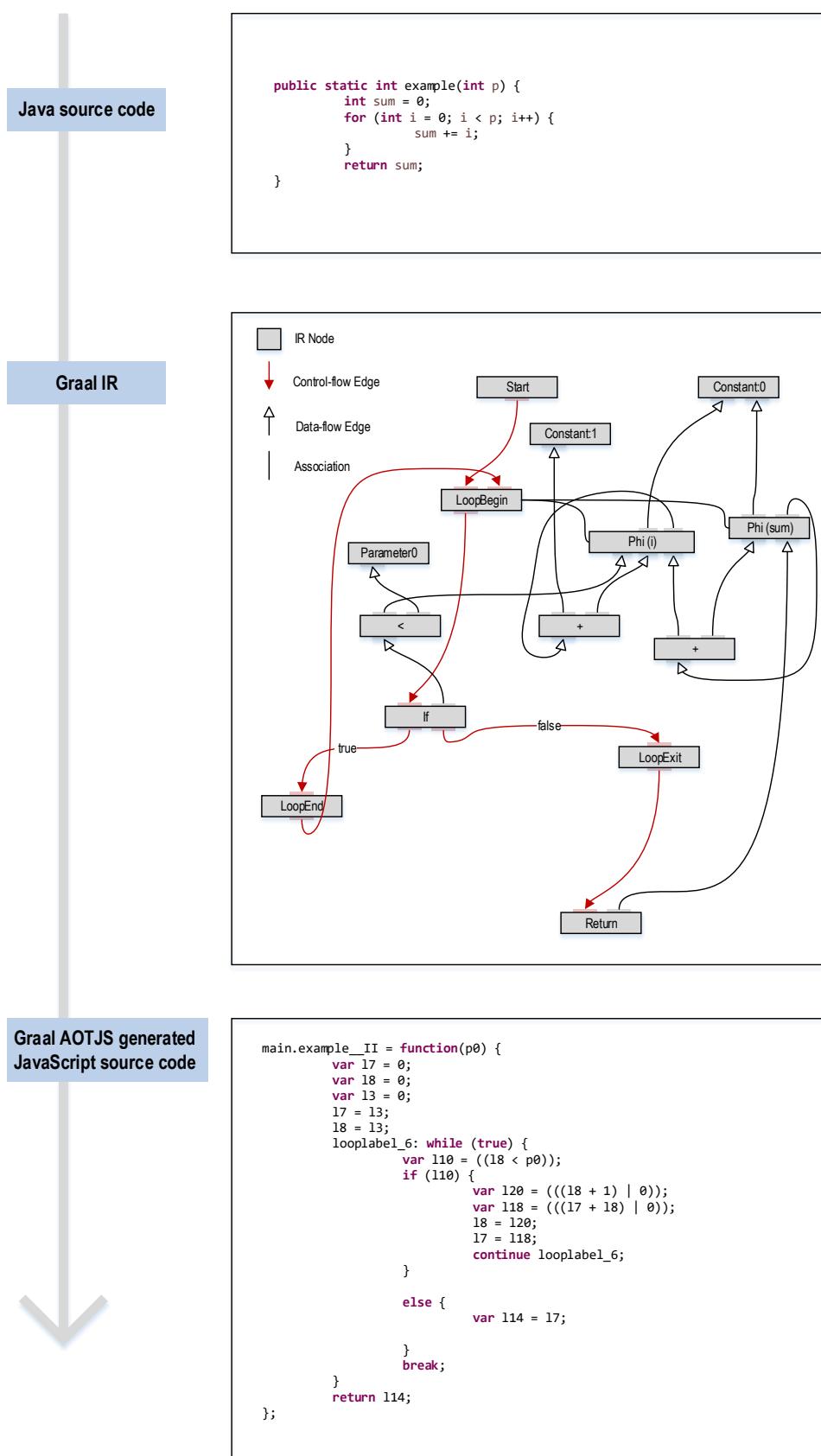


Figure 4.1: Java to JavaScript code generation example.

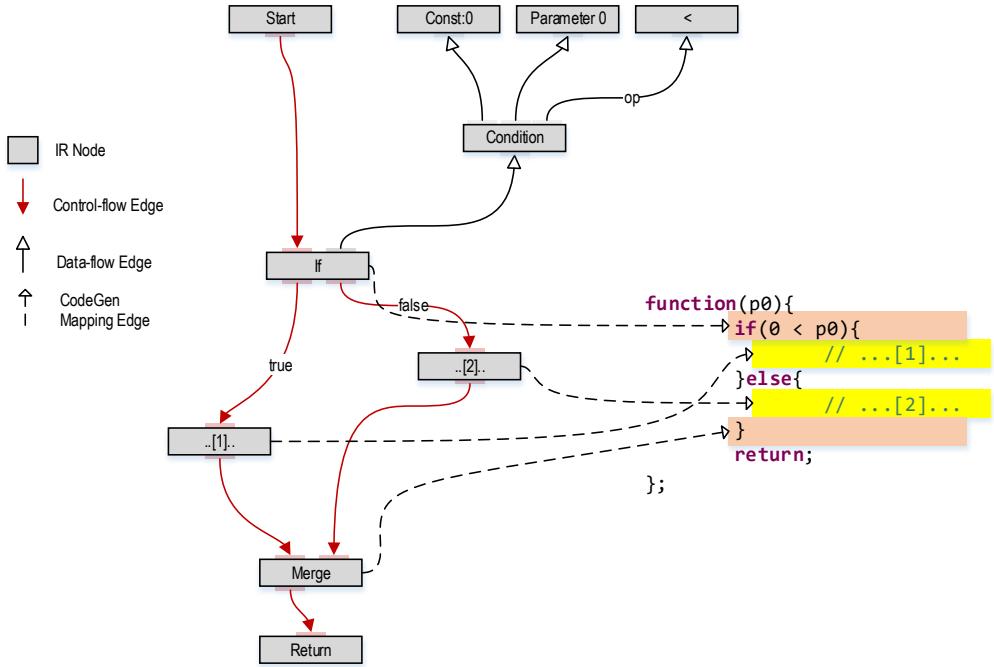


Figure 4.2: Nodes to braces matching.

CFG nodes and blocks in the generated, decompiled code is not always trivial. The mapping between CFG structures and blocks is in most cases very complex and cannot be decided ad hoc. Therefore, a mapping heuristic needs to be found, which satisfies different functional and quality attributes:

- **Determinism:** The mapping heuristic must be deterministic. For cases where a semantically and syntactically correct mapping cannot be guaranteed, code generation must fall back to a deterministic result that is valid in its execution semantic. Such a fallback is then typically suboptimal in its usage of control flow instructions, but preserves semantics of the program. See Section 4.2.2 for details about Graal AOT JS' fallback mechanism.
- **Run-time Performance:** The mapping heuristic must meet certain run-time performance requirements. Offline transformation of Java to JavaScript does not demand high compilation speed. However, especially for bigger inputs (multiple ten thousand lines of code), the mapping should perform linearly over the number of nodes in the IR. This way the compiler can guarantee reasonable compilation times.

The deterministic mapping of IR graphs to high-level language code is based on the property of *structured control flow* which builds on the paradigm of structured programming established in the 1960s.

4.1 Structured Control Flow

Structured control flow is denoted by a set of high level language control flow statements that are considered to transfer control flow in a structured way. In this context a "structured way" is defined by the means of the paradigm of structured programming.

The following list of high level language control flow statements forms a *generic* set of control flow building blocks. The statements are *Sequence*, *Continuation* (unconditional jump), *Selection* (conditional jump) and *Iteration* (loops). These statements are the building blocks for every other high level language control flow statement, as presented in Figure 4.3.

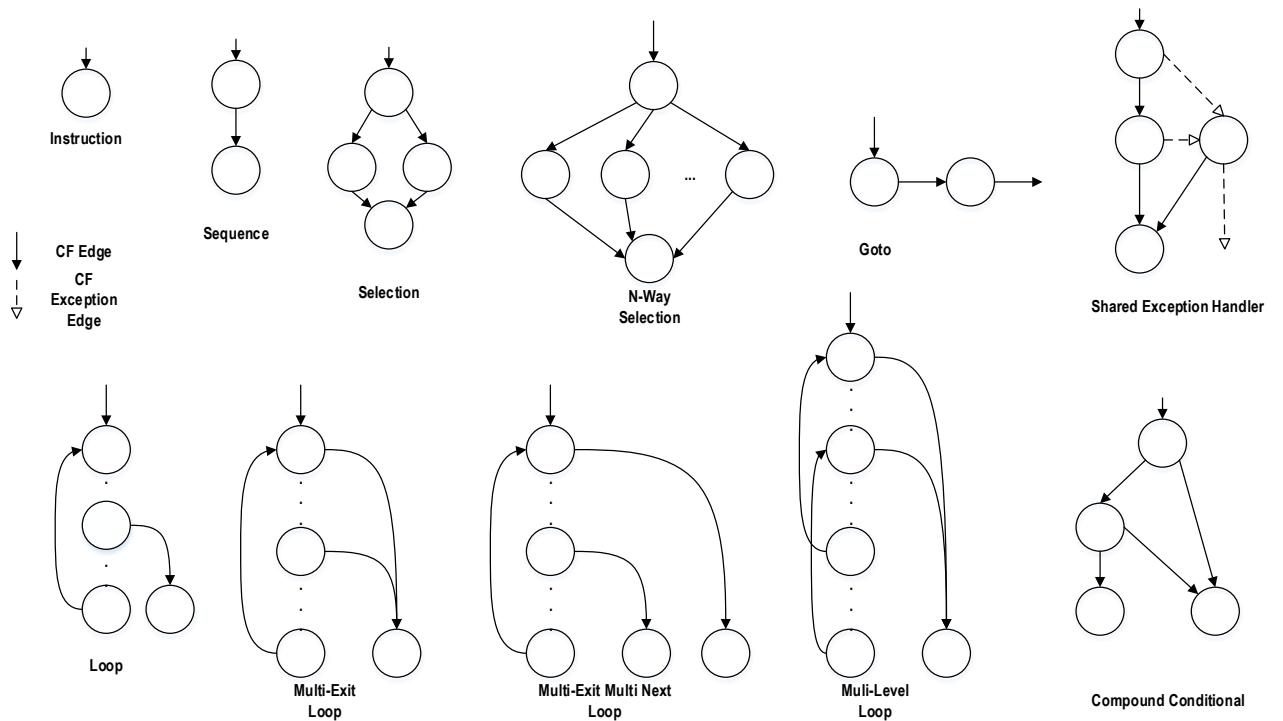


Figure 4.3: Generic CFG Patterns (Modified: originally presented by [6]).

Figure 4.3 shows the general high level language control flow patterns as flowcharts. The list was initially published in [6], however, for the purpose of cross compiling Java to JavaScript, we modified the existing patterns to fit to Graal's IR. We removed patterns we do not need, and added the *Shared exception handler* pattern, as it is a contribution over [6]. This list of patterns does not claim to be complete, it is a foundation for various high level language control flow constructs.

Structured control flow only transfers control flow in a program by the usage of sequence, selection and iteration. The usage of continuations (e.g. the `goto` statement) in a program may transfer control flow in a way it is not possible to do via the other generic control flow instructions. Thus, unstructured control flow is denoted by the usage of continuations which generates control flow that is not expressible without continuations. However, certain high level language constructs like, e.g., the `break` of a labeled block, also introduce unstructured control flow. Such control flow constructs offer a semi-structured way of transferring control flow to points in the program that would not be supported by structured control flow.

The Java language supports several patterns that generate unstructured flow charts. E.g., the break of an arbitrary labeled block cannot be re-transformed to a structured representation after compilation to bytecode, as the meta-information about the begin of the labeled block is lost. Figure 4.4 illustrates such an example, is shows the original Java source code, and the Graal IR graph after parsing the bytecodes and building the IR. After the compilation to Java bytecode, the information about the three different labeled blocks, as seen on the right in Figure 4.4, is completely lost and the `if` instruction in the false branch of the outermost `if` simply performs a `goto` operation to the `true` and `false` branch of if instruction in the true branch of the outermost `if`.

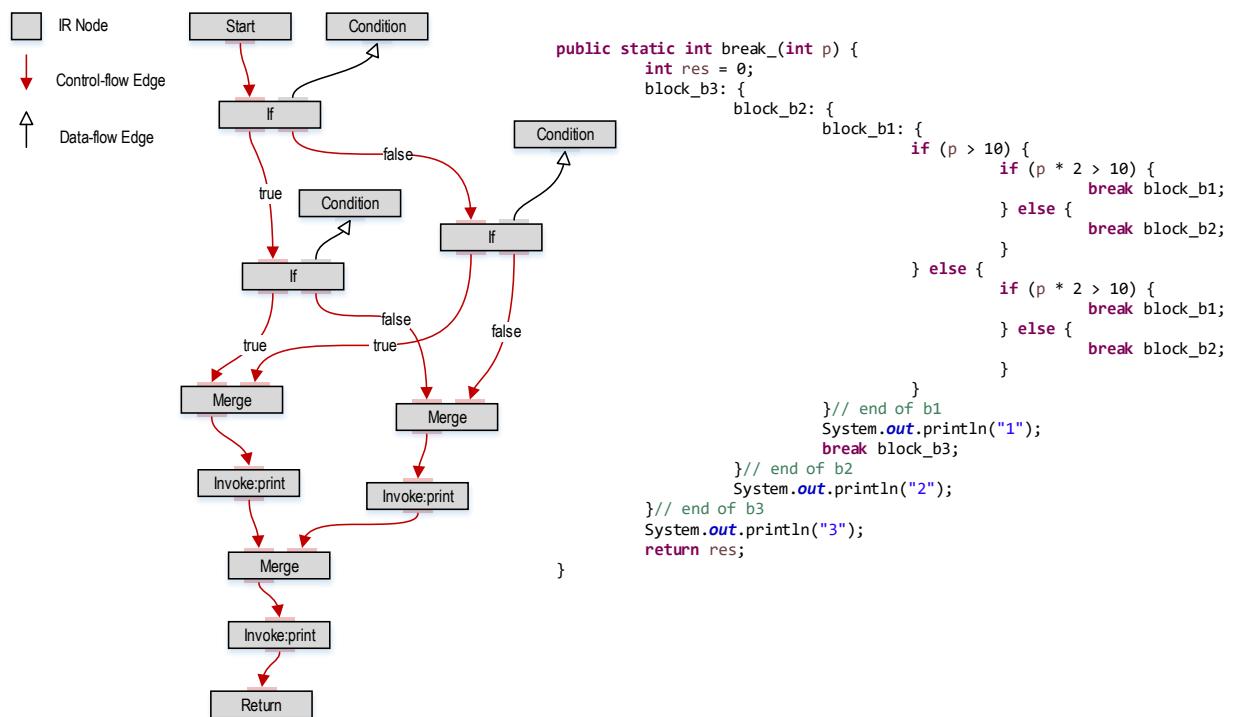


Figure 4.4: Unstructured If-Goto with labeled blocks

In the following we present a detailed description of the control flow patterns from Figure 4.3. We especially focus of the property of structuredness of each control flow pattern. Additionally, we directly refer to the Java representation for each pattern. In the following examples of control flow patterns we explicitly omit the Java bytecode representation as it is tedious to read and does not contribute to the illustrations of the examples. The CFG graphs presented follow the definition of Graal IR. However, they are Graal agnostic in their representation of the underlying bytecode CFG.

- **Instruction:** The single instruction is the simplest possible control flow element. An instruction is the smallest building block of a control flow graph. A basic block consists of a list of instructions with just one entry and one exit, the first and last instruction of the basic block.
- **Sequence:** A sequence is the sequential concatenation of instructions. A basic block itself is a sequence of instructions.
- **Selection:** A selection is the classical if-then or if-then-else construct. Java offers if selections with and without an else branch. Control is, depending on the evaluation of the condition, transferred to either the true or the false branch. Figure 4.5 shows Graal IR for a simple function performing an if instruction.

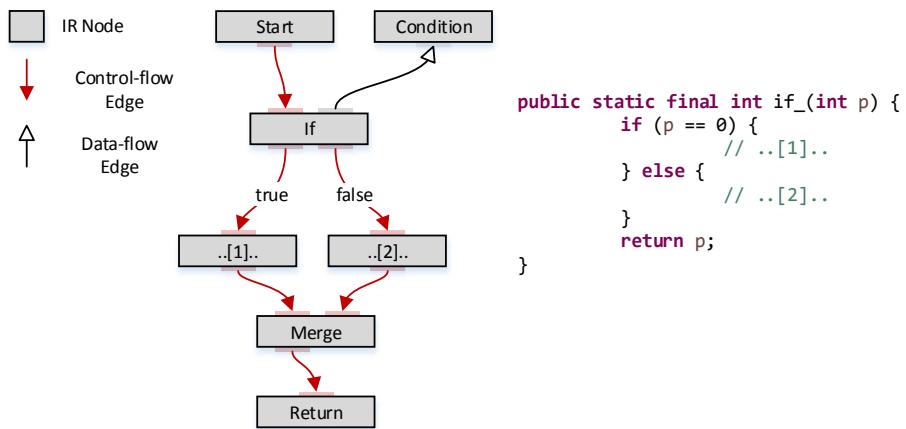


Figure 4.5: If Graal IR

- **N-Way Selection:** The N-Way selection is a classical generalization of the two-way selection (if). Typically this construct is called switch. Figure 4.6 shows Graal IR for a simple function performing a switch instruction. Modern languages like Java or C# also support a special form of an unstructured case statement. Java requires a case branch to jump to the MergeNode with a break instruction, like shown in the source code of Figure 4.6. However, the Java language also allows a control flow transfer known as a "fall-through" case. This happens if the break statement is missing as the last instruction of the last basic block in a case branch. Control flow is then no longer transferred to the merge node, but to the first instruction of the

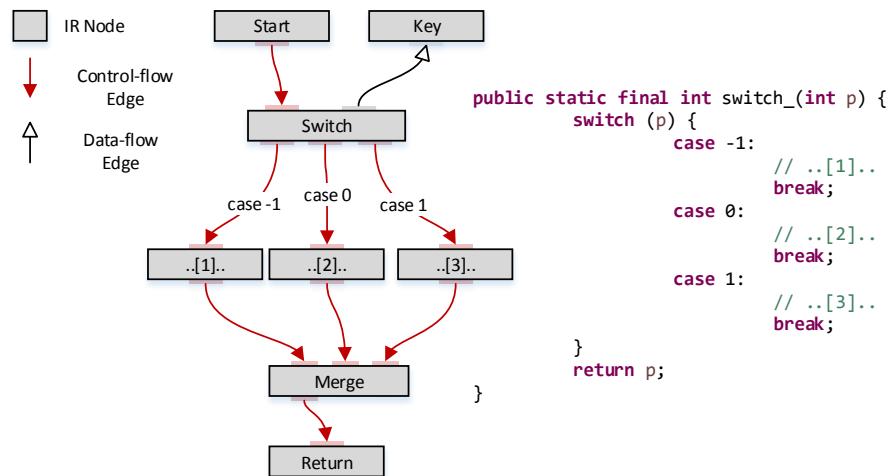


Figure 4.6: Switch Graal IR

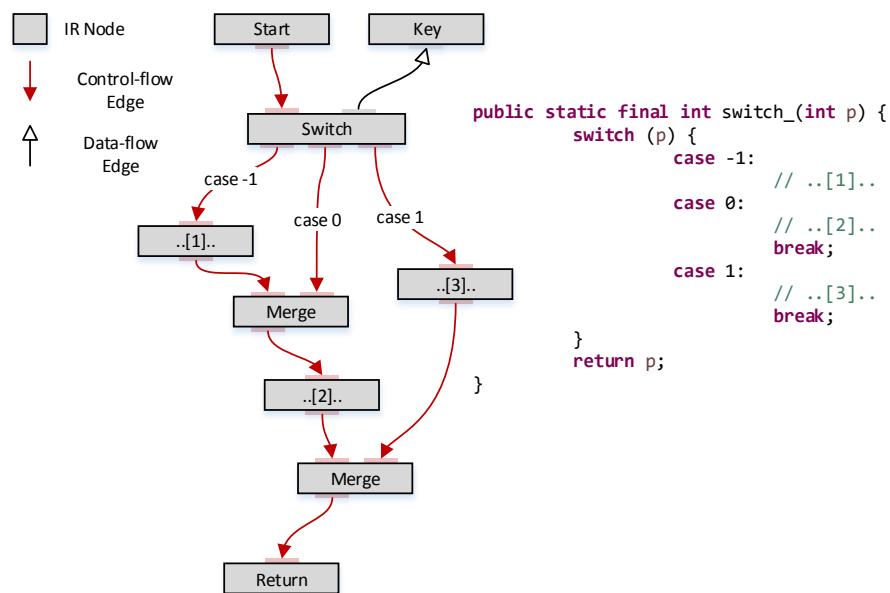


Figure 4.7: Fall through Switch Graal IR

```

1 int structuredGoto(int p){
2     int sum = 0;
3     int i = 0;
4     label_header:
5         if (i >= p){
6             goto end;
7         } // else enter body
8         sum += i;
9         i++;
10        goto label_header;
11    label_end:
12        return sum;
13 }
```

Listing 4.1: Structured Goto C

subsequent `case` branch. Figure 4.7 shows an example of the resulting CFG with a fall-trough `case` from the first to the second `case`. These kind of fall-troughs always generate unstructured control flow, as they act like a `goto` instruction. If we take a look at Figure 4.7 we see that the switch construct now has two different merge blocks, one where the cases `-1` and `0` merge and one where the end of case `-1/0` and the branch of case `1` merge. Every fall-trough generates an additional merge, with the subsequent branch, introducing a large set of merges. However, it is no longer deterministically known if a merge belonged to the switch statement, or a structure which was before the switch in the CFG. To know which merge belonged to which case, an elaborate CFG analysis would be required. Graal AOT JS omits this kind of analysis during control flow reconstruction, as fall troughs are not very frequently used in general purpose Java programs. Switch fall troughs generate a bail out of the CFG reconstruction algorithm and code is then generated with Graal AOT JS' fallback mechanism presented in Section 4.2.2.

- **Goto:** The `goto` statement construct transfers control flow to an arbitrary position in a program. Java on the source level does not feature the `goto` keyword. However, different other statements can be used to fully emulate a `goto` statement.

The usage of the `goto` statement does not necessarily generate unstructured control flow. A `goto` can transfer control flow also to the points in a program that would be used by structured control flow instructions. The simplest example is a `while` loop without the loop header instruction, where the back edge jump is modeled via a `goto` instruction. Listing 4.1 illustrates the structured usage of `goto` statements.

- **Loop:** The loop construct represents the iteration in the list of generic control flow statement building blocks. There are many different types of loops. A non-exhaustive list of the most common ones, including all Java loop types, can be found below:
 - **Pre-Tested Loop:** Those kinds of loops are classical `while(condition)` or counting loops like, e.g., a `for` loop. Pre-tested loops perform the test of the loop condition before entering the body of the loop.
 - **Post-Tested Loop:** Post-tested loops move the check of the loop condition at the end of the loop body. The loop body is always entered once, even if the condition will fail in the first iteration.
 - **Endless- Loop:** Endless-loops are loops without conditions. They do not have nodes outside the loop in the CFG.

Figure 4.8 shows a simple example of Graal IR for a `for` (pre-tested) loop. Generally loops do not introduce unstructured control flow, as long as they only have one exit. In the following we explain loops that generate unstructured control flow.

Graal’s IR does not make a distinction between pre- and post-tested loops. Both kinds of loops can be transferred into-another.

Graal AOT JS generates code for loops always with a `while(true)` statement. The `break` and `continue` statements are used to end and exit a loop. This simplification of the code generation makes no semantic difference and removes the burden of analyzing the type of the loop.

- **Multi-Exit Loop:** Multi-Exit loops are loops which have multiple points in the body of the loop where the loop is exited. Many high level languages like Java or C offer the `break` instruction to exit a loop early. However, as long as the successor node of all loop exit nodes is the same, multiple exits do not introduce unstructured control flow. Figure 4.9 shows Graal IR for a simple loop with an additional exit.
- **Multi-Exit Multi-Next Loop:** Loops with multiple exits leading to different successor nodes or loops with multiple backward edges are unstructured [6]. Figure 4.10 shows an example of a multi-exit loop in Java. It shows Java code for a loop with multiple exits going to different successor nodes and the Graal IR after parsing the bytecodes. After the compilation to bytecodes there is no corresponding high-level representation for the given code snippet, except with the usage of labeled blocks. Code generation using labeled blocks requires one labeled block per

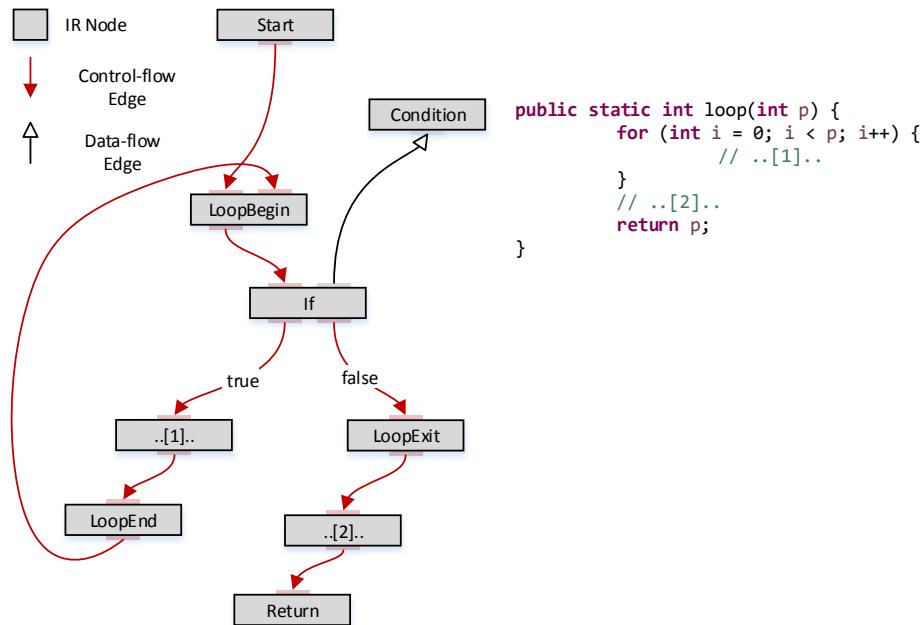


Figure 4.8: Loop Single Exit Graal IR

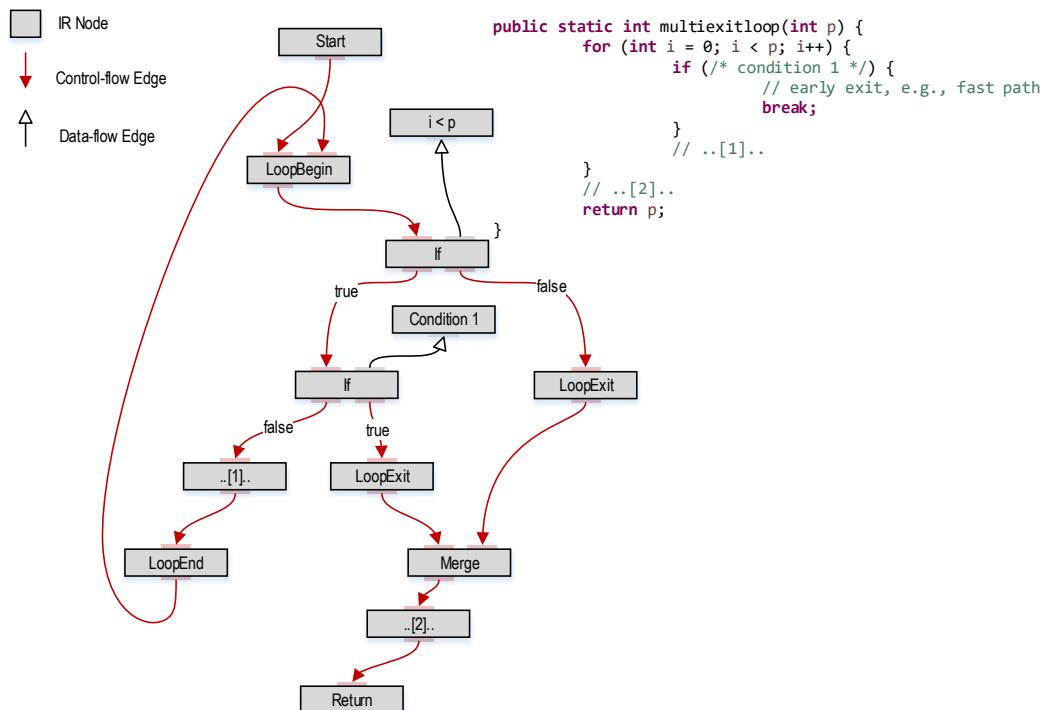


Figure 4.9: Loop Multi Exit Graal IR

loop exit. Graal AOT JS uses a different approach for the reconstruction of multi-exit loops which allows for an intuitive structuring of loop bodies. For details about Graal AOT JS's decompilation of loops we refer to Section 4.2.

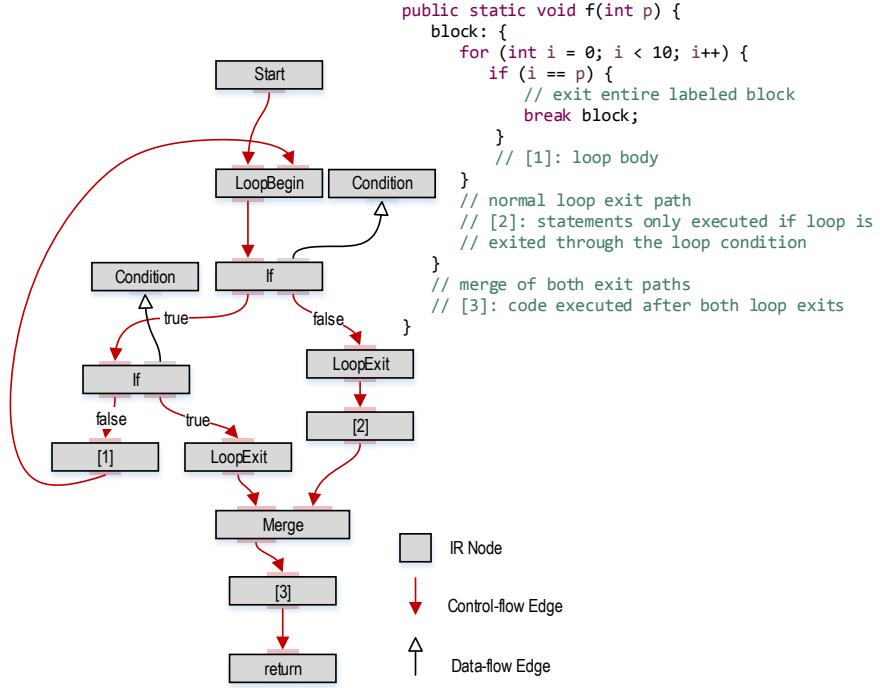


Figure 4.10: Multi-Exit Multi-Next Node Loop Graal IR

- **Multi-Level Loop:** Multi-Level Loops are generally no special form of loops, however, they can only occur when using the `continue` statement. The `continue` statement in Java can be used to immediately jump to a labeled loop header. For staged loops this enables a program to jump from a loop to a loop in an outer nesting level.

For decompilation of Java bytecode to JavaScript multi-level loops only impose one further restriction to the control flow reconstruction: A loop can be compiled to a structured piece of JavaScript code only if the entire control flow in the loop is structured *and* if all `continue` statements to outer loops go to loops that are completely structured themselves. These restrictions are necessary, as an outer loop might be unstructured due to any of the mentioned reasons. A loop one level deeper may be structured, but contains a back edge to the outer loop(s). In this case, the semantic of the `continue` statement depends on the used code generation policy for unstructured control flow, as initially referred to as the unstructured codegeneration *fallback*.

- **Shared Exception Handler:** Java has a language built in exception handler mechanism. In Java bytecode exception handlers are plain jump tables with byte code indexes [35]. Miecznikowski and Hendren [41] present the problems that arise from Java exception handler tables in detail.

The main problem of the exception handler tables are that they can be nested multiple levels deep and refer to each other. There is no high level information present in bytecode about the lexical try-blocks of Java source code.

Shared exception handlers introduce problems for the control flow reconstruction from Graal IR. Graal IR only supports one distinct unwind node per method. Unwind nodes are nodes denoting a Java `throw` operation, an operation unwinding the current stack frame to the caller and continue execution in the caller. Therefore, every node in a method potentially producing an exception has an exception edge to the unwind node, if a corresponding exception handler is missing. All exception edges merge before the unwind and generate unstructured control flow. An example for shared exception handlers can be seen in Figure 4.11. It shows the source code of shared exception handlers and Graal IR after the parsing of the bytecodes and the building of the IR. The exception handler from the Java source code catches exceptions of type `CustomException` for both invocations in the `try` block. Graal IR represents this with one exception handler that is reachable via the exceptional path from both invocations. Therefore the IR merges for both exceptions before the exception handler.

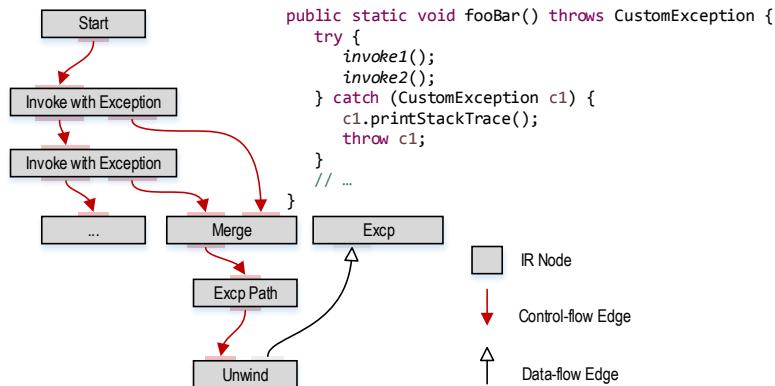


Figure 4.11: Shared Exception Handler Graal IR

- **Compound Conditional:**

Compound conditionals, often referred to as short circuit evaluation, are patterns in a CFG introduced by via the usage of the `&&` and `||` instructions. Figure 4.12 shows an example for a simple compound conditional in Java.

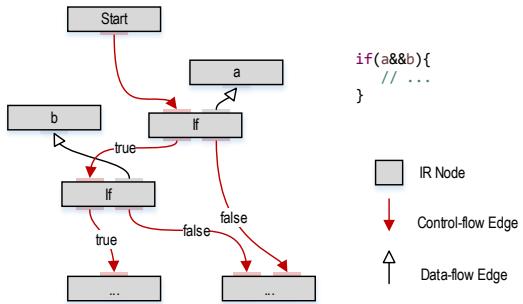


Figure 4.12: Compound Conditional Graal IR

4.2 Graph Tagging

This section introduces a novel structured control flow reconstruction algorithm for Java bytecode to JavaScript compilation. We refer to this structured control flow reconstruction heuristic as *graph tagging*. An arbitrary, reducible CFG is analyzed and structured sub-graphs are detected which denotes the *tag* operation. During the detection of structured sub-graphs, additional structural rewritings take place that ensure a structured graph. The presented heuristic is based on general control flow concepts and is not limited to our Graal IR.

Below the terms *walk*, *link* and *final path* are used. A walk is an ongoing or currently stalled bottom-up traversal of the CFG to find structured sub-graphs. A link is a sub-graph that has already been identified to be structured. During the walks previously established links are skipped, as they already contain structured control flow. Final paths are those walks that start at an instruction which has no successor in the CFG (except one reached via a back edge), e.g., `break`, `continue`, `throw` and `return`.

Figure 4.13 illustrates the concept of final paths. There are three paths in the CFG marked, each of which is a final path as it either ends on a return node, a unwind node or a loop end. The simplest example of a link can be seen in Figure 4.14 which shows explicitly the link from Figure 4.15.

4.2.1 Algorithm

The control flow tagging algorithm is applicable to deduce structured control flow, thus, to ease the process of analysis, Graal AOT JS applies a set of structural transformations prior to control flow analysis. For those graphs containing unstructured control flow or on which the tagging algorithm bails out, code generation uses the *block interpreter* as illustrated in figure 4.17.

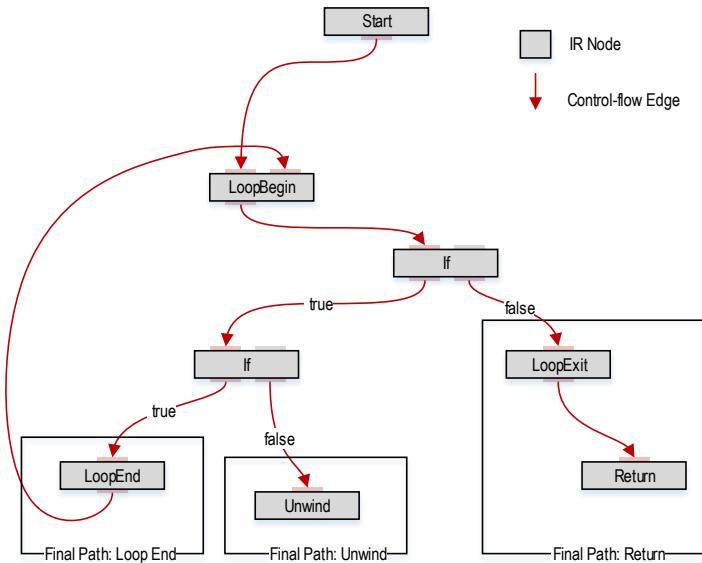


Figure 4.13: Final Paths

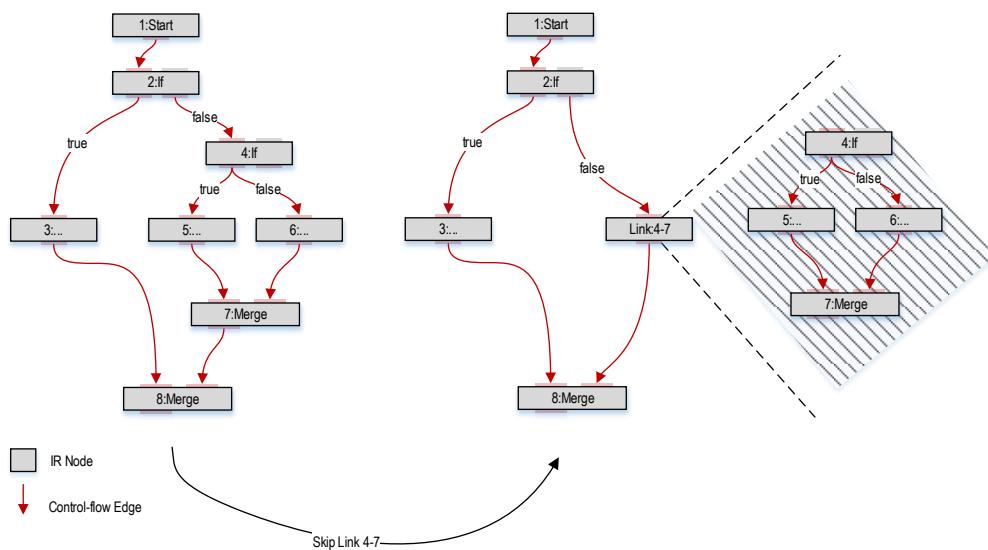


Figure 4.14: Tagging Algorithm Link

```

1 // list of all nodes
2 List nodes = ir.allNodes();
3 void tagGraph(){
4     foreach(Node n: nodes){
5         if(n isa Merge){
6             foreach(Node e:n.predecessors){
7                 walkBack(e, n, n);
8             }
9         } else if(isFinalNode(n)){
10            walkBack(n.predecessor, null, null);
11        }
12    }
13 }
14 void walkBack(Node curr, Node prev, Node start){
15     if(curr isa Merge){
16         if(isMergeOfLink(curr)) {
17             Node split = splitOfLinkAtMerge(curr);
18             // skip link - continue walk after the link
19             walkBack(split.predecessor, split, start);
20         } else{
21             // stall the walk
22             save(curr /*key*/, new Walk(curr, prev, start));
23         }
24     } else if(curr isa Split){
25         if(splitMustBeTagged(split)){
26             tagWalk(curr/*split*/, prev, start/*merge*/);
27         }
28         if(allTagged(curr)){
29             // create new link
30             createLink(curr);
31             // respawn walks stalled at the new link's merge
32             respawnSavedWalk(mergeOfLinkAtSplit(curr));
33         }
34     } else{
35         // arbitrary CFG node
36         walkBack(curr.predecessor, curr, start);
37     }
38 }
```

Listing 4.2: Control flow tagging algorithm.

Graph tagging is based on the following facts which have their roots in the definition of structured control flow:

- All upward paths from a merge node m lead to the same split node s . Similarly, all non-final downward paths from a split node s lead to the same merge node m . In both cases, there must not be other merge or split nodes between s and m except in links that are already known to be structured.
- Every final path can always be mapped to JavaScript code. A loop can always be exited early or continued and a return can always be emitted in JavaScript.
- Additionally to the two definitions, it is required that the number of predecessors of merge node m is less than or equal to the number of successors of the split node s . There might be fewer predecessors of m than successors of s , if top-down paths starting at s are final.

In our algorithm we perform the following steps:

1. From every merge node and from every final node traverse all incoming edges upwards until a split or another merge is encountered. Such a traversal is called a walk (In Listing 4.2 this operation is denoted by the method `walkBack(curr, prev, start)`).
2. If a merge is encountered during a walk, do the following:
 - If the merge is already associated with a split (`isMergeOfLink(merge)`), and is thus marked as being structured, skip the entire link and continue the walk at the split's predecessor.
 - If the merge is not associated with a split, save the walk in a map with the merge as the key, thus, stall the entire walk for possible later continuation (`save(key, Walk(curr, prev, -start))`).
3. If a split is encountered during a walk, do the following:
 - If the split is already part of a link, i.e., if it has already been encountered during a different walk, this can only happen with unstructured control flow, thus, abort the walk.
 - If the split is not yet part of a link, store this walk at the split node. Once the number of stored walks for a split node equals the number of successors for this split node, check, if all these walks started at the same merge node. If so, consider the sub-graph between the

split and the merge as a link and consider it as structured. As the sub-graph between the merge and the split (and further potential links) is tagged, all walks that were stalled at the particular merge are restarted.

4. For all other nodes that are encountered, continue the walk at the predecessor.

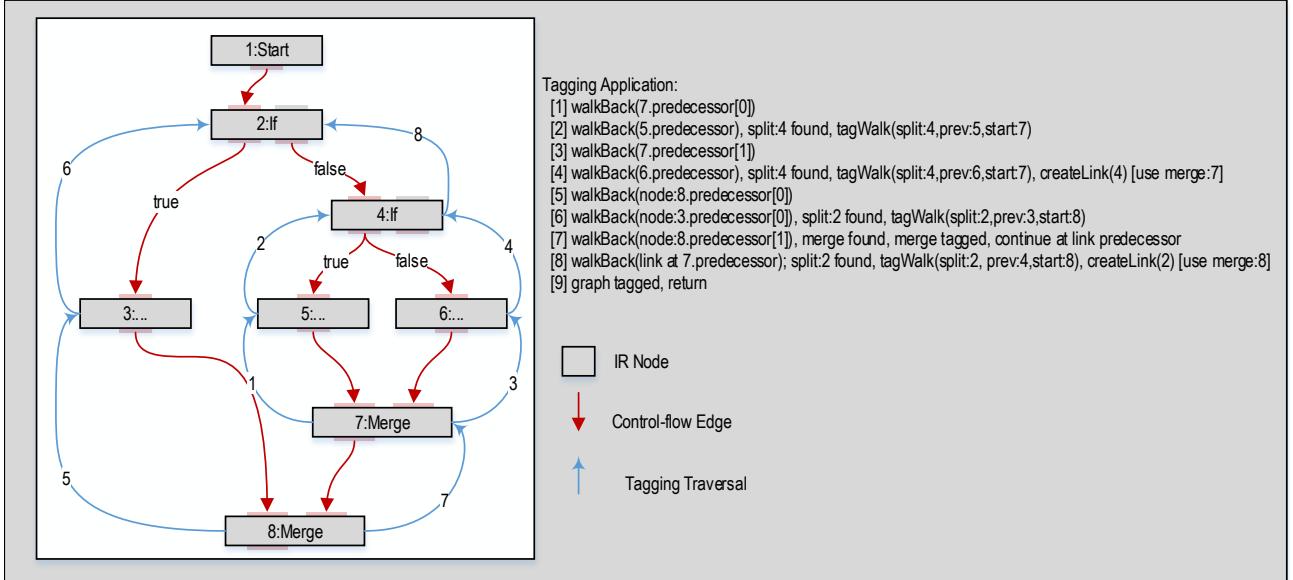


Figure 4.15: Graal AOT Tagging Algorithm Application

Figure 4.15 illustrates the tagging algorithm with a trivial example.

4.2.2 Block Interpreter

To model cases where graph tagging is not capable of generating structured control flow, or for cases where analysis bails out, we use a generic solution for modeling and transforming arbitrary control flow during code generation. It is based on an approach for the removal of `goto` statements presented by Erosa and Hendren [18]. In the domain of control flow obfuscation the approach is known as *control flow flattening* [33, 65]. Control flow is expressed by an endless loop dispatching CFG successors with a `switch` statement. Figure 4.16 illustrates the code resulting from a compound condition of the form `a && b`. Every case in the `switch` statement is a basic block of the CFG. This generic pattern, although elegant and easy to adapt, has major disadvantages. The performance is between $2x$ and $> 10x$ slower than a structured representation of control flow. In order to improve the performance and the readability of the code, it needs to be restructured.

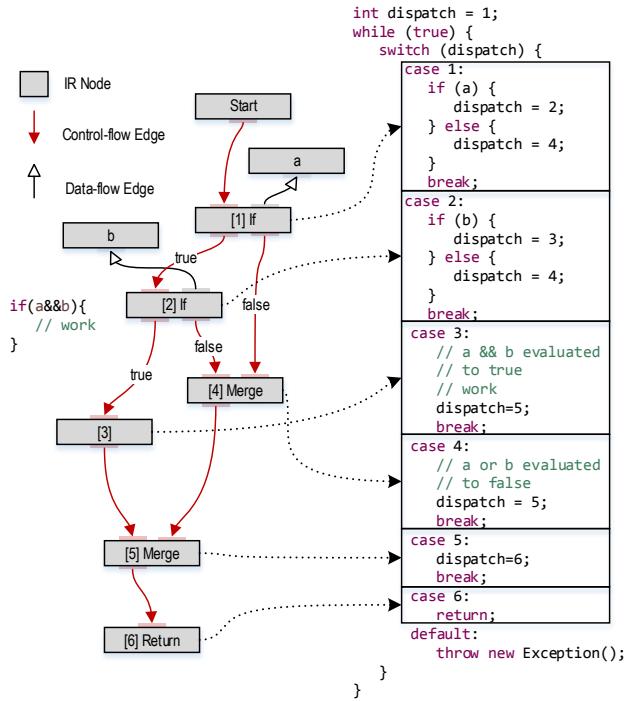


Figure 4.16: Control flow graph block interpreter.

4.3 Reconstruction Pipeline

This section covers all phases that are applied on Graal IR prior to graph tagging in order to transform as much unstructured control flow to structured one. Figure 4.17 gives an overview over the big picture of the entire control flow reconstruction process including all steps in the pipeline. After bytecode parsing, graph building, inlining, canonicalization and partial escape analysis which all happens in the front end, a set of phases are applied on the IR to ease the later step of graph tagging. Each phase applied prior to graph tagging either removes patterns of unstructured control flow or canonicalizes the graph to a simpler representation.

In the following we use a running example to illustrate the impact of major transformation steps on the IR and the resulting generated code. Listing 4.3 is an artificial Java program. It illustrates the impact of the presented reconstruction phases. Additionally, it is a non-trivial input for the control flow reconstruction algorithm.

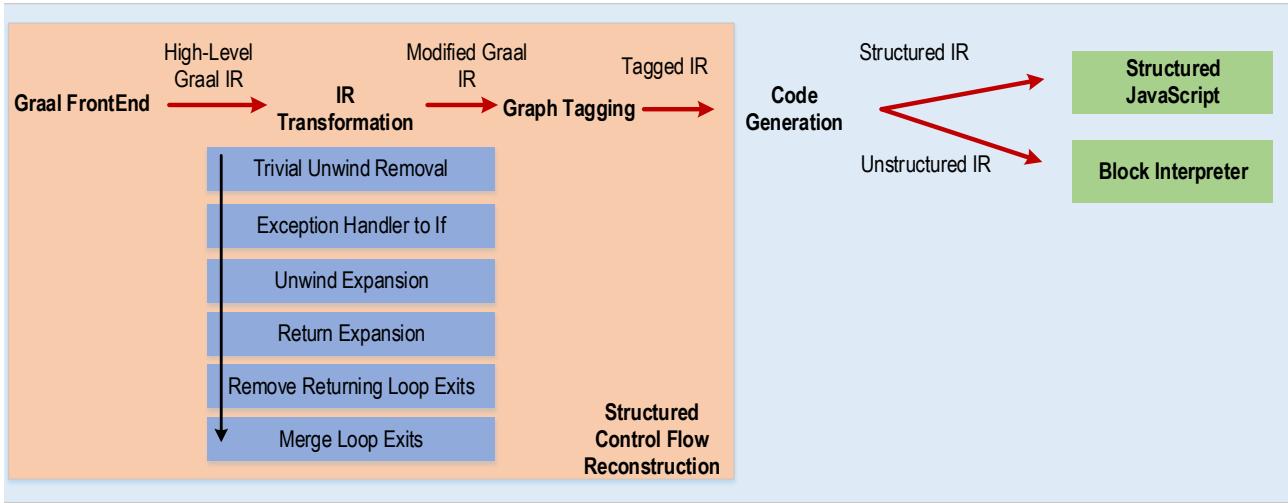


Figure 4.17: Graal AOT JS Control Flow Reconstruction Pipeline

Listing 4.4 shows the code generated for a compound conditional statement as seen in Listing 4.3 from line 3 to line 7. Figure 4.18 shows the subgraph of the IR prior to the compiler phase that removes the unstructured control flow and the resulting graph after removal of compound conditional IR.

Removing Plain Unwind Paths Phase Graal IR contains exception edges to represent exception handlers in the IR. Every node that potentially produces an exception has an exception edge attached to this node. In most cases, the edge simply leads to an unwind instruction, unwinding the current stack frame to the caller. Only if the bytecode contained code in the exception handler table for the associated byte code index, the exception edge will contain the exception handler. Graal AOT JS uses native exception handling in JavaScript. Operations that raise an exception and do not have an exception handler on the exception edge but only an unwind operation can be ignored. Every exception edge that plainly unwinds to the caller can be removed, as native exception handling in JavaScript has the same semantic. If an exception is thrown and no exception handler is present for the given type, execution continues at the caller. This removes every exception edge in the IR that only contains an unwind instruction.

Rewrite Unwind If Phase Nodes having additional code on the exception edge can be re-written to structured control flow in order to make exception control flow analyzable by the graph tagging algorithm. Graal AOT JS rewrites nodes with an exception edge attached to this node to a node followed by an if instruction and a special condition node called an *unwindIf*. If the node potentially producing an exception does so, the subsequent *UnwindIf* instruction which has the same semantic as an if, will enter the true branch. Basically, this means nodes that have an exception handler in the exception edge are rewritten to nodes followed by a plain if instruction. The condition of the if checks whether the preceding node produced an exception. Re-writing exception edges to plain control flow enables our control flow reconstruction and

detection algorithm to also analyze exception handlers which in most cases can reconstruct structured control flow for exception handlers. Listing 4.5 shows the code generated by Graal AOT JS for a non empty exception handler as seen in the original program from line 16 to line 18. Figure 4.19 shows the subgraph of the IR prior to the re-write of nodes with potential exception to an unwinding if node that represents structured control flow for exception handlers.

Expand Unwind Phase This phase duplicates unwind nodes in the IR. Graal IR normally only has one `UnwindNode` in the IR. All control flow paths unwinding the current stack frame merge on a common node in the IR. If the invariant for Graal is to only have one unwinding node in the IR, if there are more than one exception handlers in one method that can unwind, they all must merge before the unique unwind node. This merging introduces unstructured control flow, as control flow cannot be transferred in such a way without `goto` statements or `throws`. Therefore, Graal AOT JS duplicates the unwind nodes, removes the preceding merge node and lets each preceding branch end in an unwind.

Expand Return Phase Bytecode may lead to a graph where a merge is preceding a returning node. In such a case Graal AOT JS removes the merge node and duplicates the return node for each merging branch. This creates final paths and removes potentially unstructured control flow introduced through CF merges.

Remove Unused Loop Exit Phase If a loop exit precedes a node returning or unwind the current frame to the caller, the loop exit can be removed, as the return or unwind operation in a high level language exits the loop anyway.

Merge Loop Exits Phase To overcome the problem of having different successors for multiple loop exits Graal AOT JS uses a mechanism to have a structured set of loop exits. All loop exits are merged on a common node. A new switch instruction is added that dispatches the successors of the exits depending on the exit that was taken out of the loop. Figure 4.20 shows the example from Figure 4.10 after the merging of loop exits.

Listing 4.6 shows the code generated by AOT JS for the first loop of the original example from Listing 4.3 that has different code executed on different loop exits. Graal AOT JS merges the loop exits as already explained. Figure 4.21 shows the re-write of the IR subgraph for the merging of loop exits.

```

1 public static int triggerPhases(int p) {
2     // triggered from compound conditional phase
3     if (p > 10 && p < 100) {
4         System.out.println("Compound conditional");
5     } else {
6         System.out.println("Must be triggered");
7     }
8     b: {
9         for (int i = 0; i < p; i++) {
10            System.out.println(i);
11            try {
12                /*
13                 * without inlining t it cannot be proven that t will not throw an exception
14                 */
15                t();
16            } catch (Throwable t) {
17                System.out.println("Invoke with excp to unwinding if");
18            }
19            if (i > 10) {
20                break b;
21            }
22        }
23        System.out.println("only on non breaking exit printed");
24    }
25    int qsum = 0;
26    for (int i = 0; i < p; i++) {
27        for (int j = 0; j < i; j++) {
28            qsum += j;
29            if (qsum > 10)
30                System.out.println(qsum);
31        }
32    }
33    return qsum;
34 }
35 public static void t() throws Exception {
36     throw new Exception();
37 }
```

Listing 4.3: CF Reconstruction Full Example

```

1 ...
2 if((((((13 < p0))&&(((p0 < 100))))){
3     llog((const_140).toJSString());
4 }else{
5     llog((const_142).toJSString());
6 }
7 ...
```

Listing 4.4: Generated JavaScript for a Compound Conditional

```

1 ...
2 var excp_dispatch_0 = 0;
3 try{
4     Lcom_oracle_svm_aotjs_res_TriggerAllPhases_.t__V();
5 }catch( e0 ){
6     excp_dispatch_0 = 1;
7     var exception_object_63=e0;
8 }
9 if(excp_dispatch_0 == 0){}
10 else{
11     var 163 = exception_object_63;
12     llog((const_144).toJSString());
13 }
14 ...

```

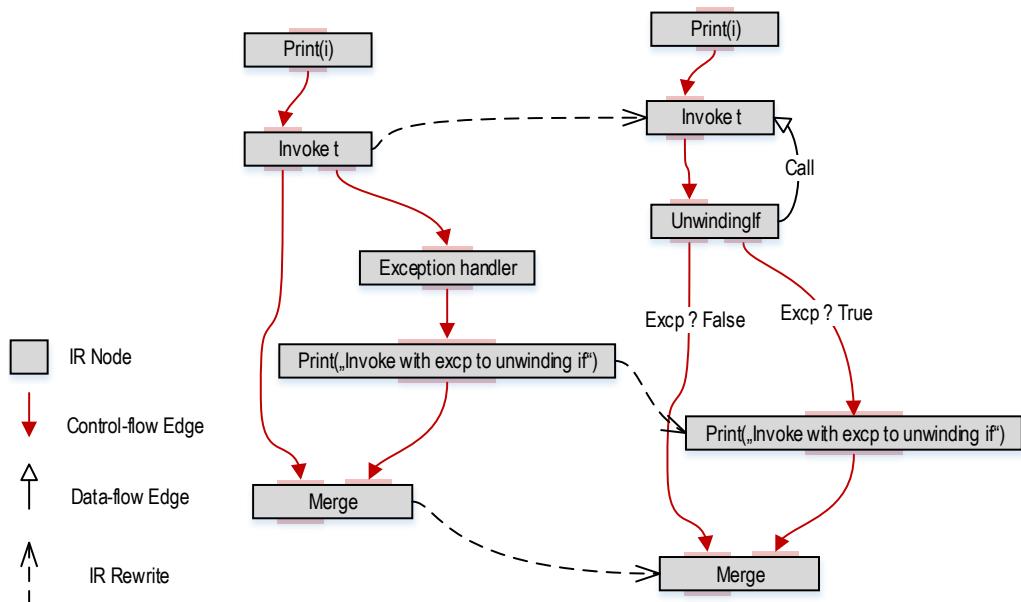
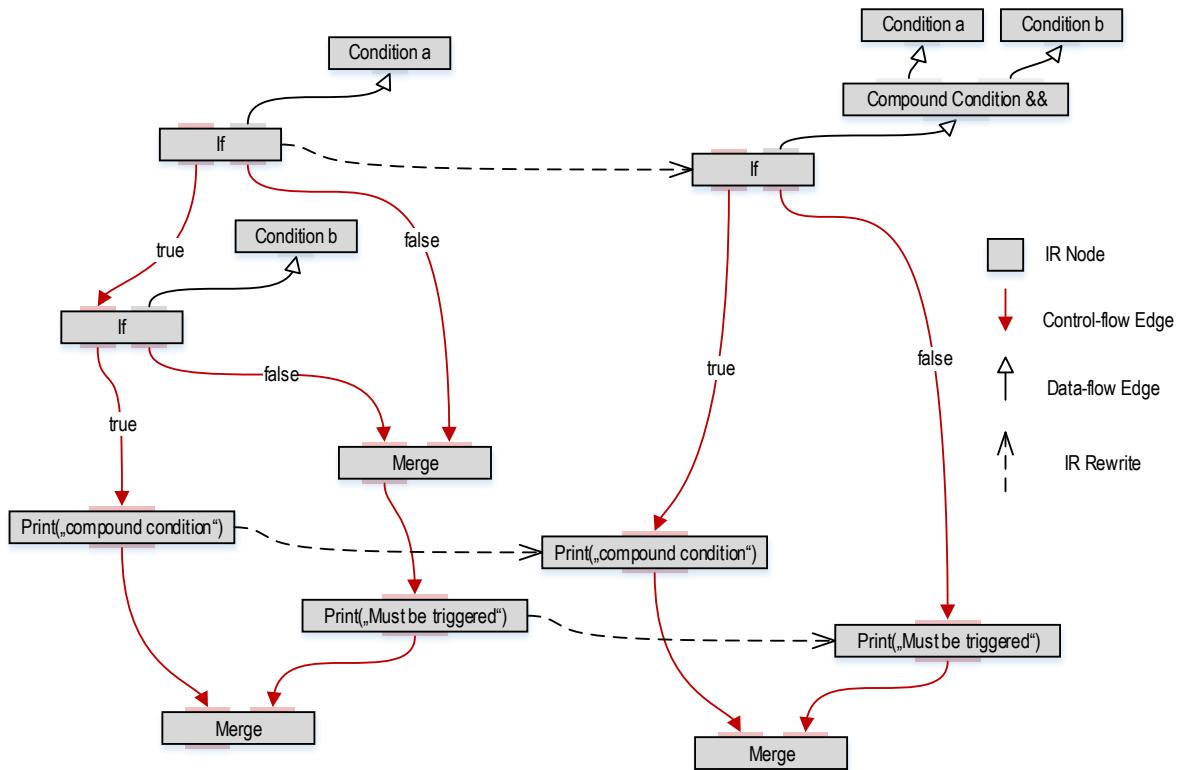
Listing 4.5: Generated JavaScript for an Unwind If

```

1 ...
2 looplabel_42 :while(true){
3 ...
4 }
5 switch(l188){
6     case 0:{
7         llog((const_146).toJSString());
8         break;
9     }
10    default:{
11        break;
12    }
13 }
14 ...

```

Listing 4.6: Generated JavaScript for multiple Loop Exits



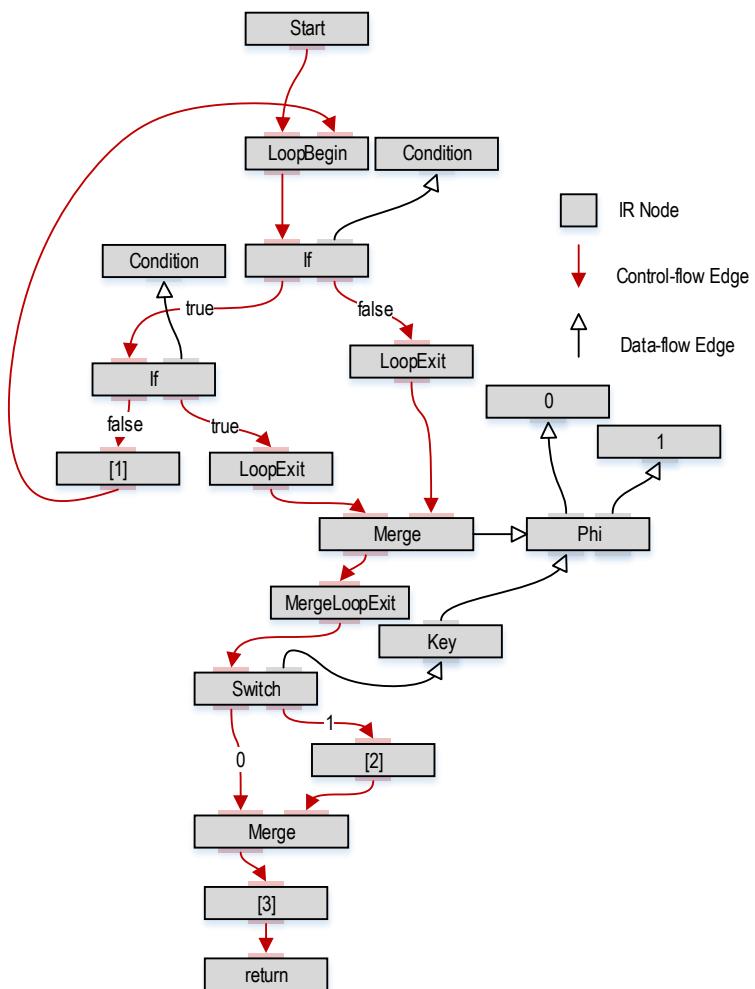


Figure 4.20: Structured control flow: multiple loop exits are merged.

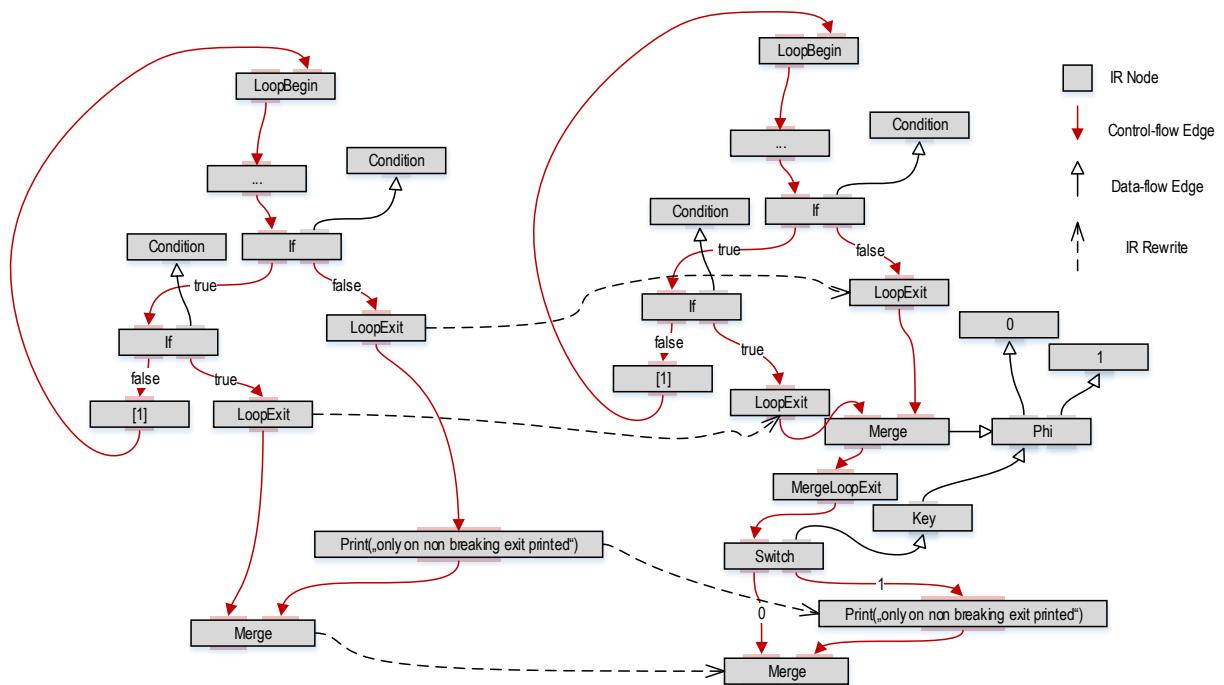


Figure 4.21: Graal AOT JS Merge Loop Exit Phase Example

Chapter 5

JavaScript Code Generation

This chapter presents the major paradigms of the Java programming language and their mapping to the generated JavaScript source code. The mapping of the type system, exception handling and memory management from Java to JavaScript are covered in detail.

5.1 Compilation of the Java Type System to JavaScript

JavaScript is a dynamically typed language. A local variable in JavaScript can be of any type at run time, and even change the type dynamically [16]. Java, however is strongly typed, it requires each expression to have a static type that can be statically determined at compile time. Polymorphism allows the dynamic type of an expression to be a sub-type of the real type, which prohibits certain optimization at compile time, as the *exact* type (dynamic type) is not known.

5.1.1 Primitives

boolean, char, short, int Graal AOT JS models integer based primitives in 32 bit range using ASM.js [43] as native JavaScript integers. JavaScript integers are 32 bit signed integers in two's complement representation. Runtime checks are necessary to ensure the valid value ranges of Java types smaller than `int`, e.g., for `short`. Operations on `short` might lead to values outside the 16 bit range and must therefore be checked for overflow and adjusted. Warren presents techniques to do so in [66], which are adapted by Graal AOT JS in order to generate code with valid numeric ranges.

long Graal AOT JS needs to emulate Longs since the standard number type in JavaScript is the IEEE 754 double precision floating point type, which has a maximum integral part in the range of $[-2^{53} : 2^{53}]$. We emulate longs with a JavaScript object that has two properties representing the high and low word of the long value, each of which is a signed 32 bit two's complement number.

float Java expressions of type float never exceed single precision. However, JavaScript only provides double precision numbers. To ensure an expression of type float never exceeds its precision, every expression of type float must be checked. The ECMAScript 6 (Harmony) standard [17] defines a function `Math.fround` [44] that rounds a double precision floating point number to the nearest single precision number. To enable competitive performance, Graal AOT JS features a mode that omits checks for single precision floating point ranges and therefore might lead to floats that exceed single precision.

double Java double primitives are in the IEEE 754 double precision floating point format [28]. Therefore Graal AOT JS maps Java primitive doubles directly to JavaScript primitive numbers.

5.1.2 Objects

Graal AOT JS directly maps Java objects to JavaScript objects. All fields and methods of a type that are deduced to be reachable by the closed world analysis are marked for compilation. Listing 5.1 shows a trivial Java *Person* class with three member variables. Listing 5.2 shows the JavaScript code generated by Graal AOT JS for the *Person* class.

Inheritance Graal AOT JS directly maps the Java inheritance model to JavaScript's prototype-based inheritance model. We build a JavaScript prototype chain reflecting the Java type tree with a `java.lang.Object` prototype as the root prototype. Figure 5.1 illustrates the type tree built with `java.lang.Object` as the "root" object and, e.g., an instance of a `java.lang.Integer` whose `__proto__` pointer references the `java.lang.Integer` prototype object.

Listing 5.2 illustrates code generation for inheritance. The `Person` function object's prototype is an object created from the `java.lang.Object` prototype. After the `Object.create` call the `__proto__` pointer of the prototype of the `Person` function object points to the `java.lang.Object.prototype`. Graal AOT JS maps Java inheritance to prototype-based inheritance leveraging the prototype chain of JavaScript created via the `__proto__` pointer.

```

1 public class Person {
2     private final String name;
3     private final int age;
4     private final String email;
5     public Person(String name, int age, String email) {
6         super();
7         this.name = name;
8         this.age = age;
9         this.email = email;
10    }
11    public String getName() {
12        return name;
13    }
14    public int getAge() {
15        return age;
16    }
17    public String getEmail() {
18        return email;
19    }
20 }

```

Listing 5.1: Person Class Java

```

1 Lcom_oracle_svm_aotjs_res_Person_.prototype = Object.create(Ljava_lang_Object_.
    prototype);
2 Lcom_oracle_svm_aotjs_res_Person_.constructor = Lcom_oracle_svm_aotjs_res_Person_;
3 function Lcom_oracle_svm_aotjs_res_Person_(){
4     Ljava_lang_Object_.call(this);
5     this.prop_email_Lcom_oracle_svm_aotjs_res_Person_ = null;
6     this.prop_name_Lcom_oracle_svm_aotjs_res_Person_ = null;
7     this.prop_age_Lcom_oracle_svm_aotjs_res_Person_ = ( 0 | 0 );
8 };
9 Lcom_oracle_svm_aotjs_res_Person_.prototype.
    init__Ljava_lang_String_ILjava_lang_String_V = function(p0,p1,p2,p3){
10    this.prop_name_Lcom_oracle_svm_aotjs_res_Person_ = p1;
11    this.prop_age_Lcom_oracle_svm_aotjs_res_Person_ = p2;
12    this.prop_email_Lcom_oracle_svm_aotjs_res_Person_ = p3;
13    return;
14 };
15 Lcom_oracle_svm_aotjs_res_Person_.prototype.getAge__I = function(p0){
16     return (this.prop_age_Lcom_oracle_svm_aotjs_res_Person_);
17 };
18 Lcom_oracle_svm_aotjs_res_Person_.prototype.getEmail__Ljava_lang_String_ = function(p0)
    {
19     return (this.prop_email_Lcom_oracle_svm_aotjs_res_Person_);
20 };
21 Lcom_oracle_svm_aotjs_res_Person_.prototype.getName__Ljava_lang_String_ = function(p0){
22     return (this.prop_name_Lcom_oracle_svm_aotjs_res_Person_);
23 };

```

Listing 5.2: Person Class JavaScript

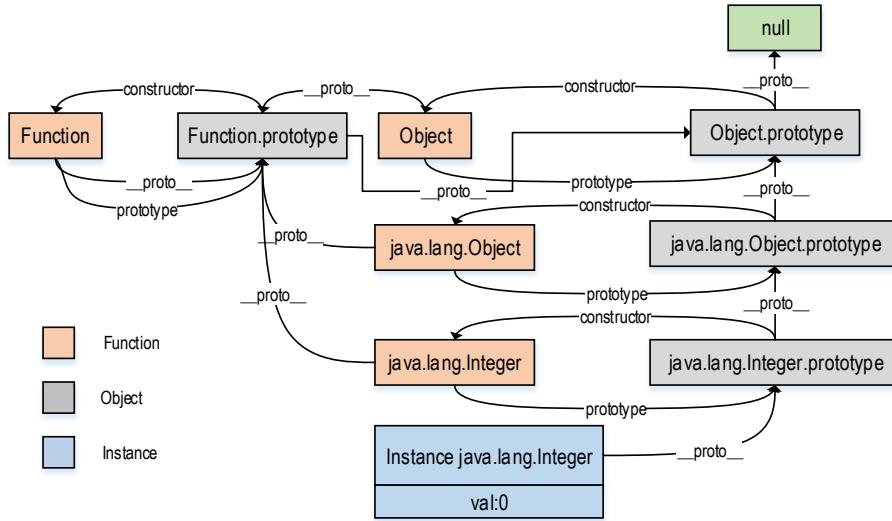


Figure 5.1: AOT JS Inheritance Tree.

Garbage Collection Java semantic relies on *exact* GC like JavaScript does. As Graal AOT JS uses "native" JavaScript objects, we can leverage the garbage collector of the executing JavaScript VM. This removes the burden of manually performing GCs. Other approaches compiling to JavaScript like the XMLVM [51] which compiles Java to C and then to JavaScript using Emscripten [72], requires a distinct GC. This is the case as during the compilation from Java to C, GC is no longer automatically available. The XMLVM therefore compiles the implementation of a Boehm GC [4] algorithm from C to JavaScript.

Interfaces A Java variable can have an interface as its static type. JavaScript is an untyped language and does not support the concept of static types. To support type checks against interfaces we compile additional type bits to compiled `java.lang.Class` objects enabling those checks. Additionally, default methods introduced in Java 8 [35] require interface types to be compiled to JavaScript.

Arrays JavaScript's concept of Arrays is similar to the one of Java. JavaScript offers an intrinsic `Array` object that is capable of representing a Java array. Code generated by Graal AOT JS uses this `Array` object, however, Java requires all Objects, including Arrays, to support the `getClass` method to access an instances class object. This requires Graal AOT JS to extend the prototype of every `Array` object with a member variable for the `java.lang.class` object of the instance. Graal AOT JS features an optional compilation mode using `TypedArrays` [44] for primitive arrays.

Type Checks Java knows two different type checks, static and dynamic ones. Static type checks are those, where the type is known at compile time like, e.g., the `instanceof` keyword which requires as a second operand a type. Dynamic type checks are those, where the type against which is checked is not known at compile time, but only at run time. For example, the type of the concrete receiver of a call to the `java.lang.Class.isInstance` method is only known at runtime.

Graal models this behavior with two different type checks, the static and dynamic type check. Graal AOT JS has a special treatment for the static type check, as types are partly mapped to JavaScript natives and objects. The static type check rules are:

- Arrays: Graal AOT JS generates code for a runtime call to `dynamic instance of`.
- Interfaces: Graal AOT JS generates code for a runtime call to `dynamic instance of`.
- Primitives: Graal AOT JS generates code performing the type check inline using the JavaScript `typeof` keyword.
- Long: Graal AOT JS generates code performing an `instanceof` operation, as Java Longs are emulated with objects.
- Objects: Graal AOT JS generates code performing an `instanceof` check.

Dynamic type checks are all performed at run time and implemented via a simple JavaScript function. Listing 5.3 shows Graal AOT JS' implementation for the dynamic type check. It is implemented using JSIntrinsicsification, which is explained in Section 5.4. Dynamic type checks are always object type checks, as the dynamic type of an expression at runtime can never be unboxed.

5.2 Constant Data

The Java language specification has three bytecodes for loading of constants. The `ldc`, `ldc_w`, `ldc2_w` handling constants of 16, 32 and 64 bit, including reference types. Graal AOT JS loads all classes during compilation. This is necessary for the static analysis and ensuing optimizations. Loading the classes for compilation will already execute static initializers and result in initialized constants and static data of all classes. Every time our byte code parser encounters one of the load constant byte codes or load/store static, the constant or static field needs to be registered for compilation, as it is

```

1 function isA (object ,mirror){
2   if(object==null){
3     return false;
4   }
5   if(mirror==null || mirror==undefined){
6     throw new Error("Object mirror cannot be null");
7   }
8   var hub=null;
9   if(Array.isArray( object )) {
10    hub=object.hub;
11  } else{
12    hub=object.gethub_generated();
13  }
14  return mirror .@java.lang.Class#isAssignableFrom#(mirror ,hub);
15 };

```

Listing 5.3: Graal AOT JS Dynamic Type Check

needed at run time by the referencing code. Thus, we create AOT, an initial "heap" for the compiled JavaScript code. This heap will be initialized during startup of the JavaScript code and initialize all static fields and constants that are referenced by the compiled JavaScript Code.

Creating the initial JS heap increases also the code size of the generated JavaScript code. This is the case, as every object property must be accessed and initialized. Cyclic dependencies must be broken with initializations.

5.3 Exception Handling

Java's exception class hierarchy is vital for the distinction between logical errors, runtime exceptions or even control flow dispatches. JavaScript makes a less effective distinction. In JavaScript every object can be thrown. The intrinsic `Error` object denotes runtime errors, but exception rules are less strict, e.g., division by zero does not produce a runtime exception. Graal AOT JS is able to use the native JavaScript exception handling mechanism. Code for exception handlers is generated with the `try-catch` statement. Code generation for the code in the IR's exception edge emits an if statement after the `try-catch` block that is entered if the preceding statement produced an exception. Listing 5.4 shows code for a simple exception handler in Java. Listing 5.5 shows the generated JavaScript code after compilation of the example from Listing 5.4. The true branch of the if instruction in Listing 5.5 is executed if no exception happened during the execution of the preceding node, whereas the false branch marks the exception handler. Graal AOT JS re-writes exception handlers to structured control flow in order to guarantee that control flow is analyzable.

```

1 public class SimpleException {
2     public static void e() throws Exception {
3         throw new Exception();
4     }
5     public static void catchE() {
6         try {
7             e();
8         } catch (Throwable t) {
9             System.out.println(t.getMessage());
10        }
11    }
12    public static void main(String[] args) {
13        catchE();
14    }
15 }
```

Listing 5.4: Java Simple Exception Handler

```

1 Lcom_oracle_svm_aotjs_res_SimpleException_.prototype = Object.create(Ljava_lang_Object_
 .prototype);
2 Lcom_oracle_svm_aotjs_res_SimpleException_.constructor =
 Lcom_oracle_svm_aotjs_res_SimpleException_;
3 function Lcom_oracle_svm_aotjs_res_SimpleException_(){
4 Ljava_lang_Object_.call(this);
5 };
6 Lcom_oracle_svm_aotjs_res_SimpleException_.catchE__V = function(){
7     var excp_dispatch_0 = 0;
8     try{
9         Lcom_oracle_svm_aotjs_res_SimpleException_.e__V();
10    }catch( e0 ){
11        excp_dispatch_0 = 1;
12        var exception_object_3=e0;
13    }
14    if(excp_dispatch_0 == 0){
15        return;
16    }
17    else{
18        var l3 = exception_object_3;
19        llog((l3.prop_detailMessage_Ljava_lang_Throwable_));
20        return;
21    }
22 };
23 Lcom_oracle_svm_aotjs_res_SimpleException_.e__V = function(){
24     var l2 = (new Ljava_lang_Exception_());
25     throw l2;
26 };
```

Listing 5.5: JavaScript Simple Exception Handler

5.4 JavaScriptify - Compiler Intrinsics & Native Calls

Compiling Java bytecode AOT introduces two additional problems to the limitations mentioned in Chapter 1: *Native Calls* and *Compiler Intrinsics*.

As Graal AOT JS re-uses the byte code parser of the Graal platform, compiler intrinsics are in the IR after parsing as special nodes. A typical example for a compiler intrinsics would be the `java.lang.Math` class where certain methods with dedicated CPU support are intrinsified during bytecode parsing.

Native calls are invocations to methods with the native identifier. The JDK, however, has several points, where native methods are called, some of which are intrinsified in Graal. A good example for a native call is the `java.io.FileInputStream.read()` method.

In order to support intrinsics and native calls Graal AOT JS has to provide runtime support libraries modeling the desired behavior in JavaScript. This introduces the problem of accessing Java types from JavaScript. In order to ensure all types referenced from the runtime libraries, as well as to handle the name mangling, we use a mechanism called *JSIntrinsification*. JSIntrinsification allows the programmer of the Graal AOT JS compiler to write JavaScript code and use Java types safely. JSIntrinsification guarantees that all referenced Java types are seen by the points-to analysis and properly complied in their naming.

JSIntrinsification is a text substitution mechanism. Whenever a programmer wants to use Java types in JavaScript, he / she adds a JavaScript file to the compilation unit for the AOT JS compilation. Graal AOT JS then reads the file, extracts the intrinsified types and substitutes them in the final JavaScript with the correct type, field and method names. Additional Graal AOT JS guarantees all referenced types, as long as they are on the class path, are compiled to JavaScript. To trigger a substitution of an intrinsified type in JavaScript, special escape sequences must be used during programming, to distinguish normal JavaScript code from JSIntrinsifications.

Intrinsification	Escape Sequence
Type	@TypeName@
Method	@TypeName@# MethodName#
Field	@TypeName@\$FieldName\$

Table 5.1: JSIntrinsify Escape Sequences

Most of the Graal AOT JS runtime support libraries use the mechanism of JSIntrinsification like, e.g., type checks at runtime or `sun.misc.unsafe` support. Listing 5.6 shows the implementation of the `unsafe.load` API with JSIntrinsification.

```

1 function unsafe_load_runtime( object , offset ){
2   var ooffset=offset;
3   if( offset instanceof Long64){
4     ooffset = offset . low ;
5   }
6   if( Array . isArray ( object )) {
7     // array case
8     var dynamic_hub=object . hub ;
9     // call in java field layout encoding
10    var layout_encoding=dynamic_hub . @com . oracle . svm . core . hub . DynamicHub@§layoutEncoding
11    § ;
12    var base_offset=@com . oracle . svm . core . hub . LayoutEncoding@ . @com . oracle . svm . core . hub .
13      LayoutEncoding@#getArrayBaseOffset #( layout_encoding );
14    var index_scale=@com . oracle . svm . core . hub . LayoutEncoding@ . @com . oracle . svm . core . hub .
15      LayoutEncoding@#getArrayIndexScale #( layout_encoding );
16    var realoffset= (( ooffset - base_offset ) / index_scale)|0;
17    return object [ realoffset ];
18  } else {
19    // object case
20    var dynamic_hub=object . gethub_generated ();
21    var name = object . __field__name__( ooffset );
22    return object [ name ];
23  }
24 };

```

Listing 5.6: Graal AOT JS Unsafe Load JavaScript

5.5 Unsafe Memory Access

The Sun class `sun.misc.unsafe` [53] class is a low-level Java API to perform low-level and unsafe operations. It allows for systems programming in Java, but was never public API, nor intended to be used outside the JDK. Graal AOT JS enables the usage of all non-thread related unsafe operations via field offset tables. Graal AOT JS generates for every type a table containing a mapping of native field-offsets to field-names. The native-field offset is the offset in bytes of a field from the start of an object in, e.g., Hotspot on an AMD64. Therefore Graal AOT JS is capable of modeling "native" memory access by compiling the field offsets during unsafe accesses together with type field offset tables to JavaScript. Listing 5.6 shows the JavaScript implementation using JSIntrinsicsification for an unsafe load operation.

Chapter 6

Ahead-of-Time Optimizations

This chapter presents optimizations applied to the generated JavaScript code to increase run-time performance. The first part of the chapter lists those optimizations that can be leveraged via the usage of the Graal eco system, which features a rich optimization framework. The second part proposes optimizations that are unique to the generation of JavaScript code.

6.1 Graal Optimizations

Graal offers a set of standard high-level compiler optimizations including global value numbering [8], constant folding, strength reduction [3], conditional elimination [57], method inlining and partial escape analysis [58]. Graal AOT JS can leverage these sophisticated compiler optimizations without any additional implementation effort.

However, compilation to a high-level language is different than to native code for different reasons. The most important reason is code size, as AOT compilation requires the entire code to be pre-compiled. Therefore Graal AOT JS does not perform aggressive inlining like Graal or the server compiler does. We only inline small functions and constructors, to keep code size reasonable. Especially, the inlining of small constructors is valuable for the run-time performance of the code generated by Graal AOT JS. Inlining of constructors enables the partial escape analysis [58] of Graal, which removes many heap allocations and replaces them with stack allocated scalars or moves them in branches where objects escape.

6.2 Exception Handler Optimizations

This optimization mainly pays off in JavaScript VMs that do not optimize exception handlers. The optimizing compiler of Google's V8 [20], for example, bails out on methods containing the `try-catch` keywords.

Graal AOT JS is aware of all points in the code, where Java exceptions might be thrown, which enables us to model exception handling completely without using "native" JavaScript exception handlers. Whenever an implicit exception would be thrown we do not throw an exception with the JavaScript keyword "throw", but rather return a custom exception and exit the function on the return path. Typical implicit exceptions of Java are, e.g., de-referencing null or violating an array bound. Listing 6.1 illustrates the approach. Function `caller1` calls function `f1` in plain exception mode and `caller2` calls `f2` in wrapped exception mode. It is crucial to point out that with exception wrapping *every* callsite must allocate the call's return value to a local variable. The return value must be checked for a pending exception even if the called function's return type is void. The advantage of this optimization is that (although the return type is not fixed and may be megamorphic) the function is still compiled with V8's optimizing compiler which makes a big difference in performance.

6.3 SSA Node Inlining

As presented in Chapter 2, Graal's IR is in SSA form. Every node producing a value represents a new local variable in the underlying program. Performance and code size of the generated code would be compromised, if every node producing a value would be associated with a new local variable in JavaScript. Thus, we use an optimization for inlining static-single-assignment nodes at their usages. Inlining of SSA nodes generates the code for the value of the node at the usage and does not allocate a local variable for each node. Our static-single-assignment node inlining heuristic is based on the subsequent list of assumptions:

- Nodes with less than 2 usages can be inlined.
- Null can always be inlined.
- Constants of primitive types can always be inlined.
- Nodes with side-effects are never inlined.

```
1 // native JS Exceptions
2 function f1(){
3     if(a){
4         throw new java_lang_NullPointerException();
5     }
6 }
7 function caller1(){
8     try{
9         f1();
10    }catch(e){
11        if(e instanceof java_lang_NullPointerException){
12            ...
13        }
14        // else unwind
15    }
16 }
17 // wrapped exception handlers
18 function f2(){
19     if(a){
20         return new ExceptionWrapperType(new java_lang_NullPointerException());
21     }
22 }
23 function caller2(){
24     var ret=f1();
25     if(isException(ret)){
26         var exc=ret.exception;
27         if(exc instanceof java_lang_NullPointerException){
28             ...
29         }
30         // else unwind
31     }
32 }
```

Listing 6.1: Exception wrapping optimization example (JavaScript)

```
1 static void foo(int p) {
2     int a = p + 9 * 11 + 12 * p;
3     int b = p * 10 * 7 + p;
4     int s = 0;
5     for (int i = 0; i < p; i++) {
6         s += a;
7         s += b;
8         s += i;
9     }
10    System.out.println(s);
11 }
```

Listing 6.2: Inlining Example Java

- Nodes whose usage contains a conditional expression of the form $a ? b : c$ are not inlined.
- Array accesses are not inlined, as this requires additional bounds checks.

Listing 6.2 shows a simple function `foo` computing a function and printing the result. Listing 6.3 shows the code generated by Graal AOT JS after the compilation of the Java bytecode without the inlining of SSA nodes. For each SSA value node a local variable is generated also for the constants like, e.g., 99 or 12. Listing 6.4 shows the code generated by Graal AOT JS with the SSA node inlining optimization enabled.

Using our static-single-assignment inlining heuristic between 20% and 30% of all IR data nodes can be inlined at their usage(s).

```

1 Lcom_oracle_svm_aotjs_res_Inlining_.foo__IV = function(p0){
2   var l12 = 0;
3   var l13 = 0;
4   var l13 = 99;
5   var l14 = (( ( (p0) + (l13) ) | 0));
6   var l15 = 12;
7   var l16 = (( ( (p0) * (l15) ) | 0));
8   var l17 = (( ( (l14) + (l16) ) | 0));
9   var l138 = 70;
10  var l137 = (( ( (p0) * (l138) ) | 0));
11  var l18 = (( ( (p0) + (l137) ) | 0));
12  var l125 = 1;
13  var l19 = 0;
14  l12 = l19;
15  l13 = l19;
16  looplabel_11 :while(true){
17    var l115 = ((l13 < p0));
18    if(l115){
19      var l126 = (( ( (l13) + (l125) ) | 0));
20      var l122 = (( ( (l17) + (l12) ) | 0));
21      var l123 = (( ( (l18) + (l122) ) | 0));
22      var l124 = (( ( (l13) + (l123) ) | 0));
23      l13 = l126;
24      l12 = l124;
25      continue looplabel_11;
26    }
27    else{}
28    break;
29  }
30  log(l12);
31  return;
32 };

```

Listing 6.3: Inlining Example JavaScript without inlining of SSA nodes.

```

1 Lcom_oracle_svm_aotjs_res_Inlining_.foo__IV = function(p0){
2     var 112 = 0;
3     var 113 = 0;
4     var 17 = ((((( ( (p0) + (99) ) | 0))) + ((( ( (p0) * (12) ) | 0))) ) | 0));
5     var 18 = ((( ( (p0) + ((( ( (p0) * (70) ) | 0))) ) | 0)));
6     var 19 = 0;
7     112 = 19;
8     113 = 19;
9     looplabel_11:while(true){
10     var 115 = ((113 < p0));
11     if(115){
12         var 126 = ((( (113) + (1) ) | 0));
13         var 122 = ((( (17) + (112) ) | 0));
14         var 124 = ((( (113) + ((( (18) + (122) ) | 0))) ) | 0));
15         112 = 124;
16         113 = 126;
17         continue looplabel_11;
18     }
19     else{}
20     break;
21 }
22 log(112);
23 return;
24 };

```

Listing 6.4: Inlining Example JavaScript with inlining of SSA nodes.

Chapter 7

Evaluation

This chapter presents the evaluation of the Graal AOT JS compiler. It contains a number of benchmarks that are evaluated with respect to number of inlined local variables, the run-time performance, the amount of reconstructed control flow and the code size. The first part of the chapter explicitly shows an evaluation of the SSA node inlining optimization. Then, the chapter presents the evaluation of the control flow reconstruction algorithm in terms of the amount of restructured control flow. The next part of the chapter deals with evaluation of the code size of the generated JavaScript code. The last part of the chapter shows the evaluation of the run-time performance of the generated JavaScript code.

The presented Graal AOT JS compiler which has been implemented on-top of the Graal just-in-time compiler, was evaluated by running and analyzing a set of benchmarks of different sizes. All benchmarks were executed on a desktop-class Intel i5 processor (2010) with 2 cores, 4 virtual threads featuring 8GB of RAM and a core speed of 2.4 GHz running Fedora 21(64 bit).

We ran each of the presented benchmarks with different optimization configurations. As we wanted to compare the results against the Java HotSpot VM and against hand-written JavaScript versions of (some) of the benchmarks, we not only measure the peek performance but also the start-up performance. The HotSpot server compiler has a slow start-up performance, whereas Google's JavaScript engine V8 which compiles every method upon its first execution with a simple base-line compiler, has a fast start-up performance. On the other hand, the server compiler reaches a better peak performance than V8. Each test run consisted of three different configurations of the benchmark and ten iterations for each configuration. We took the arithmetic mean of all configurations and put them in direct comparison to the HotSpot server compiler. For the benchmarks binarytrees, deltablue, nbody, fasta, mandelbrot and SPECjbb2005 we measured their execution time in milliseconds, whereas for Linpack, SciMark2a and JBox2D we measured their score result.

7.1 Benchmarks

In the following we give a short description for each benchmark used in the experiments:

binarytrees is part of the Computer-Language-Benchmark-Game (CLBG); it is a small benchmark performing many binary tree allocations and recursive calls up to a certain depth [7].

deltablue is a prominent one-way constraint solver, originally developed for the *Smalltalk* language by Maloney and Wolczko. [12].

nbody is a simple numeric benchmark performing an nbody simulation [7].

fasta (V#4) is also part of the CLBG generating and writing random DNA sequences.

linpack is a benchmark solving a set of linear equations, heavily relying on floating-point performance [36].

SciMark2a is a benchmark performing scientific computations including a fast Fourier transformation, a Jacobi successive over-relaxation, a Monte Carlo simulation, a sparse matrix multiplication and a LU matrix factorization [54].

mandelbrot is a CLBG benchmark generating a mandelbrot fractal.

SPECJbb2005 is a single-threaded sequential version of the original *SPECjbb2005* Java server benchmark [56] emulating a three tier client-server application. The modifications include a removal of the file logging and execution of all threads on the main thread. For Graal AOT JS usage warehouses are loaded sequentially and a warehouse thread runs on the main thread. Iterations are limited to 5000 per warehouse. The execution was measured on four to eight warehouses with an increment of one.

JBox2D is a Java port of the prominent physics engine Box2d [5]. The Java port is released with its own benchmark called "Piston" [31] which sets up and performs a physical simulation over several iterations in various configurations.

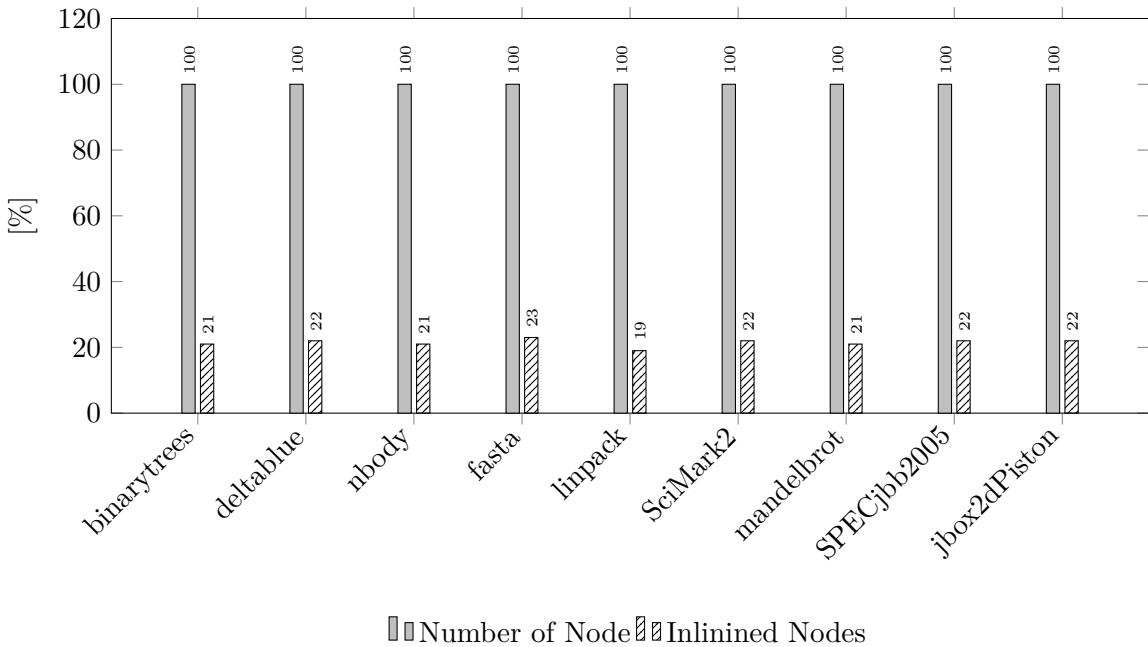


Figure 7.1: Percentage of inlined nodes using the SSA Node Inlining Optimization in relation to all SSA nodes.

7.2 SSA Node Inlining Optimization

Figure 7.1 shows the evaluation of the SSA node inlining optimization presented in Chapter 6. For each benchmark between 19% and 23% of all SSA value nodes were inlined. These numbers are stable across all presented benchmarks, no matter of the size of the benchmark.

Analysis of the results showed that the type of nodes used in Graal IR on average during compilation of the presented benchmarks yields a stable distribution of node types in the IR. This is not depending on the benchmark. Therefore, SSA node inlining which makes inlining decisions based on the type of the node, directly correlates with the percentage of inlinable nodes in the IR during compilation. This percentage of inlinable nodes is stable.

7.3 Control Flow Reconstruction

The control flow reconstruction algorithm presented in Chapter 4 works on arbitrary Java bytecodes and yields a high amount of restructured control flow. The evaluation of the algorithm on the presented benchmarks can be seen in Figure 7.2.

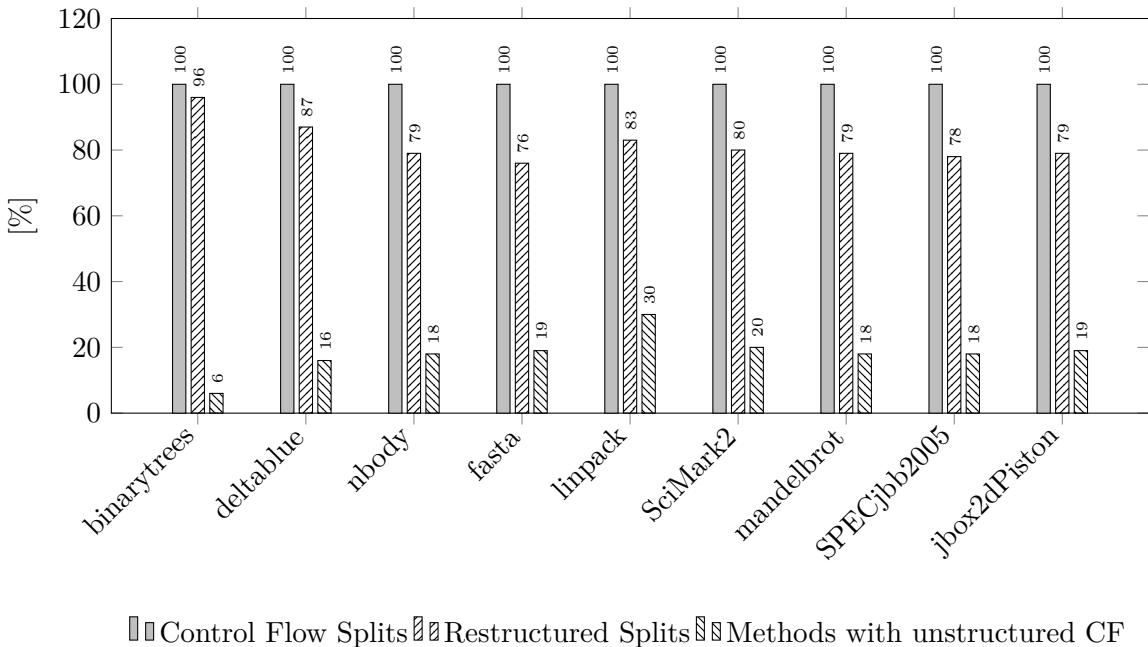


Figure 7.2: Number of restructured split nodes and number of methods with unstructured control flow compared to all methods.

More than 75% of the control flow splits could be restructured leaving less than 25% of the methods with unstructured control flow. The numbers leave space for further optimizations. Especially nested loops with shared exception handlers may currently lead to improperly restructured control flow. One major problem of the remaining 25% of unstructured control flow is the heavy usage of labeled blocks of certain methods. If labeled blocks are used to emulate the `goto` statement, graph tagging bails out, as Graal AOT JS only optimizes and reconstructs structured control flow. For such cases a different solution than the *BlockInterpreter* should be found, that, with the usage of labeled blocks, reconstructs control flow of emulated `goto` statements. There are approaches to remove `goto` statements [18] that could be combined with graph tagging to eliminate `gos` with labeled blocks.

7.4 Code Size

Figure 7.3 shows the sizes of the source code and the target code of the benchmarks compiled by the Graal AOT JS compiler. The sizes range from a few lines (CLBG) to several thousand lines of code. Even small programs may result in an enormous amount of compiled code due to heavy usage of the JDK. For example, the `nbody` benchmark requires big parts of the JDK to be compiled which results in a code size comparable to the one of `SPECjbb2005`. The general overhead in code size is high. 20% to 50% of the code size can be attributed to the initial heap initialization. If more compact code is

Java Benchmark	Java LOC (approx.)	JS Code Size	Unit
binarytrees	58	330	KB
delta blue	607	836	KB
nbody	130	9.9	MB
fasta	125	950	KB
linpack	284	569	KB
Sci Mark 2a	1890	3.9	MB
mandelbrot	90	10	MB
SPECjbb2005	12 800	14.3	MB
JBox2D	13 478	12.5	MB

Table 7.1: Total code sizes of the presented benchmarks.

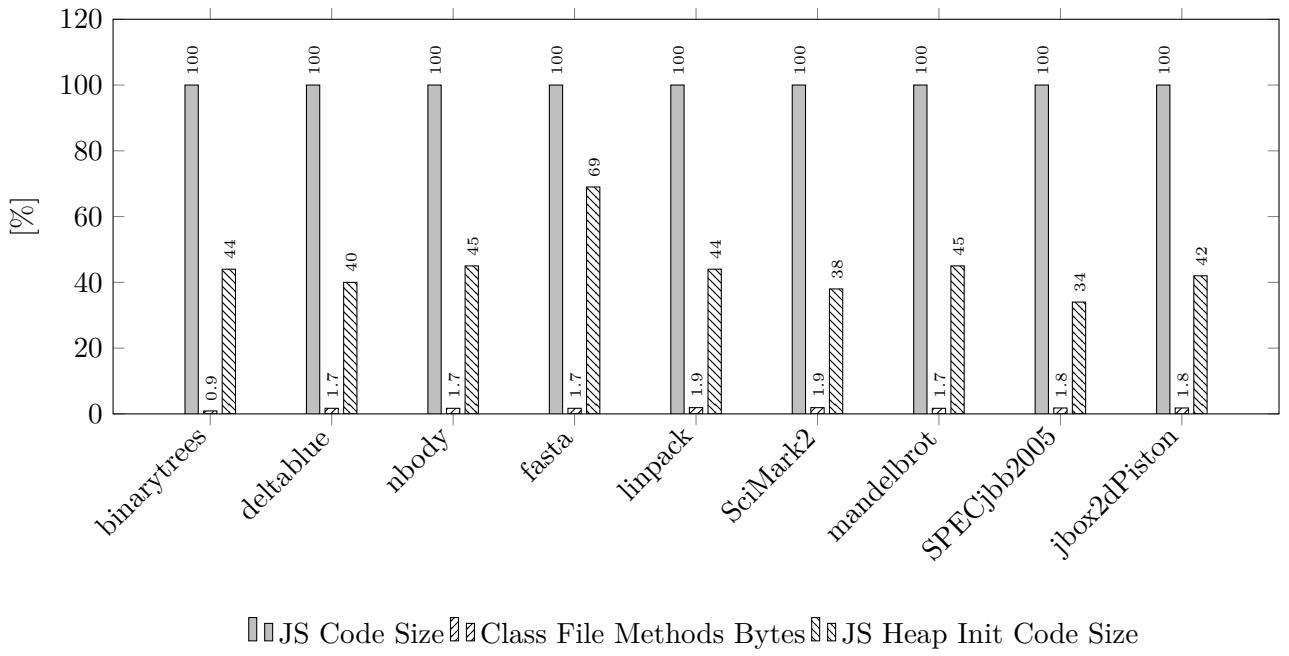


Figure 7.3: Code Size of the generated JavaScript of Graal AOT JS.

desired, tools like [21] or [37] can be used, which achieve high compression rates and would decrease the code size by 30 – 60%. For the presentation of benchmarks we decided to evaluate the performance with unmodified code produced by Graal AOT JS.

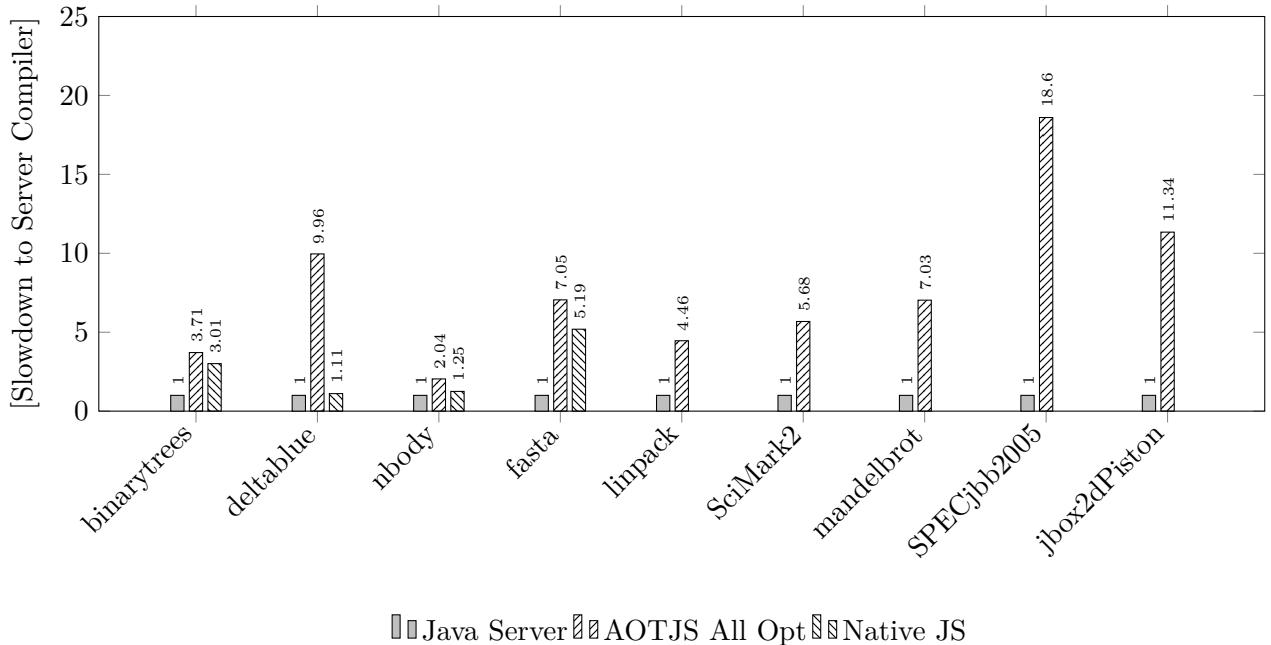


Figure 7.4: Relative performance of Graal AOT JS compared to the Java HotSpot Server Compiler.

7.5 Run-time Performance

The run-time performance measurements relative to the HotSpot server compiler¹ are presented in Figure 7.4. The server compiler offers the highest peak performance of all Java virtual machines currently on the market. For the execution of the JavaScript benchmarks that were generated from Java we used Google’s Chrome Browser² as the target platform. Internally, Chrome uses Google’s V8 [20] engine for executing JavaScript. For details about V8 we refer to [20] and for Crankshaft, V8’s optimizing compiler to [19].

As we can see in Figure 7.4, JavaScript generated by Graal AOT JS is slower than Java on the server compiler and slower than JavaScript on V8. The benchmarks binarytrees, nbody and fasta are $1.3x$ to $1.4x$ slower under Graal AOT JS than on V8, which is reasonable, if one considers the overhead introduced by the Java semantics. Also linpack performs quite well, although a native JavaScript baseline is missing.

Different optimizations have different impacts on the benchmarks. Small benchmarks such as binarytrees profit from the inlining of constructors, which enables escape analysis and increases the performance. Nbody and linpack are examples for benchmarks that do not rely on elaborate Java features. Applying certain optimizations on them, besides control flow analysis, does not result in sig-

¹JDK1.8.0_11 (64 bit)

²Chrome 42.0.2311.135 (64-bit)

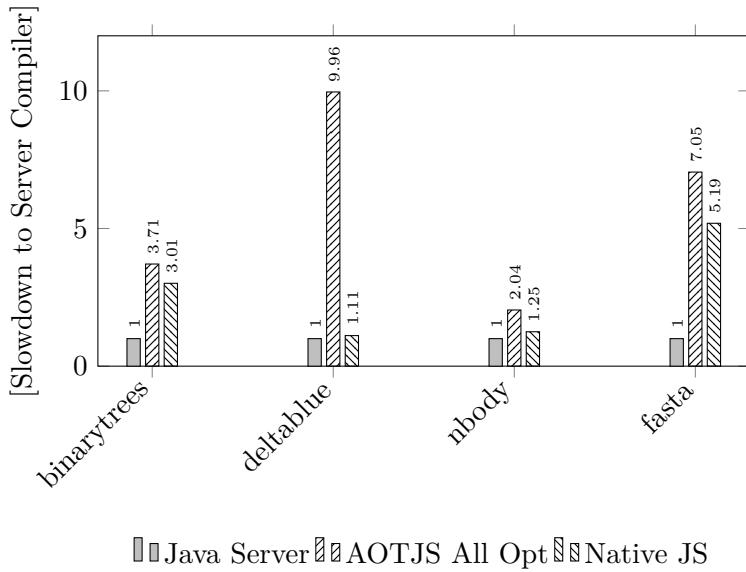


Figure 7.5: Relative performance of Graal AOT JS and native JavaScript compared to the Java HotSpot Server Compiler.

nificant speedups. Nbody has basically zero call- and type-overhead. Linpack, even without exception wrapping which does not compile methods with exception handlers with V8’s optimizing compiler, as they do not optimize exception handlers, is simple enough for V8’s baseline compiler to be optimized properly. Binarytrees has a high call overhead with call stack depths of 15 and more. Compiling it with the optimizing compiler makes a huge difference compared to the full compiler. The "number crunching" benchmarks generally profit less from the optimizations as they rely less on an elaborate type system and exception handling.

The interesting benchmarks of Figure 7.4 are deltablue, JBox2D, SciMark2a and SPECjbb2005. The high slowdown of deltablue is mainly due to the control flow analysis. Deltablue has a set of hot loops with control flow that cannot be fully reduced by the control flow reconstruction optimization.

The SPECjbb2005 benchmark has several performance problems. The main reasons for its slow performance are its large number of memory allocations and the emulation of `long` arithmetic. About 15% of the benchmark’s time is spent in long arithmetic, mostly in addition, multiplication and division. For every computation using `long` values, immutable long objects are created. About 7 – 10% of the benchmarks’s time is spent in the GC. Considering the allocations of `long` objects, it seems that V8’s escape analysis is not able to remove all of them, although inlining of the arithmetic functions should never allocate more than one object, namely the result of the computation. In addition to that, SPECjbb2005 also suffers from performance problems introduced by unstructured control flow.

The SciMark2a and JBox2D benchmarks illustrate further examples of performance penalties for unstructured control flow. The benchmarks spend a lot of time in hot loops, which were not fully analyzed by the control flow reconstruction optimization. A large set of those loops is not identified as structured control flow and thus, compiled with the generic block interpreter. This certainly leaves space for optimizations.

JBox2D is the benchmark that profits most from partial escape analysis. Inlining of small functions and constructors removes many object allocations.

Performance slowdown to code compiled with the server compiler ranges from $2.04x$ to $18x$. However, for many cases this speed is certainly enough, especially for code that runs in websites. The slowdown compared to handwritten JavaScript ranges from $1.3x$ to $9x$ and can be seen in Figure 7.5. For small benchmarks the average slowdown is around $2 - 3x$.

Chapter 8

Related Work

This chapter presents related work in the context of decompilation and compilation for the browser. First it presents related work in the domain of structured control flow decompilation. In the second part this chapter presents different prominent approaches that can be compared to Graal AOT JS, although each of them differs from Graal AOT JS in certain design decisions having different impact on run-time performance, JDK conformity and code size. At the end of the chapter we compare the different approaches with Graal AOT JS and show that the presented Java to JavaScript compiler is unique and a novelty.

8.1 Structured Control Flow Reconstruction & Decomposition Existing Approaches

The process of control flow reconstruction is one major part of the process of *decompilation*. De-compilation is the process of transferring a native program to a human readable (and compilable) representation in a high level language. In the following we discuss related work about control flow reconstruction and decompilation. As this thesis presents an AOT compiler from Java bytecode to JavaScript attention lies on approaches from related work that either deal with Java bytecode as the input or approaches handling and reconstructing high level language constructs existing in Java. We presented a novel control flow reconstruction algorithm, therefore special attention lies on the comparison in the reconstruction process and pipeline. The proposed control flow reconstruction algorithm is presented in section 4.2

Work on decompilation of unmanaged language has been done by Cifuentes in her PHD thesis "Reverse Compilation Techniques" [6]. The thesis describes the complete pipeline of a decompiler for the unmanaged language C. The decompiler includes the runtime system, the data flow analysis, the control flow analysis and finally high level language code generation. The thesis clearly motivates and

defines the terms *structured control flow*, *unstructured control flow* as well as *reducibility* in the context of CFGs. Trivially explained, the decompiler is based on a peephole approach for CFG decompilation. Every time a structured sub-graph is encountered, the nodes are grouped and collapsed. The process of collapsing the sub-graph simply replaces the set of nodes with one node. Each previous predecessor of the set of nodes now points to the collapsed node, the same applies for successors. However, this is a very simplistic view on the process, but it is enough to point out the differences to the control flow reconstruction algorithm of Graal AOT JS. Graal AOT JS has different demands on the control flow reconstruction than [6]. On the one hand C does not support exception handling like Java or C# does and on the other hand the loop reconstruction of Graal AOT JS is simpler than in pure decompilers. Loop detection in Graal AOT JS always generates endless loops with `break` for loop exits and `continue` instructions for loop ends. This is the case as Graal AOT JS' major goal is run-time performance and not readability. Every `while(true)` loop is semantically equivalent to a `for` and `do-while` loop.

A novel approach for decompiling Java was given in [40] and [41]. They describe their Java bytecode decompiler *Dava*. Dava focuses on decompiling arbitrary Java bytecode to a structured representation. Dava's restructuring process is built on the *Soot* [62] bytecode optimizing framework. The reconstruction algorithm uses three intermediate representations including a list of unstructured program statements, a CFG and a structure encapsulation tree (SET). The interesting and novel part of the decompilation process is the SET representation. The SET is a tree representing Java source code high-level-language control flow constructs like, e.g., `if-else`, `switch`, `while`. The nodes of the SET tree have references to the code of a given control flow construct in the CFG. This means a `while` node in the SET tree has a body and a condition child node. Every node points to the source code location it represents. The algorithm processes in a top-down approach the entire method. It starts with a SET tree containing only the method body. It then, iteratively, discovers high level Java control flow statements and add corresponding set nodes to the SET tree. Miecznikowski and Hendren [40] state that the key features behind a SET based decompilation are the different structuring stages, where each stage inspects the CFG for a given Java high level control flow construct like, e.g., a subgraph capable of representing an `if-else` construct. Dava was also the first Java bytecode decompiler explicitly supporting exception edges in their CFG intermediate representation. Their examples of unstructured control flow are also easily compilable by Graal AOT JS, however, Graal AOT JS has a different policy for unstructured control flow, dispatching CFG successors with a big switch, instead of duplicating the code. Dava's SET approach is completely different to Graal AOT JS tagging algorithm, also taking into account that Dava has several CFG detection phases where the tagging algorithm only has one phase for detection and one phase for code generation. The papers about Dava do not describe how unstructured control flow and the heuristics for code duplication increase or decrease run-time performance of the generated code and compilation time of the compiler.

Zakai [72] presents in his C/C++ to JavaScript compiler *emscripten* which is based on LLVM [39] similar problems like Graal AOT JS has with the control flow reconstruction, although unmanaged languages normally have less exception edges in the IR than Java. Emscripten’s control flow reconstruction algorithm called *relooper* works on LLVM basic blocks with labels and is a fix point based approach incrementally improving the result. Compared to Graal AOT JS’s tagging algorithm for control flow, the relooper algorithm works on CFG basic blocks and not IR nodes directly. The relooper performs several iterations to incrementally improve the result of restructuring until a fixpoint is reached, whereas graph tagging is one linear phase over the CFG.

For further background about the performance of general Java bytecode decompilers we refer to [24], where Hamilton and Danicic conducted several benchmarks for different Java decompilers. Their work shows that although Java is described in literature as a language which is easy to decompile this is not entirely true. A lot of decompilers are not able to perform a correct decompilation for all benchmarks, as they relied on bytecode to be generated by the Oracle Java compiler.

8.2 Java to JavaScript Translation

Since 2007 there has been a vast performance increase of JavaScript VMs. In [42] the authors of Mozilla’s ION Monkey just-in-time compiler [43] maintain a snapshot of performance evaluations of different JavaScript VMs. The increasing importance of JavaScript has led many research groups to work on JavaScript VMs and compilers.

For a list of languages that have cross compilers for JavaScript see [30]. Nearly every prominent language has a representative in there.

There are various different approaches from literature featuring some kind of cross-compilation or interpretation to / of JavaScript. To motivate a clean distinction between those different approaches we propose a simple classification method for them: the kind of compilation or interpretation and the point in time. Systems may either perform the transformation offline or online or in a hybrid mode they either compile code or interpret it.

Table 8.1 gives an overview of the features of the presented compilers in comparison to the Graal AOT JS.

8.2.1 Offline Transformation

Approaches performing the transformation offline typically compile the source language to the target language ahead-of-time or compile the runtime of a certain language to the target language. Those approaches typically offer competitive performance, but often lack compatibility problems with the target platform. Typical problems of approaches featuring offline transformation is the compilation of the JDK. The JDK contains thousands of classes, making it impossible to compile the entire JDK to the browser, from the perspective of code size. Therefore a common solution is to implement subsets of the JDK in JavaScript, and compile only user applications to JS. This often also means that no the entire JDK is supported, as it is not fully implemented in JavaScript. This is a shortcoming of many AOT to JavaScript compilers.

Ahead-of-time compilation is the traditional way for cross-compilation. A typical approach to tackle cross compilation establishes a mapping of features from the source to the target language. The properties of the feature mapping is fully explored and patterns for cross-compilation are established. The compilation happens offline, without the demands of, e.g., a JIT compiler, on short compilation time.

Emscripten Emscripten [72] is Mozilla’s back end for the Low-Level-Virtual-Machine (LLVM) [39] that generates JavaScript code from LLVM assembly. The uniqueness of this approach is the usage of ASM.js [43]. ASM.js is a low-level subset of JavaScript that can be easily optimized as it allows the executing JavaScript VM to specialize on 32 bit integers for expressions. Emscripten is mainly targeting the LLVM C/C++ front end *Clang* [38]. Their approach tries to produce fast and small JavaScript code. For the compilation of Java code to JavaScript, Emscripten offers a tool-chain described in [51]. The approach pipelines a Java to C and a C to JavaScript cross-compiler. The Java to C transformation uses the XMLVM [51] and the C to JavaScript transformation emscripten.

Emscripten is targeting a different source language than Graal AOT JS. It uses the Clang front end for LLVM and thus compiles unmanaged C & C++ code which is not comparable to a managed language like Java. Their XMLVM extension fully supports the JDK and also features a way of reducing the code size via so-called *red lists*. However, this approach still suffers from larger code sizes as it lacks a static analysis. Additionally, due to emscripten’s optimizations and its memory model the readability of the generated JavaScript code is reduced. The approach does not use native JavaScript objects and compiles its own GC. Execution of allocation-intensive Java benchmarks such as SPECjbb2005 may lead to performance issues, if the garbage collection is performed in JavaScript rather than by a highly tuned JavaScript VM.

teavm Teavm is a Java bytecodes to JavaScript *transpiler* with a rich compilation pipeline [60]. The compilation pipeline includes bytecode parsing, IR generation in static-single-assignment form, followed by a dependency analysis over the imports to reduce the code size. It applies AOT optimizations, control flow reconstruction and a control flow optimization step. The approach’s *dependency checker* is used to analyze imports of Java classes and to remove unused imports. The analysis works on method granularity so the removal of unused imports decreases the code size dramatically. Teavm’s documentation lacks precise information about AOT optimizations. It just says that “*all major optimizations should be applied*”. Teavm does not offer native support for the JDK, but features a custom compatibility API for it.

Google Web Toolkit (GWT) GWT is Google’s client-sided JavaScript web development toolkit which includes a Java to JavaScript compiler. The main focus of GWT lies on performance and readability of the generated JavaScript code. GWT features its own implementations of `java.util.*` classes, so its JDK usage is limited. The *Closure compiler* [21] that can be optionally used features aggressive optimizations.

Google’s web toolkit operates on the Java source-code level and features a compiler-intrinsic compatibility set of the JDK, which limits the usage prospects of the system. The approach results in better performance because it does not need control flow reconstruction on the source-code level. Future work on Graal AOT JS will tackle this issue and improve the control flow reconstruction optimization.

Whalesong Whalesong [71] is a *Racket* [52] to JavaScript compiler. The approach features different compilation strategies, the fastest being a Racket to bytecode to JavaScript compiler. Slowdown to native Racket ranges from $25\times$ to more than $100\times$. Problems arise from the mapping of Racket’s advanced language constructs to JavaScript like, e.g., preemption and advanced control. The paper presents limited control flow reconstruction optimizations during compilation but it lacks detailed description.

JS_of_ocaml JS_of_ocaml [64] is an *OCaml* [46] bytecode to JavaScript compiler. The presented compiler builds a SSA based IR from OCaml bytecode, applies several optimizations and control flow reconstruction. Peak performance of the generated JavaScript code is in the range of $1.0\times$ to $2\times$ slowdown to handwritten JavaScript code.

8.2.2 Online Transformation

These kind of approaches move the transformation of the source language to the target language into the browser. Typical implementations are interpreters running in the browser or systems offering online just-in-time (JIT) compilation to JavaScript. Systems implementing a JIT often suffer from a long start-up due to feedback driven compilation. Deployment of such systems may be complex as they often rely on specific browsers or elaborated system setups for proper functionality or performance.

Online JIT

Online JIT compilers generate JavaScript code during runtime and call it with the JavaScript function `eval(code)`, which evaluates the code given as parameter in the current context. For the JIT compilation of Java to JavaScript code online, it is necessary to have a mechanism to dynamically load classes. As for AOT compilation several approaches exist, e.g., static analysis, online JIT compilers either load the JDK class files with the compiler itself which is not feasible given the size of the JDK, or they load compilation targets dynamically. Most approaches featuring online JIT load classes from remote web- and / or file servers.

Bck2Brwsr Bck2Brwsr [29] is a Java VM that compiles bytecodes to JavaScript. Its current approach is based on parsing Java classes, extracting meta information and generating JavaScript code. The approach supports just-in-time compilation in the browser via loading required classes from a web server which enables support of the Java reflection API.

Mozilla Shumway Mozilla Shumway[45] is Mozilla's web-native implementation of the small web format (SWF) [1]. Shumway uses HTML5 and JavaScript for interpretation of SWF applications. It features an ActionScript [2] interpreter as well as a just-in-time compiler generating JavaScript code.

8.2.3 Online Bytecode Interpreters

Another category of systems performing the transformation are interpreters executing code in the browser. As every interpreter, such systems generally suffer from bad performance compared to compilers.

Doppio Doppio [63] is a JavaScript-based runtime system for general purpose language support. The system features a large set of low-level API and runtime functionality emulations. The original paper presents a case study of a Java bytecode interpreter running on-top of Doppio. However, the approach aims for general-purpose language support rather than generation of efficient JavaScript code.

The Doppio JVM covers the execution of the entire JDK. It supports threading, the entire file system API and full reflection. However, its Java bytecode interpreter suffers from bad performance. Their presented benchmarks are compared to the HotSpot interpreter which is significantly slower than the server compiler.

Runtime Systems

The third category are systems adding additional runtimes to the browser itself like, e.g., browser plugins for general purpose languages. Those systems rely on the additional runtime systems that are normally not integrated into a clean build of a certain browser language ahead-of-time (AOT) or compile the runtime of a certain language to the target language. Those approaches offer competitive performance, but often lack compatibility problems with the target platform. To overcome those compatibility issues limited hand crafted abstractions of application programming interfaces are designed that are in multiple cases intrinsic to the system and therefore limit the future prospects of the system.

	JDK Support	Threading	Reflection	GC	Control Flow Reconstruction	AOT Optimization	Compilation Source	Dependency Analysis
Graal AOT JS	Compilation	No	limited	JS VM manually	Yes	Yes	Java Bytecode	Yes
Emscripten XMLVM	Compilation	No	Full	Yes	Yes	Yes	Java Bytecode	No
GWT	Custom	No	limited	JS VM	N/A	N/A	Java SourceCode	N/A
teavm	Custom	Yes	limited	JS VM	Yes	N/A	Java Bytecode	Yes
Bck2Bwsr	Custom	No	Full	JS VM	No	No	Java Bytecode	N/A
Doppio JVM	Interpretation	Yes	Full	JS VM	N/A	No	Java Bytecode	No
Shumway	N/A	N/A	N/A	JS VM	Yes	Yes	ActionScript	N/A
WhaleSong	N/A	Yes	N/A	JS VM	Limited	Limited	Racket Source	N/A
JS_of_ocaml	N/A	No	N/A	JS VM	Yes	Yes	OCaml ByteCode	N/A

Table 8.1: Transformation to JavaScript Approaches

N/A indicates that a certain approach is not design to be evaluated to a given property, or literature does not provide detailed information.

Chapter 9

Future Work

This chapter lists possible future work on Graal AOT JS. It presents research ideas that could be realized using Graal AOT JS as a foundation for further development.

Control Flow Reconstruction CF reconstruction is crucial for run-time performance. Therefore future work will include improvements of the control flow restructuring optimization especially for shared exception handlers and staged loops.

Loop Type Detection Currently Graal AOT JS only generates `while(true)` loops, as they are semantically equivalent to any `for` or `while` loop. However, readability is declined if a loop, which is actually a `for` loop, is generated as a `while(true)` loop. If readability is desired in the future, detecting the type of a loop (pre-tested, post-tested, infinite) and using this information during code generation would increase readability.

Run-time Performance As for any compiler, run-time performance and memory usage of the generated code is a major concern. Code generated by Graal AOT JS, for Java applications, still is $2x$ to $20x$ slower than native Java code on the HotSpot server compiler. A general slowdown of code generated by Graal AOT JS below $5x$ is certainly desirable. Future work on Graal AOT JS therefore will be about new optimizations for the generated JavaScript code and improvements of the existing code generator.

Code Size Code size is still very high even for simple applications. The largest part of the code size can be attributed to the initial heap initialization. The usage of a compression algorithm to encode the initial heap data would be interesting. The reduction in code size for big applications like Truffle interpreters would certainly be high.

Truffle Graal AOT JS’s capabilities for the support of runtime compilation for Truffle as presented in [70] could be explored. Such a system will require Graal itself to be compiled to JavaScript, which is a challenge in terms of code size rather than in terms of performance.

Chapter 10

Conclusion

This thesis presents a novel approach for AOT compilation of Java to JavaScript. The approach is able to produce structured JavaScript code from possibly unstructured Java bytecode. The thesis presents a novel algorithm for structured control flow reconstruction yielding a reasonable amount of restructured control flow. The evaluation of the approach on a set of benchmarks shows that AOT compilation from a just-in-time compiler's IR to JavaScript is feasible, but there is some slowdown to hand-written code depending on the benchmark.

Acknowledgments

Most of my gratitude belongs to Lukas Stadler for his constant feedback and the endless discussions about all aspects of this thesis. He always had impressive ideas and valuable comments on each and every issue. Then, I would also especially like to thank my second advisor Matthias Grimmer for his feedback on this thesis. I would then also like to thank my professor Hanspeter Mössenböck and the manager of the Graal project Thomas Würthinger who always had constructive feedback on my work.

This work was performed in a research cooperation with, and supported by, Oracle.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

List of Figures

2.1	HotSpot Client Compiler IR	7
2.2	Graal Compilation Tiers	9
2.3	Graal Eco System	11
3.1	Graal AOT JS System Overview	12
3.2	Graal AOT JS System Architecture Overview	15
4.1	Java to JavaScript code generation example.	19
4.2	Nodes to braces matching.	20
4.3	Generic CFG Patterns (Modified: originally presented by [6]).	21
4.4	Unstructured If-Goto with labeled blocks	22
4.5	If Graal IR	23
4.6	Switch Graal IR	24
4.7	Fall through Switch Graal IR	24
4.8	Loop Single Exit Graal IR	27
4.9	Loop Multi Exit Graal IR	27
4.10	Multi-Exit Multi-Next Node Loop Graal IR	28
4.11	Shared Exception Handler Graal IR	29
4.12	Compound Conditional Graal IR	30
4.13	Final Paths	31
4.14	Tagging Algorithm Link	31
4.15	Graal AOT Tagging Algorithm Application	34
4.16	Control flow graph block interpreter.	35
4.17	Graal AOT JS Control Flow Reconstruction Pipeline	36
4.18	Graal AOT JS Compound Condition Phase Example	40
4.19	Graal AOT JS Unwinding If Example	40
4.20	Structured control flow: multiple loop exits are merged.	41
4.21	Graal AOT JS Merge Loop Exit Phase Example	42
5.1	AOT JS Inheritance Tree.	46

7.1	Percentage of inlined nodes using the SSA Node Inlining Optimization in relation to all SSA nodes.	60
7.2	Number of restructured split nodes and number of methods with unstructured control flow compared to all methods.	61
7.3	Code Size of the generated JavaScript of Graal AOT JS.	62
7.4	Relative performance of Graal AOT JS compared to the Java HotSpot Server Compiler.	63
7.5	Relative performance of Graal AOT JS and native JavaScript compared to the Java HotSpot Server Compiler.	64

List of Tables

5.1 JSIntrinsify Escape Sequences	50
7.1 Total code sizes of the presented benchmarks.	62
8.1 Transformation to JavaScript Approaches	73

Listings

4.1	Structured Goto C	25
4.2	Control flow tagging algorithm.	32
4.3	CF Reconstruction Full Example	38
4.4	Generated JavaScript for a Compound Conditional	38
4.5	Generated JavaScript for an Unwind If	39
4.6	Generated JavaScript for multiple Loop Exits	39
5.1	Person Class Java	45
5.2	Person Class JavaScript	45
5.3	Graal AOT JS Dynamic Type Check	48
5.4	Java Simple Exception Handler	49
5.5	JavaScript Simple Exception Handler	49
5.6	Graal AOT JS Unsafe Load JavaScript	51
6.1	Exception wrapping optimization example (JavaScript)	54
6.2	Inlining Example Java	55
6.3	Inlining Example JavaScript without inlining of SSA nodes.	56
6.4	Inlining Example JavaScript with inlining of SSA nodes.	57

Bibliography

- [1] Adobe. Small Web Format, 2015. URL <http://www.adobe.com/devnet/swf.html>.
- [2] Adobe. ActionScript, 2015. URL <http://www.adobe.com/devnet/actionscript/documentation.html>.
- [3] David F Bacon, Susan L Graham, and Oliver J Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, 1994.
- [4] Hans-J. Boehm. A garbage collector for C and C++, 2015. URL <http://www.hboehm.info/gc/>.
- [5] Box2d. Box2D Physics Engine, 2015. URL <http://box2d.org/>.
- [6] Cristina Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.
- [7] CLBG. Computer Language Benchmark Game, 2015. URL <http://benchmarksgame.alioth.debian.org/>.
- [8] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–257. ACM Press, 1995. doi: 10.1145/207110.207154.
- [9] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. pages 35–49. ACM Press, 1995. doi: 10.1145/202529.202534.
- [10] Clifford Noel Click, Jr. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995. URL <http://hdl.handle.net/1911/16807>.

- [11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. doi: 10.1145/115372.115320.
- [12] DB. DeltaBlue Benchmark, 2015. URL <https://github.com/xxgreg/deltablue>.
- [13] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. ACM Press, 2013.
- [14] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the ACM Workshop on Virtual Machines and Intermediate Languages*, pages 1–10. ACM Press, 2013. doi: 10.1145/2542142.2542143.
- [15] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without regret: Reducing deoptimization meta-data in the graal compiler. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 187–193. ACM, 2014. doi: 10.1145/2647508.2647521.
- [16] ECMA. ecmascript, 2015. URL <http://www.ecmascript.org/docs.php>.
- [17] ECMA. Ecmascript-262 edition 6, 2015. URL http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts.
- [18] Ana Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *In Proceedings of the International Conference on Computer Languages*, pages 229–240. IEEE Computer Society Press, 1994. doi: 10.1109/ICCL.1994.288377.
- [19] Google. Crankshaft: V8’s optimizing compiler, 2012. URL <http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html>.
- [20] Google. V8 JavaScript Engine, 2012. URL <http://code.google.com/p/v8/>.
- [21] Google. Closure Compiler, 2015. URL <https://developers.google.com/closure/compiler/>.

-
- [22] Google. Closure long library, 2015. URL <https://code.google.com/p/closure-library/source/browse/closure/goog/math/long.js?spec=svn17772b52769e72755a566c3676d455966ed545d5&r=17772b52769e72755a566c3676d455966ed545d5>.
 - [23] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Java Language Specification, Java SE 8 Edition*, 2015. URL <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>.
 - [24] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 129–136. IEEE, 2009.
 - [25] Matthew S Hecht and Jeffrey D Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250. ACM, 1972.
 - [26] Matthew S Hecht and Jeffrey D Ullman. Characterizations of reducible flow graphs. *Journal of the ACM (JACM)*, 21(3):367–375, 1974.
 - [27] Urs Hözle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM, 1992.
 - [28] IEEE. The IEEE 754 Standard for Binary Floating-Point Arithmetic, 2015. URL <http://grouper.ieee.org/groups/754/>.
 - [29] Jaroslav Tulach. DukeScript: Bck2Brwsr VM, 2015. URL <http://wiki.apidesign.org/wiki/Bck2Brwsr>.
 - [30] jashkenas. List of Languages that compile to JavaScript, 2015. URL <https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS>.
 - [31] JBox2D. Piston Bechnmark, 2015. URL <http://www.jbox2d.org/>.
 - [32] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspotTM client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7, 2008.

- [33] Tímea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.
- [34] David Leopoldseder, Lukas Stadler, Christian Wimmer, and Hanspeter Mössenböck. Java-to-JavaScript Translation via Structured Control Flow Reconstruction of Compiler IR. 2015. doi: 10.1145/2816707.2816715.
- [35] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*, 2015. URL <https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>.
- [36] Linpack. Linpack Benchmark, 2015. URL <http://www.netlib.org/benchmark/linpackjava/>.
- [37] Lisperator. UglifyJs2, 2015. URL <http://lisperator.net/uglifyjs/>.
- [38] LLVM. Clang, 2015. URL <http://clang.llvm.org/>.
- [39] LLVM. Low-Level Virtual Machine, 2015. URL <http://llvm.org/>.
- [40] Jerome Miecznikowski and Laurie Hendren. Decompiling java using staged encapsulation. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 368–374. IEEE, 2001.
- [41] Jerome Miecznikowski and Laurie Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *Compiler Construction*, volume 2304, pages 111–127. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45937-5_10.
- [42] Mozilla. Are We Fast Yet?, 2015. URL <http://arewefastyet.com/>.
- [43] Mozilla. ASM.js, 2015. URL <http://asmjs.org/>.
- [44] Mozilla. Developer Network (MDN): JavaScript, 2015. URL <https://developer.mozilla.org/de/docs/Web/JavaScript>.
- [45] Mozilla. Shumway, 2015. URL <http://mozilla.github.io/shumway/>.
- [46] OCaml. The OCaml Language, 2015. URL <https://ocaml.org/>.

- [47] OpenJDK. Graal, 2015. URL <http://openjdk.java.net/projects/graal/>.
- [48] Oracle. Hotspot jvm, 2015. URL <http://openjdk.java.net/groups/hotspot/>.
- [49] Oracle. Oracle and Sun Microsystems, 2015. URL <http://www.oracle.com/us/sun/index.html>.
- [50] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, 2001. URL <http://dl.acm.org.acmdigitallibraryportal.han.ubl.jku.at/citation.cfm?id=1267847.1267848>.
- [51] Arno Puder, Victor Woeltjen, and Alon Zakai. Cross-compiling Java to JavaScript via tool-chaining. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 25–34. ACM Press, 2013. doi: 10.1145/2500828.2500831.
- [52] Racket. The Racket Language, 2015. URL <http://racket-lang.org/>.
- [53] Mark Reinhold. The Secret History and Tragic Fate of sun.misc.unsafe, 2015. URL <http://www.oracle.com/technetwork/java/javase/community/jlssessions-2015-2633029.html>.
- [54] SCI2. SciMark 2 Benchmark, 2015. URL <http://math.nist.gov/scimark2/>.
- [55] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 46(1):17–30, 2011.
- [56] Spec. SPECjbb2005 Java Server Benchmark, 2015. URL <https://www.spec.org/jbb2005/>.
- [57] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings of the 4th Workshop on Scala*, page 9. ACM, 2013. doi: 10.1145/2489837.2489846.
- [58] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of the International Symposium on Code Generation and Optimization*, page 165. ACM, 2014. doi: 10.1145/2544137.2544157.

- [59] Codruț Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. Safe and efficient hybrid memory management for java. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, pages 81–92. ACM, 2015.
- [60] TEA. TEA VM, 2015. URL <http://teavm.org/>.
- [61] TIOBE. TIOBE Index September 2015, 2015. URL <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.
- [62] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.
- [63] John Vilk and Emery D Berger. Doppio: breaking the browser language barrier. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 508–518. ACM, 2014. doi: 10.1145/2594291.2594293.
- [64] Jérôme Vouillon and Vincent Balat. From bytecode to javascript: the js_of_ocaml compiler. *Software: Practice and Experience*, 44(8):951–972, 2014.
- [65] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, 2000.
- [66] Henry S. Warren. *Hacker's delight*. Addison-Wesley, Upper Saddle River, NJ, 2nd ed. edition, 2013. ISBN 0321842685.
- [67] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 170–179. ACM, 2010.
- [68] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5-6), 2009. doi: 10.1016/j.scico.2009.01.002.
- [69] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the Dynamic Languages Symposium*, page 73. ACM Press, 2012. doi: 10.1145/2384577.2384587.

- [70] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 187–204, 2013. doi: 10.1145/2509578.2509581.
- [71] Danny Yoo and Shriram Krishnamurthi. Whalesong: Running racket in the browser. In *Proceedings of the Dynamic Languages Symposium*, pages 97–108. ACM, 2013.
- [72] Alon Zakai. Emscripten: An LLVM-to-JavaScript compiler. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 301–312. ACM Press, 2011. doi: 10.1145/2048147.2048224.

Curriculum Vitae

Personal Information

Name	David, Leopoldseder
Address	Brucknerstraße 28/5, 4020 Linz
Telephone	+43 676 9268779
Email	david.leo@web.de
Nationality	Austrian
Date of Birth	February 29 1992
Gender	male



Professional Experience

03/2014–today	Student Assistant, Institute for System Software, Linz.
07/2013–	ekey biometric systems GmbH, Internship (BSc Thesis: "Driver Development for a Finger-
09/2013	print Swipe Sensor: Incorporating Upek's TCS4H into the AT91SAM9G20").

Education

2011–2014	BSc in Computer Science, Johannes Kepler University, Linz, Austria.
2002–2010	BG / BRG Freistadt, Austria.
1998–2002	Volksschule Weitersfelden, Austria.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am

David Leopoldseder, BSc.