DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

On implementing multiple pluggable dynamic language frontends on the JVM, using the Nashorn runtime

ANDREAS GABRILSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

On implementing multiple pluggable dynamic language frontends on the JVM, using the Nashorn runtime

Om att implementera flera dynamiska språk-frontends på JVM med användning av Nashorns exekveringsmiljö

July 2015

Master's thesis in Computer Science
Oracle Sweden AB
Andreas Gabrielsson (andresgus@kth.se)
Supervisors at Oracle: Marcus Lagergren and Attilla Szegedi
Supervisor at KTH: Per Austrin
Examiner: Johan Håstad

Abstract

Nashorn is a JavaScript engine that compiles JavaScript source code to Java bytecode and executes it on a Java Virtual Machine. The new bytecode instruction `invokedynamic` that was introduced in Java 7 to make it easier for dynamic languages to handle linking at runtime is used frequently by Nashorn. Nashorn also has a type system that optimizes the code by using primitive bytecode instructions where possible. They are known to be the fastest implementations for particular operations.

Either types are proved statically or a method called *optimistic type guessing* is used. That means that expressions are assumed to have an `int` value, the narrowest and fastest possible type, until that assumption proves to be wrong. When that happens, the code is deoptimized to use types that can hold the current value.

In this thesis a new architecture for Nashorn is presented that makes Nashorn's type system reusable to other dynamic language implementations. The solution is an intermediate representation very similar to bytecode but with untyped instructions. It is referred to as Nashorn bytecode in this thesis.

A TypeScript front-end has been implemented on top of Nashorn's current architecture. TypeScript is a language that is very similar to JavaScript with the main difference being that it has type annotations. Performance measurements which show that the type annotations can be used to improve the performance of the type system are also presented in this thesis. The results show that it indeed has an impact but that it is not as big as anticipated.

Referat

Nashorn är en JavaScriptmotor som kompilerar JavaScriptkod till Java bytekod och exekverar den på en Java Virtuell Maskin. Nashorn använder sig av den nya bytekodinstruktionen `invokedynamic` som introducerades i Java 7 för att göra det lättare för dynamiska språk att hantera dynamisk länkning. I Nashorn finns ett typsystem som optimerar koden genom att i så stor utsträckning som möjligt använda de primitiva bytekodinstruktioner som är kända för att vara de snabbaste implementationerna för specifika operationer. Antingen bevisas typen för ett uttryck statiskt om det är möjligt eller så används något som kallas för *optimistisk typgissning*. Det innebär att uttrycket antas ha typen `int`, den kompaktaste och snabbaste typen, ända tills det antagandet visar sig vara falskt. När det händer deoptimeras koden med typer som kan hålla det nuvarande värdet.

I det här dokumentet presenteras en ny arkitektur för Nashorn som gör det möjligt för andra dynamiska språk att återanvända Nashorns typsystem för bättre prestanda. Lösningen är en intermediate representation som påminner om bytekod men som är uttökat men otypade instruktioner. I det här dokumentet refereras den som Nashorn bytekod.

En TypeScript front-end har implementerats ovanpå Nashorns nuvarande arkitektur. TypeScript är ett språk som liknar JavaScript på många sätt, den största skillnaden är att det har typannoteringar. Prestandamätningar som visar att typannoteringarna kan användas för att förbättra prestandan av Nashorns typsystem presenteras i det här dokumentet. Resultaten visar att typannoteringar kan användas för att förbättra prestandan men de har inte så stor inverkan som förväntat.

Contents

Chapter 1

Introduction

The Java Virtual Machone (JVM) provides a solid runtime platform that is convenient to use by many languages, not only Java. It already contains well performing garbage collectors and optimizing Just-In-Time (JIT) compilers which have been fine-tuned for man decades. On top of all this, Java is also platform-independent. Therefore, a lot less code is required to implement a platform-independent language on the JVM than a native runtime [11].

This thesis is about implementing dynamic languages and there are several issues with implementing such languages on the JVM. For instance, JVM bytecode is strongly typed while dynamic languages typically are not. Dynamic languages often require linking at runtime and there has not been any mechanism for that prior to the release of Java 7 when `invokedynamic` was introduced [20].

Despite this, many attempts have been made during the years to implement dynamic languages on top of the JVM due to the good characteristics mentioned above [20]. The dynamic linking previously had to be solved by using something like a virtual dispatch table and the `invokeinterface` bytecode and when a call site needed to be relinked the method containing the call site would have to be recompiled. This is a solution that the JVM can not infer enough about to optimize well.

1.1 Background

1.1.1 Bytecode and the JVM

Initially the JVM was built to execute Java code. The Java code gets compiled to bytecode which is then executed on the JVM [14]. Because of many good characteristics of the JVM, bytecode has become a common compile target for other languages as well. Scala, Groovy and Clojure are all compiled to bytecode and executed on the JVM. Compilers and runtime environments that compile to bytecode also exists for other languages that were not initially designed to be executed on the JVM, such as Ruby and Python.

There are several reasons why the JVM is such a popular compile target. The JVM is available on most of the common operating systems out there and on a variety of processor architectures. It also contains high performing JIT-compilers and garbage collectors which makes it possible to execute code efficiently. Making use of these good characteristics relieves the runtime developers from a big part of the work and makes it possible to implement a runtime with less code than a native implementation would require [14].

The JVM is a stack machine, meaning that all bytecode instruction operates on a stack instead of registers, which is common for physical processors [14]. Depending on the instruction zero or more values are popped from the stack and if the instruction has a return value it is pushed to the top of the stack. E.g. the bytecode instruction `iadd` pops the two topmost `int` values of the stack and pushes the sum back at the top.

Like the `iadd` instruction many bytecode instructions specify the type they operate

on. There are other add instructions used for other types, `fadd`, `ladd` and `dadd` are also available, they operate on `float`, `long` and `double` types. Bytecode also has builtin support for arrays of all the primitive types. In Java and bytecode the primitive types are the following:

- byte
- char
- short
- boolean
- int
- long
- float
- double
- references to Objects

The `byte`, `char`, `short` and `boolean` types do not have any arithmetic instructions but are represented as `int`s on the stack. Int instructions are also used to operate on them but they can be converted to their more memory efficient representation for storage in, for instance, arrays. The `reference` type is used for references to instances of `java.lang.Object` and its subclasses (every other class is a subclass of `java.lang.Object`). The listed types are commonly referred to as the *primitive types* of Java/bytecode and differ from class instances in the sense that they can be placed on the stack, they have no methods and are not subclasses of or assignable to `java.lang.Object`.

1.1.2 Dynamic languages on the JVM

The term dynamic language is vaguely defined but generally refers to languages that perform actions in runtime that more static programming languages perform at compile time. What actions those are differs from language to language.

Probably the most common is dynamic linking and dynamic typing. Those are key concepts of many common dynamic programming languages such as JavaScript [9], Ruby [5], Groovy [10] and Clojure [8]. All those languages have implementations that compile the source code to bytecode and execute it on the JVM.

With the new Java bytecode instruction `invokedynamic` came the tools needed for dynamic linking on the JVM, more about that in Section 1.1.3. Dynamic typing is however still a big issue on the JVM because of the typed nature of bytecode. The types are seldom known at compile time and can usually change at runtime. In dynamic runtime implementations, this is usually solved by using `java.lang.Object` as type for everything since all object types can be assigned to it and instead of primitives their boxed types are used, e.g. `java.lang.Integer` and `java.lang.Long`. While that works fine, it has a lot worse performance than using the primitive types.

All these languages also implement some kind of concept for *closure*. A closure is a first-class function that can access variables that were accessible in the lexical context where the function was declared. The set of accessible variables of a closure is often referred to as *the lexical scope*, or only *the scope*, of the function. In JavaScript all functions are closures, Listing 1.1 shows an example of how they can be used, the function call on line 8 will print `Hello World!` since `a` is accessible in the lexical context the function is declared in.

```
1 function parent() {
2     var a = "Hello World!";
3     return function () {
4         return a;
5     }
6 }
7 var nested = parent();
8 print(nested()); // prints "Hello World!"
```

Listing 1.1: Example of closures in JavaScript

One consequence of closures is that local variables cannot be stored on the stack as usual since they can live on after the function has returned, like variable `a` on line 2 does. This can be solved by, for example, storing variables in a scope object when needed, see Chapter 2 for more details.

The other dynamic languages mentioned also have support for closures. In Ruby ordinary methods are not closures but it has special functions called lambdas and procs which can be used as closures [5]. Groovy has a separate language construct for closures[10] while regular functions are not closures.

TypeScript

The TypeScript programming language is a superset of JavaScript and is typically compiled to JavaScript [16]. It does not have its own runtime environment but is compiled to JavaScript and executed in a JavaScript runtime environment.

TypeScript was designed to make it easier to build big and complex JavaScript applications and does so by adding new language constructs such as modules, classes, interfaces and types [4]. Since the code is compiled to JavaScript and executed in a JavaScript runtime environment these new constructs are mainly to be considered syntactic sugar. They just provide a different way to express what can already be expressed in JavaScript [16, 4, 15]. It also means that TypeScript has the same set of builtin functions as JavaScript [16].

The TypeScript compiler however performs type checking which a JavaScript compiler does not [16, 4, 15]. Typing variables and functions is optional, if no type is specified the compiler infers the type for a variable or a function from the assigned expression or return statements if possible. If that is not possible the default type is `any` which means that that expression bypasses the type checking and behaves exactly like it would in JavaScript.

The type inference has the effect that not all valid JavaScript is valid TypeScript even though that is commonly claimed [15, 4]. For example the code in Listing 1.2 is valid JavaScript but in TypeScript `a` is inferred to be of the primitive type `number` since `5` is assigned to it when it is declared. So assigning a string to `a` on line 2 will raise a type error at compile time.

```
1 var a = 5;
2 a = "Hello World!";
```

Listing 1.2: Example of code that is valid JavaScript but not TypeScript

1.1.3 Invokedynamic

In Java 7 the `invokedynamic` bytecode instruction was introduced to tackle the problem with dynamic linking for dynamic languages [20, 17].

It gives full control of the linkage to the developers to handle at runtime and does so in a way that does not require the calling method to be recompiled and replaced [17].

Every `invokedynamic` instruction specifies a *bootstrap method* that returns a `java.lang.invoke.CallSite` instance. The `CallSite` instance contains a reference to the method that should be invoked in the form of a `java.lang.invoke.MethodHandle` in-

stance [18]. The first time a `invokedynamic` instruction is executed, the bootstrap method is invoked and the instruction is linked to the returned `CallSite`. For all consecutive executions of an `invokedynamic` instruction, the linked method is invoked directly without needing to bootstrap it again [18, 17]. It remains linked until the `CallSite` gets invalidated, that can happen for several reasons, for example if the target `MethodHandle` is invalidated manually or because of a guard that fails (an optional check that is executed before each invocation).

There are different kinds of `CallSite` classes available, the two most relevant being `ConstantCallSite` and `MutableCallSite`. They differ in the sense that the target of a `ConstantCallSite` can never be changed [18] while a `MutableCallSite`'s target can. The immutability of `ConstantCallSite` allows the JVM to optimize the call site more aggressively since it knows it will never have to deoptimize it due to the target changing. It is also possible to create custom `CallSite` classes by extending one of the available classes [18].

A `MethodHandle` is in many ways Java's equivalent of a function pointer in C. It can be passed around as a regular variable just like a function pointer, invoked like any other function and it can be exchanged without any need to recompile the class or function that invokes it [18, 20].

A linked `invokedynamic` instruction is something that the JVM understands and that it can optimize and inline with the same mechanisms as for methods invoked by any of the static invoke instructions. When the `CallSite` changes it uses its standard deoptimization mechanisms and can then optimize the newly linked method [20, 17].

1.1.4 Compiler design

This section covers some basic concepts of compiler design that are referred to throughout this thesis.

A compiler is typically separated into two main parts, a front-end and a back-end, with an intermediate representation (IR) (or intermediate language) in between them [1]. The purpose of the compiler front-end is to compile the source language into the IR according to the language's syntax and semantics. The compiler back-end compiles the IR into a language that is executable on a specific platform. Examples of such languages are X86 machine code for an Intel processor and Java bytecode, for the JVM.

Front-end

The compiler front-end typically consists of syntactic and semantic analysis [1].

Syntactic analysis is the process of converting the source code to a representation that is more suitable for a computer to process. The syntactic analysis is generally divided into lexical analysis and parsing. The lexer converts the stream of characters that is the source code to a stream of tokens to be processed by the parser.

An Abstract syntax tree (AST) is what is typically output by the syntactic analysis and used as representation in the semantic analysis [1]. The AST is directly derived from the syntax of the programming language and contains language dependant nodes such as functions, loops, if statements and different kinds of expressions [1]. Unlike the Concrete syntax tree which contains all information from the grammar the AST contains only information that is relevant for the compiler. Meaning that semicolons, parentheses and other tokens needed only to parse the source code are not explicitly represented in the AST, only implicitly in the sense that they effect how the AST is constructed [1].

The semantic analysis typically outputs an IR derived from the AST and relates all language dependant syntactic structures with their language independent meaning, expressed in the intermediate language. For example, if statements could be expressed as conditional jumps, symbolic references as memory reads or writes etc., depending on what is supported in the intermediate language.

Intermediate representation

An IR is typically a lower level representation than an AST but higher level than for instance an assembly language [1]. Such a representation is more suitable for transformations like control flow analysis and data flow analysis [1].

An IR can be designed in different ways depending on its purpose. The are, for instance, general purpose IRs that target any programming language and any platform. An example of such an IR is LLVM that is described in more detail in Section 1.1.6.

The main benefit of such an IR is that by implementing a front-end for a specific language one gets the ability to execute the language on all platforms that there are back-ends for. The same applies the other way around, by implementing a single back-end it is possible to execute all languages that there are front-ends for on that platform. This approach is in many ways the school book example of how to design a compiler [1] but there might be reasons to use other approaches.

JRuby 9000, for instance, has an IR that targets the needs of Ruby, more details in Section 1.1.7.

Back-end

The purpose of the compiler back-end is, as stated above, to compile the IR into a language supported by a specific platform. Because of that the back-end has knowledge of that specific platforms weaknesses and benefits and can perform platform specific optimizations.

The output of the back-end is the language that can be executed on the targeted platform, typically in machine code form or if the platform is the JVM, Java bytecode.

1.1.5 Nashorn

Nashorn is a JavaScript engine that executes JavaScript on top of the JVM. Unlike its predecessor Rhino, it relies heavily on the new bytecode instruction `invokedynamic` that was introduced in Java 7 [20]. Nashorn is, like Rhino, 100% implemented in Java.

Nashorn has a sophisticated type system that mainly performs two actions. First, it tries to infer types of expressions statically for as many expressions as possible. In most cases however, that will not be possible to do. If that is the case, Nashorn resorts to a concept called *optimistic type guessing*. That basically means that expressions are assumed to be of primitive types, preferably `int`s. If that assumption turns out to be wrong, the function is recompiled with new types that can hold the current value.

JavaScript has many characteristics in common with other dynamic languages, such as dynamic linking and dynamic typing. Because of that the intention is to turn Nashorn into a generic runtime library for dynamic languages rather than a runtime for JavaScript only. While the JavaScript front-end was implemented by Oracle, they do not at the moment aim to implement front-ends for any other languages on top of Nashorn but rather to provide a tool box for others to use.

1.1.6 LLVM

LLVM was initially a research project with the intent to design a reusable intermediate representation for a compiler [13], making it possible to implement compiler front-ends that can be executed on multiple processor architectures by making use of the already existing back-ends. At the same time it is making it possible to implement compiler back-ends that automatically can be used by the already implemented front-ends. Initially the name was an acronym of *Low Level Virtual Machine* but LLVM is in fact not a virtual machine so nowadays the name is not considered an acronym but rather a name [13]. There is support for a variety of languages and architectures on the LLVM platform, many of them are listed on the official website.[1]

[1] LLVM's official website: `http://www.llvm.org`

1.1.7 JRuby 9000

JRuby is a Ruby implementation[2] that compiles Ruby code to bytecode and executes it on the JVM. JRuby 9000 is the new upcoming version of JRuby and it introduces concepts that are relevant to this thesis.

One of the main differences in the new version is that JRuby's runtime and compiler has gone through major design changes. Previously an abstract syntax tree (AST) was used as internal representation throughout the compiler. In the new version the AST is transformed to an intermediate representation (IR) for further optimizations before bytecode generation and/or interpretation [3]. Optimizations such as dead code eliminations, liveness analysis and other control and data flow optimizations are easier to perform on a lower level IR representation than on an AST. Söderberg et al. show that the data and control flow graphs needed to perform such optimizations efficiently can actually be constructed from an AST [21]. However, they also say that constructing data and control flow graphs from an AST is mostly useful for constructing text editor tools where it is beneficial to keep the representation as close to the source code as possible rather than actual compilers because the construction of the graphs is more complicated and not as efficient.

Another reason for using an intermediate representation is that the difference from traditional compiler design practices are reduced [3, 1].

JRuby's IR does not claim or intend to be a general purpose IR like LLVM's does [3]. Although it would probably be possible to fit other languages on top of JRuby's IR, it is written for Ruby and has some built in constructs and semantics that are Ruby specific. For example it has builtin support for Ruby's scope rules with concepts such as class scope, global scope and local scope which are treated according to JRuby semantics.

1.2 Problem

1.2.1 Motivation

Interest in running dynamic languages on the JVM has increased for the past years [20] and `invokedynamic` was a big step for the JVM to support implementation of such languages since it made it possible to link dynamically in a way that the JVM can optimize. But despite `invokedynamic` there are still quite big thresholds to implement dynamic languages efficiently. The biggest of them all is probably the dynamic typing which is not as easy to implement on the JVM with good performance.

Dynamic typing is already handled by Nashorn with good performance by the use of optimistic type guessing. If Nashorn's solution were made reusable, that would even further reduce the threshold to implement other dynamic languages with good performance on the JVM.

1.2.2 Statement

What design changes would be required to make the core concepts of Nashorn reusable to implement other dynamic languages on top of? What parts of Nashorn are well designed for reuse and can be included in the new architecture and what parts needs to be redesigned?

A TypeScript frontend should be implemented as a proof-of-concept that the core concepts of Nashorn can be reused and/or extended. Better performance and warmup time is expected for TypeScript compared to JavaScript because of the type annotations and the decreased need to do optimistic type guesses. Because of that, this thesis also includes analysis of how the more statically typed nature of TypeScript affects the warm

[2] JRuby's official website: `http://www.jruby.org`

up times and overall performance of Nashorn compared to pure fully dynamically typed JavaScript.

1.2.3 Goal

The main goal of this thesis is to give recommendations on architectural design changes for Nashorn to make it easier to plug in frontends for other dynamic languages on top of Nashorn.

A second goal is to provide performance analysis and compare how typed TypeScript and weakly typed JavaScript performs on Nashorn.

Chapter 2

Current architecture

Nashorn's architecture consists of three main phases. First is the parser that creates an abstract syntax tree (AST) from the source code. The AST is then kept as internal representation throughout the different phases of the compiler, which is described in more detail in Section 2.2. The compiler is responsible for, for example, optimizing the AST and assigning types to all expressions. In the last two phases it generates bytecode and installs it in the JVM.

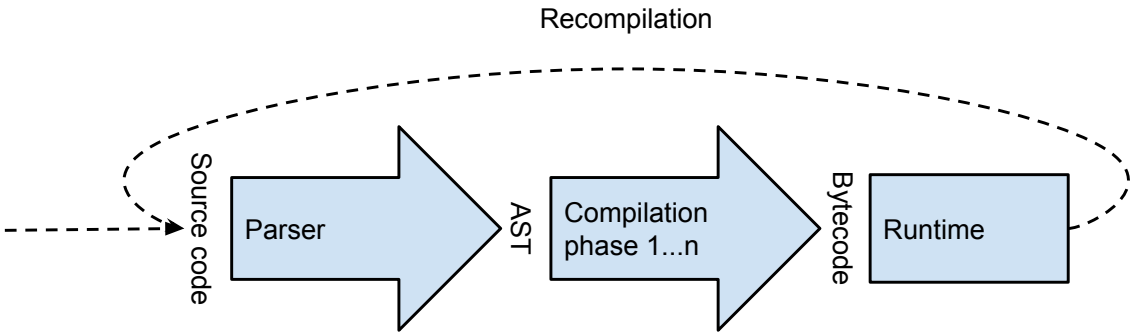


Figure 2.1: Simple overview over Nashorn's architecture

The runtime's main responsibilities are dynamic linking and handling of dynamic types. In many cases it has to trigger recompilation of certain functions, in which case the function is recompiled from the JavaScript source code.

2.1 Internal representation

The AST in Nashorn is built up by nodes that each represent language constructs in JavaScript, e.g. function nodes, loop nodes, if nodes and different kinds of expression nodes.

```
1 var a = 5;
2 print(a);
```

Listing 2.1: A simple JavaScript program

A JavaScript program itself is wrapped into a virtual function node, with the same semantics as a regular JavaScript function. Because of this, the root of every AST is a function node. The value of the last statement is returned from the function like the specification says it should [9]. The function that wraps the program is referred to as the *program function* throughout this thesis. An AST representation of the simple JavaScript program in Listing 2.1 is shown in Figure 2.2;

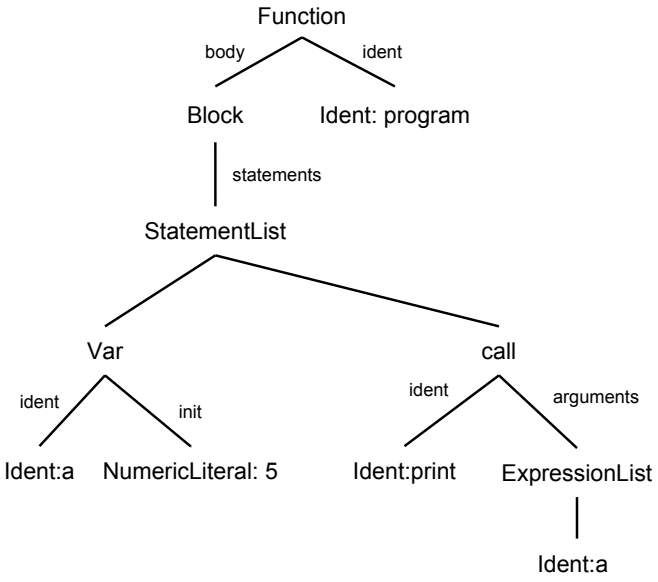


Figure 2.2: An AST representation of the JavaScript program in Listing 2.1

2.2 Compiler

The compiler consists of several phases that each take an AST as input and output a transformed AST. The different phases are required to be executed in order since one phase could depend on the results from a previous phase.

Each of the compilation phases will be described in more details in the following sections. The different phases are, in order, the following:

1. Constant folding
2. Control flow lowering
3. Program point calculation
4. Builtins transformation
5. Function splitting
6. Symbol assignment
7. Scope depth computation
8. Optimistic type assignment
9. Local variable type calculation
10. Bytecode generation
11. Bytecode installation

All functions are compiled lazily in Nashorn, meaning that they are not compiled until they are invoked. The initial compilation goes through all the compilation phases to compute symbols and other data needed in runtime. The only bytecode that is actually output is a class with a method that instantiate a runtime representation of the program function. When the program function is invoked initially, Nashorn notices that no compiled version of that function exists so first, it has to compile a bytecode version of the function. During that compilation all nested functions are skipped, all data needed to create runtime representations and to invoke them was already computed in the initial compilation. The nested functions will be compiled and executed in the same way as the program function. It might seem unnecessary to not compile the program function at the initial compilation since it is known that it will be invoked but doing so would mean special treatment and complicating the code without any significant performance gain.

2.2.1 Constant folding

The constant folding phase simplifies the AST by transforming constant expressions to equivalent shorter versions. One of the simplest transformations it does is to turn static expressions like `5 + 3` into `8`.

It also performs more complicated transformations such as removing dead code blocks from if statements where the condition is static. For example, it replaces the code in Listing 2.2 with `a = 5` since it is known that the `else`-block will never be executed (the `boolean` value of `5` is `true` in JavaScript)

```
1 if (5){
2   a = 5
3 } else {
4   a = 7;
5 }
```

Listing 2.2: If statement that can be folded

2.2.2 Control flow lowering

The control flow lowering phase finalizes the control flow of the JavaScript program. It performs actions such as copying finally-blocks of a try-catch to all places that terminates the control of the try-catch block and guaranteeing return statements to methods to ensure that the control flow of the program conforms to the ECMAScript specification.

It also replaces high-level AST nodes with lower level runtime nodes, for example it replaces the node representing builtin operators like `instanceof` and `typeof` with nodes that can be executed directly in runtime.

2.2.3 Program point calculation

Program point calculation is needed for the optimistic type system to work. It assigns program point numbers to all points in the program that could potentially fail due to optimistic type guessing. When that happens the program point number is used to determine where to resume the execution of the program after the function has been recompiled.

An example of such a place is a multiplication or addition of two variables that could overflow a numeric bytecode type such as `int` or `long` (JavaScript numbers do not overflow) or a property getter of an object where the property type is unknown.

2.2.4 Transform builtins

The transform builtins phase replaces calls to builtin JavaScript functions with other, more efficient function calls where possible. Currently this phase only replaces calls to `Function.apply` with `Function.call`. Those two functions are equivalent in the sense that they both invoke the function they are called on. The first argument to both of them is the object that should be bound to `this` inside of the invoked function. The remaining arguments are the arguments to the invoked function. How these arguments are passed to the function is the difference between the two functions, `apply` has only one parameter which is an array that contains the arguments to the invoked function while `call` takes the arguments as a regular argument list. Examples of them both are shown in Listing 2.3.

```
1 function f(a, b, c) {
2     // some JavaScript code
3 }
4 f.apply({}, [1, 2.0, {}]);
5 f.call({}, 1, 2.0, {});
```

Listing 2.3: If statement that can be folded

The problem with `apply` arises when the arguments are represented with different types. To be able to put them all in the same array all types need to be widened to `java.lang.Object` and the primitive types will have to be boxed. For `call` where each argument is passed separately, they can have different types and the primitive types will not have to be widened.

There are however limitations to when this transformation is possible to do. In Listing 2.3 it is possible, the array passed to the `apply` function can never change since it is not accessible outside the call. In Listing 2.4 a global array is passed as argument to the `apply` function and in this case it is not possible to replace `apply` with `call`. At compile time, it is not known what that array contains and it could even change from one time to the other.

```
1 f.apply({}, a); // 'a' is a global variable
```

Listing 2.4: If statement that can be folded

2.2.5 Function splitting

The JVM has a method length limit of 64KB. JavaScript functions have no such limit and can be of arbitrary length. Functions longer than 64KB cannot be directly mapped to a bytecode method since the JVM would throw an error when the class is loaded. 64KB is quite a big limit but since JavaScript is a common target language for compilers, e.g. the TypeScript compiler (see Section 1.1.2) and Mandreel[1], there's a lot of code around that is generated by computers that potentially contains longer functions.

To tackle this problem in Nashorn, the function splitting phase splits longer functions into several shorter functions. The splitting raises a few issues when it comes to variable scoping since one split part of a function might use variables declared in another one. That is solved by moving such variables to the lexical scope object of the function. That costs performance since local variables otherwise can be stored in local variable slots and accessed with simple memory reads and writes which is faster than the `invokedynamic`-instructions lexical scope accesses require, see Section 2.3.4.

No exact computation on how long the generated bytecode would be is performed. The functions are split heuristically when they are considered to be "too long". The reason for that is simply that it is not known exactly how long the bytecode representation of the function will be. To know that, the function splitter would have to operate on a lower level than the AST, preferably bytecode level.

2.2.6 Symbol assignment

This phase assigns symbols to each block in the AST. It has to keep track of which variables end up as local variables and which need to be kept as fields in the lexical scope objects to be accessible by nested functions.

2.2.7 Scope depth computation

The scope depth computation phase computes at which depth in the scope a variable is used. For example on line 4 in Listing 2.5 `a` is returned and the scope depth of `a` there is two since `a` was declared two scopes up. The scope depths are used in runtime to enable fast lookup of variables.

[1]Mandreel's official website: `http://www.mandreel.com/`

```
1 var a = 4711;
2 function b(){
3   return function c() {
4     return a;
5   }
6 }
```

Listing 2.5: Small example showing what scope depth means

2.2.8 Optimistic type assignment

The optimistic type assignment phase assigns initial types *optimistically* to all program points in the AST. The assigned types are used in the bytecode generation to decide what type of bytecode instructions to use. Optimistically means that the primitive types that the JVM can execute fast are chosen first, preferably `int`s since `int` instructions are the fastest [12]. More details on how optimistic types are handled in runtime will be presented in Section 2.3.4.

Nashorn can execute JavaScript code both with and without the optimistic type system enabled. If the optimistic type system is disabled all expressions that cannot be statically proved to be of a certain type are represented as `java.lang.Object` since all Java objects can be assigned to that, including boxed primitives such as `java.lang.Integer` and `java.lang.Double`. That reduces the performance a lot since fast bytecode instructions like `iadd` and `ladd` cannot be used to operate on the values directly and the JVM can not eliminate boxing internally.

2.2.9 Local variable type calculation

The local variable type calculation phase calculates types of expressions that can be proved statically. It only applies to variables that are local to a function, meaning variables that do not need to be kept in a lexical scope object. The reason for that is that scope variables can be accessed and changed elsewhere. For example a variable that is declared in one function and then changed by a nested function. The nested function can be passed around to other functions and then finally be invoked and might change type of the variable at a place that is not anywhere close to where the variable was declared.

If a local variable changes type inside the function it uses different local variable slots depending on the live range of the variable for each type.

Statically proved types are preferred over optimistic types since they are known to never overflow and therefore never cause recompilation. The type used is also as narrow as possible meaning that they will give at least the same performance benefits as optimistic types.

2.2.10 Bytecode generation

This phase simply generates the bytecode from the AST. Because functions are compiled lazily, each compilation will in most cases only emit one compiled bytecode method contained in a class. If the function is split, more methods will be emitted.

2.2.11 Bytecode installation

The bytecode installation phase loads the emitted classes to the JVM to prepare the code for execution.

2.3 Runtime

The JVM provides a competent runtime environment for Nashorn that already handles stuff like memory management and optimizing JIT compilation. Despite that, Nashorn still needs to handle some tasks at runtime. Nashorn relies heavily on `invokedynamic` and needs to manage all linking at runtime. The lazy compilation and the dynamic types of JavaScript means that the runtime has to be able to trigger recompilation of functions.

2.3.1 Runtime representation

At runtime all JavaScript objects and functions are represented by lower level runtime objects, namely instances of `jdk.nashorn.internal.runtime.ScriptObject` or subclasses to it. Functions are represented by `jdk.nashorn.internal.runtime.ScriptFunction` objects. There also are classes for representing built in functions and objects and the lexical scopes.

For objects that are constructed with object literals (`{...}`) Nashorn generates new classes as needed.

2.3.2 Dynamic linking

The dynamic linking in Nashorn is done by `invokedynamic` with the help of a library called *dynalink*. Dynalink is a helper library for linking dynamic call sites and handles most of the actual linking [22]. It was first implemented as a standalone library but is since Java 8 a part of OpenJDK. The library is initialized by setting up a `DynamicLinker` with a list of linkers. Each linker is asked, in priority order, to link the call site as shown in Figure 2.3. If the linker is able to link the call site it will be asked to do so and returns a `GuardedInvocation` object that contains the target `MethodHandle` and any guards that can invalidate the call site. Whether a linker can link a specific call site is determined by the type of the object (or primitive) that the method is invoked on.

Nashorn has several linkers, the main one is the `NashornLinker` which links all JavaScript objects and functions (instances of `ScriptObject` and subclasses). There are other linkers that link JavaScript primitives, access to the Java standard library, and JavaScript objects defined externally from Java code among others.

Dynalink also supports type conversions and that is also handled by the linkers, for example a conversion from a JavaScript object to a string is handled by `NashornLinker`.

All of this makes the bootstrap method for `invokedynamic` very simple to define, Listing 2.6 shows Nashorn's bootstrap method. It just propagates the control of the linkage to dynalink.

Dynalink has support for just a few basic operations that links a call site to methods performing a certain action. The operations are the following:

getProp Returns a property of an object

getElem Returns an element of an array

getMethod Returns a method of an object

setProp Sets a property on an object

setElem Sets an element on an array

call Invokes a method

new Invokes a method as a constructor

These operations are closely related to Nashorn's object model described in Section 2.4.

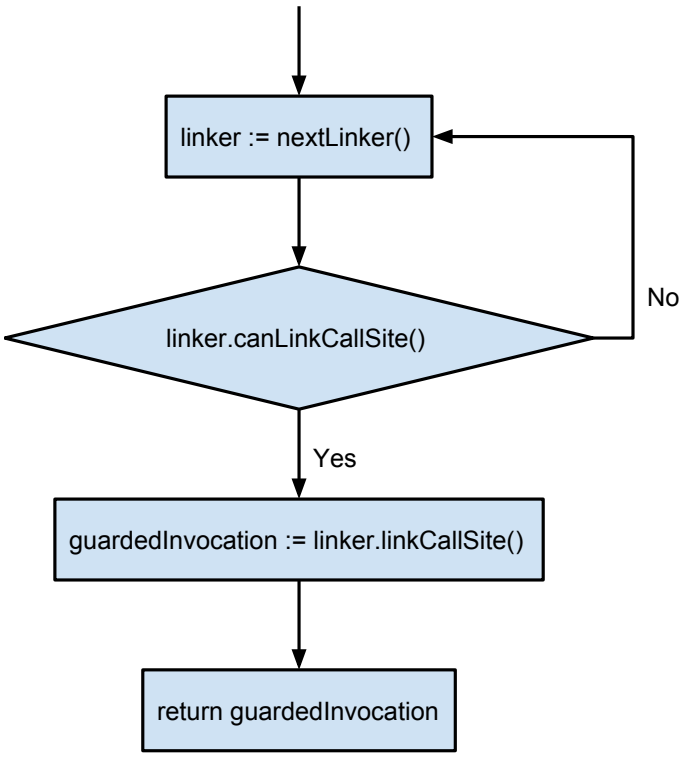


Figure 2.3: Simple flow graph of how dynalink links a call site

```
1 public static CallSite bootstrap(final Lookup lookup, final String opDesc,
      final MethodType type, final int flags) {
2   return dynamicLinker.link(LinkerCallSite.newLinkerCallSite(lookup, opDesc
        , type, flags));
3 }
```

Listing 2.6: Bootstrap method when using dynalink

A call site example

Listing 2.7 shows an example of a simple function call in JavaScript. A function named `a` is invoked with the number literal `10` as argument. When compiled to bytecode that function call looks like in Listing 2.8.

```
1 a(10);
```

Listing 2.7: A simple JavaScript call site

Line 1 in Listing 2.8 performs a dynamic invocation to fetch the `ScriptFunction` instance representing JavaScript function `a`. The call site descriptor on line 1 is somewhat interesting, it uses dynalink's operations in conjunction. What it basically means is "*Give me a method named* `a` *but if you can't find one, a property or element with the same name will do*". Then the bootstrap method returns a `CallSite` with such a `MethodHandle` which is then invoked to get the JavaScript function.

```
1 invokedynamic dyn:getMethod|getProp|getElem:a(Object;)Object;
2 getstatic ScriptRuntime.UNDEFINED : Undefined;
3 bipush 10
4 invokedynamic dyn:call(Object;Undefined;I)I;
```

Listing 2.8: Compiled version of the call site in Listing 2.7

The reason for using the operations in conjunction is that JavaScript has no clear separation between them. An example of that is shown in Listing 2.9. On line 4 the property `prop` is retrieved as an array access. Such code results in the usage of `getElem` but there is no element named `prop` on `obj`, only a property. That means that `getElem` will not be able to link the call site. When that fails, `getProp` is tried afterwards and that will find the property and link the call site properly.

```
1 var obj = {
2     prop: 5
3 }
4 var c = obj["prop"] // returns 5
```

Listing 2.9: A property of an object being retrieved as an array access.

The priority order is different depending on how the variable was requested, in Listing 2.8 `getMethod` is first operation since it was a function call, i.e., `a(10)`. Had it been a property access from the scope instead, i.e., `a`, `getProp` would have been prioritized.

The JavaScript function is invoked on line 4 in Listing 2.8 by using the dynalink operation `call`. As can be seen on the parameter list, the function expects three arguments. The function object itself is the first argument to the function and is mainly used inside the function to access the lexical scope object that belongs to the function.

The second parameter is the object that is bound to `this` in the function. Line 2 pushes the `this` object to the stack, In this case `this` is not defined so the JavaScript value `undefined` is loaded to the stack.

The first two arguments are always the function itself and the `this`-object. Those arguments are internal to Nashorn and has no equivalent in the JavaScript source code.

After the internal arguments, are the JavaScript source code arguments. Line 3 pushes the argument from the JavaScript source code to the stack. In this case the number literal `10` is the only argument. If there have been more, they would also have been pushed to the stack.

The return type of the call site is `int`, that is not necessarily the actual return type of the function but rather a guess made by the optimistic type system.

Type specializations and lazy compilation

On line 4 in Listing 2.8 the bootstrap method will return a `CallSite` with a `MethodHandle` that points to a method that takes the listed parameters and has the correct return type. Since all functions are compiled lazily, one such function might not yet exist in its compiled form, on the first call to the function, it will certainly not. What happens in that case is that a type specialization is compiled and a `CallSite` with a `MethodHandle` that points to that function is created and returned from the bootstrap method. The next time `a` is called at a different call site with the same type signature, the compiled version will be found so no additional type specializations will have to be compiled. But as soon as the function is called with a new type signature a new type specialization will be compiled, that could for example happen if a `double` value would be passed as parameter instead of an `int`.

Listing 2.10 shows an example of this. Function `f` on line 1 will not be compiled until it is called on line 4. Since the argument is an `int`, it will be compiled for an `int` argument specifically. On line 5 the function is called again but `2.1` can not be represented as an `int` so the previously compiled version can not be used. Therefore, a new compilation will be triggered specifically for when the parameter `b` has the type `double`. The invocations on line 4 and 5 invokes the same JavaScript function but they will in fact end up invoking different bytecode methods that have different parameter types.

```
1 function f(b) {
2     Some JavaScript code...
3 }
4 f(17);
5 f(2.1);
```

Listing 2.10: A JavaScript function that is being invoked with different argument types

2.3.3 Relinking

A linked call site will invoke the same method on every consecutive invocation. But what happens if the function has changed? In JavaScript it is possible to overwrite functions with new functions or even assign a completely different type to the variable, for example a `number` or an `object`.

Because of that, the call sites are created with a guard using `MethodHandles.guardWithTest()`. That method constructs a `MethodHandle` that executes a guard before each invocation. If the guard passes it invokes the linked method, if not it invokes a fall-back method. In the case of JavaScript functions in Nashorn, the guard checks that the function is still the expected one. If the guard fails, a method that triggers relinking of the call site will be invoked.

2.3.4 Typing

As mentioned earlier, types are assigned to program points in two different ways: optimistic type assignment and statically proved types. The types are assigned by the compiler but the runtime environment needs to handle the optimistically assigned types. This section will describe how that works.

Statically proved types

The statically proved types do not need any special runtime processing. They are stored in bytecode local variable slots and it is already known at compile time that they will never overflow or change to a different type.

Optimistic type guessing

Listing 2.11 shows a simple JavaScript function that is being invoked.

```
1 function a(b) {
2   b*b;
3 }
4 a((1 << 16) +  1);
```

Listing 2.11: Simple JavaScript function

The function call on line 4 gets linked to a method that looks like the one in Listing 2.12. It uses `int`-instructions throughout the function, since the function was called with an `int` argument. It loads the parameter to the stack twice, multiplies them and then returns the result. As long as the multiplication does not overflow the code will be executed and return an `int` as expected.

```
1 iload 1
2 iload 1
3 invokedynamic imul(II)I
4 ireturn
```

Listing 2.12: Compiled bytecode for function a in Listing 2.11

However, the multiplication can overflow and that needs to be handled since JavaScript numbers do not overflow. The potential overflow is the reason for not using a regular `imul` instruction on line 3 but instead a dynamic invocation of a method with the same name. The `imul` invocation is surrounded by a try-catch (not shown in Listing 2.12 for visualization reasons) and if the multiplication overflow a so called `UnwarrantedOptimismException` will be thrown by `imul`. That will in turn trigger recompilation by throwing a `RewriteException` that contains all information needed to resume execution at the same place as the error occurred. That information includes context such as variables on the stack and the program point number assigned by the compiler in the program point calculation phase.

Two new methods will be compiled when an optimistic assumption fails. First of all a version with the new types is compiled, in this case the function takes an `int` argument. Since the multiplication overflowed, the return type has to be `long`. That compiled version of the function looks like in Listing 2.13.

```
1 iload 1
2 i2l
3 iload 1
4 i2l
5 invokedynamic lmul(JJ)J
6 lreturn
```

Listing 2.13: Compiled bytecode for function a in Listing 2.11 with long return type

A special version of the function is compiled as well. The function is called a `rest-of`-method since it is used to execute the *rest of* the function from the point where the overflow occurred. The `RewriteException` that caused the recompilation is passed to the function as argument. It restores the stack and jumps to the point in the function that failed. In this case it would convert the two factors to `long`s, perform a long multiplication with `lmul` and then return a long.

2.4 Object model

The way JavaScript objects are modelled in Nashorn is actually quite simple. There are a few different kinds of objects in JavaScript that all might not be what one typically refers to as an object but they are all represented similarly and properties on the objects are accessed similarly. Some things mentioned in this section can be repetitive from previous sections but it is still worth emphasizing how objects are handled in Nashorn.

```
1 var a = {
2     b: function(){return 5;},
3     c: "Hello!",
4     d: 5,
5     e: false
6 };
```

Listing 2.14: Example of an object literal

First is the scope, each function has a scope object which contains all properties that were declared inside the function, not including nested functions' properties. From each function the parent scope is also accessible to be able access declarations from an outer function. The scope is represented as a regular Java object and is passed to the function when it is first instantiated.

Second, there are objects created with JavaScript object literals, Listing 2.14 shows an example of that. They are represented as a regular Java object, different classes are used depending on the number of properties and their types. The classes are generated by Nashorn as they are needed.

In JavaScript the new-operator can be used to invoke a function as a constructor. What happens when a function is invoked with the new-operator is that an object is

created and is bound as `this` inside the function. Then the function is executed as a normal function but the `this` object is returned from it. There is actually an exception to that, if the function's return value, in the JavaScript source code, is of an object type, that value is returned instead of `this`. That is according to the EcmaScript specification [9]. Listing 2.15 shows an example of how such an object can be instantiated in JavaScript. The constructor function's prototype is used as the methods and the class wide properties (static fields in Java) of the object.

Independent of which kind of object, the property getting mechanism described in Section 2.3.2 is used, namely an `invokedynamic` instruction with the call site descriptor being something like `getProp|getMethod|getElem`. To set properties or elements, `setProp|setElem` is used as call site descriptor for the dynamic call site.

```
1  function A (b) {
2      this.b = b;
3  }
4
5  A.prototype.c = function () {
6      return this.b;
7  };
8
9  A.prototype.CONSTANT = "a constant";
10
11 var a = new A("argument");
12 print(a.c())         // prints "argument"
13 print(a.CONSTANT)    // prints "a constant"
```

Listing 2.15: JavaScript class example

This object model differs from JVM bytecode's object model. Bytecode uses the same object model as Java does with builtin support for classes with methods and fields. In bytecode a class is the smallest possible execution unit, there are no global or standalone functions, while in Nashorn every function is a standalone function, either as a property on a scope or as a property on an object.

2.5 Warmup time

Time to warm up the code is always a concern when executing code on the JVM because of the JIT compilation. The code has to be executed a few times before it reaches a stable state where hot methods (methods frequently used) are compiled to native machine code for faster execution.

A couple of the features mentioned above are other sources for increased warm up time in Nashorn. First of all are the lazy compilations, before the first execution of the code there is not a single compiled bytecode version of it and that needs to be compiled. After the code has been compiled to bytecode, the JVM still has to compile the hot methods to native machine code before the code reaches a stable state.

The second source of increased warmup time is the optimistic type guessing. With optimistic type guessing it is not even known if the compiled bytecode can be used and in many cases it will need to be recompiled as explained in Section 2.3.4. The main benefit from this is that the stable state will consist of methods with primitive types making them efficient to execute. However, it will take longer time to get there since recompilation takes time and depending on the program it can happen very frequently before the code is warmed up.

The amount of warmup time caused by the optimistic type guessing depends on the number of program points where type assumptions are made since each assumption could cause deoptimizing recompilations. Each assumption can cause only a constant number of recompilations, one for each builtin bytecode type and `java.lang.Object`. This means that the maximum number of recompilations in a program grows linearly with the number

of assumptions. The number of assumptions is not necessarily directly mappable to the code size but a safe assumption would be that a program with a bigger code base has more assumptions and thereby has longer warmup caused by the optimistic type guessing.

There is however an important trick in Nashorn to reduce the warmup time. During recompilation, type information from runtime is used to prematurely deoptimize program points whose current runtime type is known to have been wider than the type that would have been used if no other type information was available.

Chapter 3

TypeScript implementation

This chapter is about the TypeScript implementation. First it is presented what changes were made to implement the TypeScript front-end and what limitations the implementation has. The results presented in the results section are performance measurements that show how Nashorn performs with TypeScript compared to JavaScript and observations on how well suited Nashorn is for implementing other dynamic languages on top of it.

3.1 Method

The focus of the TypeScript implementation has not been to fully support TypeScript according to the specification. Instead, the focus was to be able to run TypeScript programs with type annotation and use them during the bytecode generation phase to generate more efficient bytecode with narrower types. All differences to TypeScript's specification are listed in Section 3.1.3.

3.1.1 Parser

The TypeScript parser has a lot of common functionality with the already existing JavaScript parser, since TypeScript's syntax is just an extension of JavaScript's. Therefore, the parser was implemented by extending the JavaScript parser and adding additional functionality to parse the additional language constructs from TypeScript. The focus was to parse the constructs that are related to types, like the types themselves, interfaces, classes and typed expressions and statements.

3.1.2 Compiler

The purpose of the changes that were made to the compiler is to make use of the TypeScript types to generate bytecode with more accurate types. Two compilation phases, *Type resolution* and *Type association*, has been added and the symbol assignment phase has been extended to handle symbols for type references (type annotations referring named types).

Symbol assignment

The named types in TypeScript live in a separate declaration space from variables and functions [16] and therefore need to have separate symbols. The symbol assignment phase assigns symbols to the named types and connects them to all references to the named type.

Apart from handling symbols for named types this phase also assigns TypeScript types to all symbols where they are present in the source code.

Type resolution

There are two different ways to reference a type in TypeScript, type references and type queries [16]. On line 4 in Listing 3.1 is an example of a type reference and on line 5 an example of a type query. `c`'s type is resolved to `number` since that is the type of property `b` in interface `A`.

```
1 interface A {
2     b: number;
3 }
4 var a: A = {b: 4711};
5 var c: typeof a.b = 17; // c's type is number
```

Listing 3.1: Example of type references and type queries

Type references are resolved by setting a reference to the named type in the type reference node. There are two difficulties with named type, namely that they can have generic type parameters and they may contain circular references.

Type queries are resolved by simply replacing the type query with the type that the referred variable has.

In Listing 3.2 is an example of a generic interface. All generic types are resolved by copying the interface and replacing all occurrences of the type parameter with the type argument in the type reference. In this case line 4 would cause one copy of interface `A` to be generated with all `T`s replaced with `string` and line 5 would cause another version to be generated where all `T`s are replaced with `number`. All resolved named types are stored, and on consecutive references they will not be resolved again but use the result from previous references. The references to `A` on line 4 and line 5 are considered different references since the resulting type is different.

```
1 interface A<T> {
2     b: T;
3 }
4 var a: A<string> = {b: "Hello world!"};
5 var b: A<number> = {b: 4711};
```

Listing 3.2: A generic interface

Circular references like the ones in Listing 3.3 would, if not handled specially, cause infinite recursion when resolving the type references. Circular references are detected in a depth-first-search manner. All references that are currently being resolved are pushed to a stack and whenever a reference that is already on the stack is encountered, a circular reference is detected. When a cycle is detected, it is resolved by simply referencing that type reference back to the type already on the stack.

```
1  interface A {
2      b:A; // self reference
3  }
4
5  var a:A;
6
7  interface C {
8      b:D; // A property with type of a subtype.
9  }
10
11 interface D extends C{
12     a:string;
13 }
14
15 var d:D;
```

Listing 3.3: Circular references

At the end of the type resolution phase, all types are resolved and they themselves are aware of what properties and methods they have and the type of them, so after this phase they will not have to be edited further.

Type association

The type association phase has two purposes, first it infers the type of expressions and second it sets boundaries on which internal bytecode types can be used to represent the expressions.

The two type boundaries are the *optimistic boundary* and the *pessimistic boundary*. The optimistic boundary is the narrowest bytecode type that could be used to represent the type. The pessimistic boundary is a bytecode type that can fit all possible values that expression could have. Setting boundary internal types is what will cause the bytecode generation phase to generate bytecode with more accurate types.

The JavaScript implementation always uses `int` as the initial optimistic type for unknown types but the TypeScript implementation can use different types depending on the TypeScript type of the expression. What the optimistic boundary is for each TypeScript type follows naturally, for example the optimistic type boundary for `number` is `int` and for an object type such as an interface or class reference it is `java.lang.Object`.

Unfortunately the pessimistic boundary needs to be `java.lang.Object` for all TypeScript types. The reason for that is that values of type `undefined`, `null` and `any` can be assigned to all types. The first two are an issue because there is no way to represent `null` or `undefined` as any of the primitive bytecode types and they must therefore be represented as `java.lang.Object`.

On line 2 in Listing 3.4 the issue with the `any` type is shown. A value of type `any` is assigned to a variable of type `number`. That is permitted in TypeScript and means that `b` could actually contain any value and is by no means restricted to numbers only at runtime.

```
1 var a:any = {};
2 var b:number = a;
3 b = null;
4 b = undefined;
```

Listing 3.4: Cases that prevent narrow pessimistic boundary

This is not that big of a problem when optimistic types are enabled since the optimistic type boundary is used at the first compilation and the code will be deoptimized only when one of the three cases above are encountered. With optimistic types disabled, however it means that any expression that cannot be statically proved to be of a certain type will be represented as `java.lang.Object` since the type used will have to be able to fit all possible values. That is unfortunate since that means there can not possibly be any performance gain from TypeScript compared to JavaScript with optimistic types disabled.

The type inference is done according to the TypeScript specification [16] although it is currently somewhat limited and does not fulfill all requirements from the specification, more on that in the following section.

3.1.3 Limitations

As stated in Section 3.1, the goal with the TypeScript implementation was to make use of the type annotations in TypeScript and not implement the language fully according to the specification. This section will go through what was not implemented and why it was left out.

Modules

A TypeScript program typically consists of several source files that each corresponds to a module. All module files need to be processed together at compile time and type informa-

tion needs to be shared between them. That is quite a big change compared to JavaScript where each file is processed individually at compile time and does not interact with each other until they are linked in runtime. Implementing modules is doable but would require more work and time. Also, it would not help to achieve the primary goal of the TypeScript implementation and were therefore left for further development.

Enums

Enums is one of the three kinds of named types supported by TypeScript and is the only one of them that has not been implemented. The reason for leaving enums out is the same as for modules, they would not add enough value to this thesis to be worth the implementation time.

Type inference

Type inference is a key feature in TypeScript that makes it possible for developers to omit type annotations and still have static type checking. Although type inference is partially implemented, some parts of it were not.

There is a concept called contextual typing in TypeScript which was not implemented, Listing 3.5 shows an example of how contextual typing works. The variable `a` on line 1 is declared to be a function that returns a value of the `number` type. Because of that, the function that `a` is initialized with is said to be *contextually typed* to return a `number`. It has the consequence that the return value `b` must be assignable to `number`.

```
1 var a:() => number = function (){
2     var b = 5;
3     return b; // b must be assignable to number
4 }
```

Listing 3.5: Example of contextual typing

Another concept related to type inference is *the best common type*. The best common type is calculated for example when inferring return type of a function with multiple return statements or the element type of an array literal. Calculation of the best common type is not implemented and because of that type inferring is not working in those cases. The reason for not implementing it is the same as for contextual typing, it would not take this thesis closer to its goal and the same results, performance wise, can be achieved without them by just making sure to include type annotations in all such cases.

Type checking

Type checking is only partially implemented, to have complete type checking it would be necessary to fully establish relationships between types. Types can be related in three different ways in TypeScript, they can have subtype/supertype, assignability and identity relationship. Type relationships are only implemented for the predefined types so for those type checking works but not for any other types. The reason again is that it would be time consuming to implement and it would not take this thesis any closer to its goal.

3.1.4 Performance analysis

In this section it is described how the performance analysis of the TypeScript implementation was done, which benchmarks were used and how they were used to measure performance.

The performance was analysed both with and without optimistic type guessing and the method will differ between them. Why that is and how the measurements were done is described later in this section.

Benchmarks

The benchmarks used are from the octane JavaScript benchmark suite [6] and were ported to TypeScript. The octane suite consists of several benchmarks of different sizes. Four of the smaller benchmarks have been chosen for the performance analysis. The reason they are used is that they measure slightly different characteristics of performance and that they are fairly small (< 2000 lines of code each). It would be interesting to measure warm up time on some bigger benchmarks as well (the biggest being Mandreel which is 277 377 lines of code) but given the time limitations it was not considered feasible to port them to TypeScript for this thesis.

Here follows the four different benchmarks and what their main focuses are according to the authors [6]:

Crypto Bitwise operations

NavierStokes Reading and writing numeric arrays

Richards Property load/store and function/method calls

Deltablue Polymorphism

Without optimistic type guessing

In Section 3.1.2 it is explained why the pessimistic type boundary of expressions in TypeScript cannot be narrower than they are in JavaScript. However, there are only a few special cases that cause that unfortunate fact and the benchmarks that were used in this thesis actually can be executed with narrower types.

Why is that relevant when narrower types cannot be used in the general case? For TypeScript that is true but this thesis is not only about TypeScript. There are other dynamic languages where narrower types potentially can be used for certain types and if that would give a large performance gain it could be something to consider for the new architecture for Nashorn.

The focus of this measurement will be performance in the stable state since that is what is expected to be affected by TypeScript when not using optimistic types.

This measurement will be done with the help of JMH [19] which is a tool that is a part of OpenJDK used for running benchmarks on the JVM. JMH supports the following features that will be used in this measurement:

- Warmup iterations that are not included in the measurement to be able to measure performance of the stable state
- Executing the benchmark in fixed time iterations measuring the average execution time of the benchmark
- Calculate a 99,9% confidence interval of the execution time for a benchmark

With optimistic type guessing

With optimistic type guessing the performance of the stable state is not expected to improve because of the type information in TypeScript, both JavaScript and TypeScript are expected to reach the same stable state. To verify that the same performance measurements as explained in the last section was done with optimistic type guessing enabled as well.

TypeScript is however expected to start out closer to the stable state because of the more accurate type guesses and therefore warm up time is expected to be shorter than for JavaScript.

The warm up time is presented using warmup slopes, meaning plotting the execution time of each iteration until the code reaches a stable state. JMH [19] was used for this

measurement as well but instead of calculating average execution time the execution time of each iterations was measured individually.

All benchmarks were executed in 60 iterations on the same JVM (one iteration means executing the benchmark once), measuring the time it takes to execute each iterations. This operation will in turn be repeated 15 times with a fresh JVM instance.

System used for benchmarking

Cpu 4 Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz with 8 cores each, a total of 32 cores

RAM 16 DIMM DDR3 Synchronous 1600 MHz (0.6 ns) of 16 GB each, a total of 256 GB RAM

Operating system Ubuntu precise (12.04.5 LTS)

Java version Java HotSpot(TM) 64-Bit Server VM (build 1.9.0-ea-b37, mixed mode)

3.2 Results

In this section, the results from the performance measurements of the TypeScript implementation is presented and discussed. In the end of the section, some observations that were made during the TypeScript implementation are presented as well.

3.2.1 Performance without optimistic type guessing

Figure 3.1 shows a bar chart with the results of all the four benchmarks when executed with optimistic type guessing disabled. All results are normalized on JavaScript execution time for better visualization. Since the bar chart shows relative execution time, lower is better.

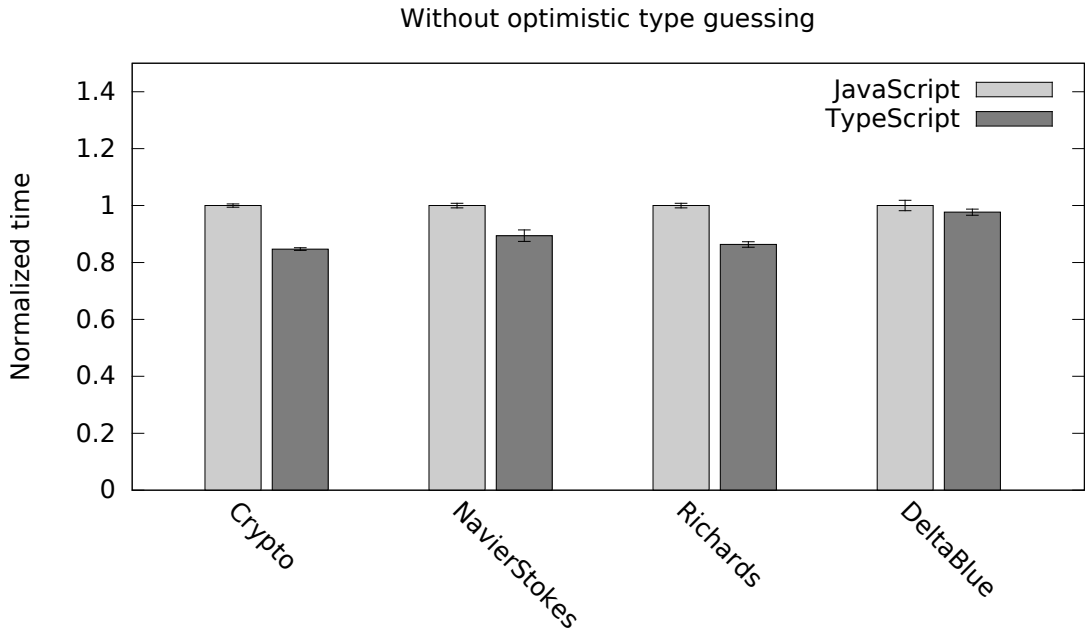


Figure 3.1: Results from benchmarks without optimistic type guessing, execution time is normalized on JavaScript performance, the error bars show the 99.9% confidence interval.

It is clear that TypeScript indeed performs better than JavaScript, how big the difference is differs from benchmark to benchmark. The biggest performance increase can be seen on the Crypto benchmark. That makes sense, since the main focus of that benchmark is bitwise operations of numbers. The biggest difference between TypeScript and

JavaScript is that JavaScript uses boxed types to represent numbers and TypeScript uses the pessimistic boundary type of the `number` type which is the bytecode primitive type `double`. Hence, it performs better on numbers and the overall performance is increased.

DeltaBlue however does not show as big difference as the others. This can be explained by the fact that benchmark is mainly about object polymorphism. For object types the pessimistic boundary in TypeScript is `java.lang.Object` which is the same type that JavaScript uses and hence the performance gain is not as big as with number intensive benchmarks such as Crypto. Richards and NavierStokes are quite number intensive as well and that could explain why the performance gain is almost as big as for Crypto.

3.2.2 Performance with optimistic type guessing

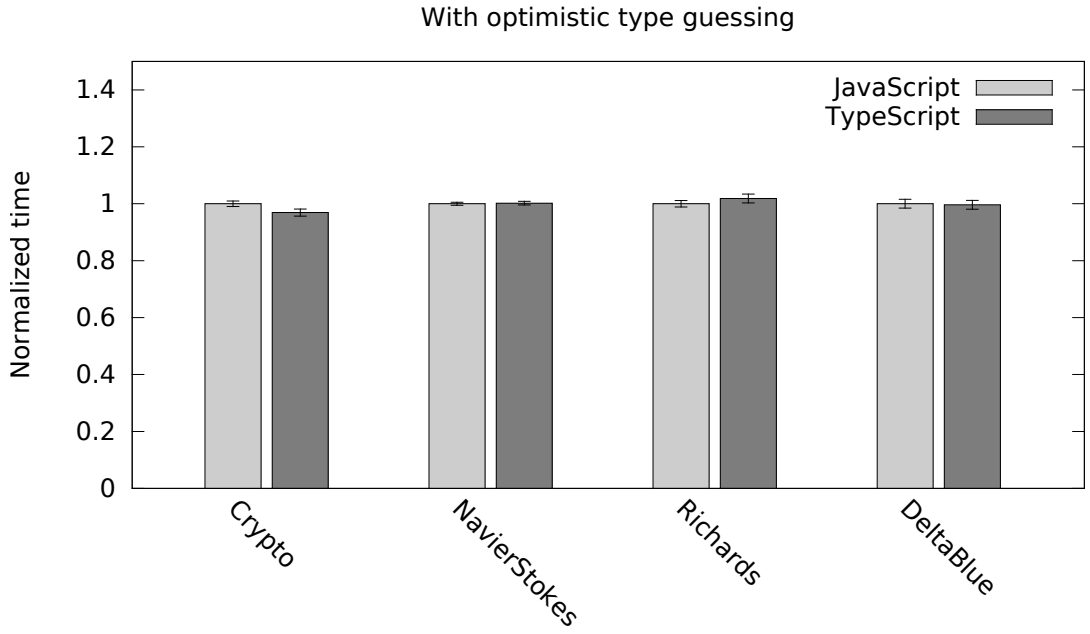


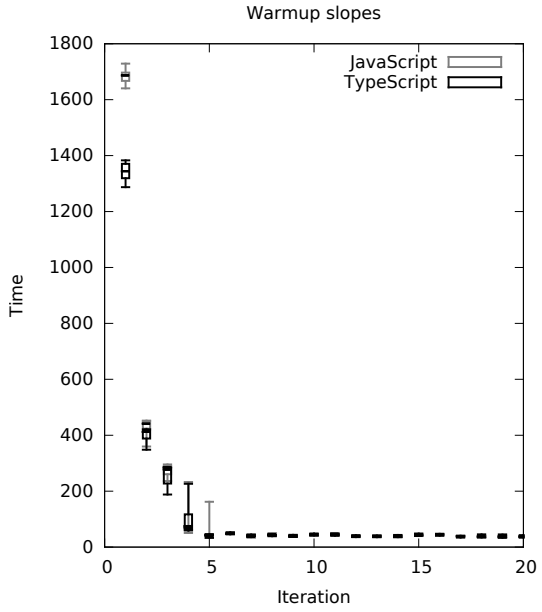
Figure 3.2: Results from benchmarks with optimistic type guessing, execution time is normalized on JavaScript performance, the error bars shows the 99.9% confidence interval.

Figure 3.2 shows the same performance measurements as in Figure 3.1 but with optimistic type guessing enabled. The performance difference for NavierStokes, Richards and DeltaBlue is not statistically significant, their confidence intervals are overlapped. Crypto however shows a significant difference to TypeScript's advantage. Even so the difference is small, under 4%, and given that the other benchmarks did not show any significant difference the results are as expected, namely no (or only a small) performance difference in the stable state.

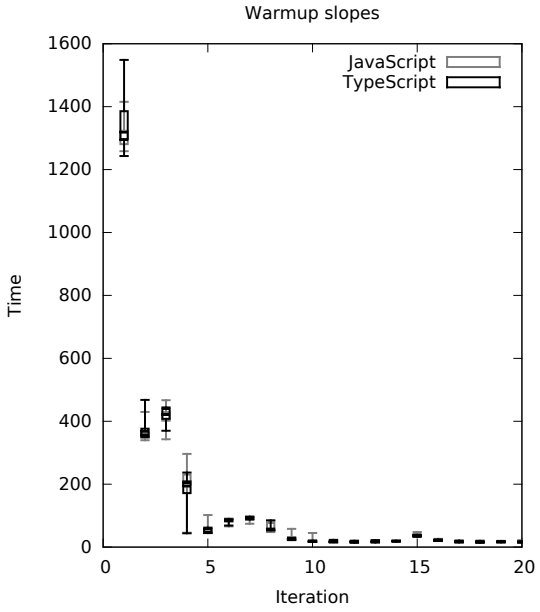
3.2.3 Warmup times with optimistic type guessing

In this section warmup slopes for JavaScript and TypeScript for each benchmark are presented, they can all be seen in Figures 3.3.

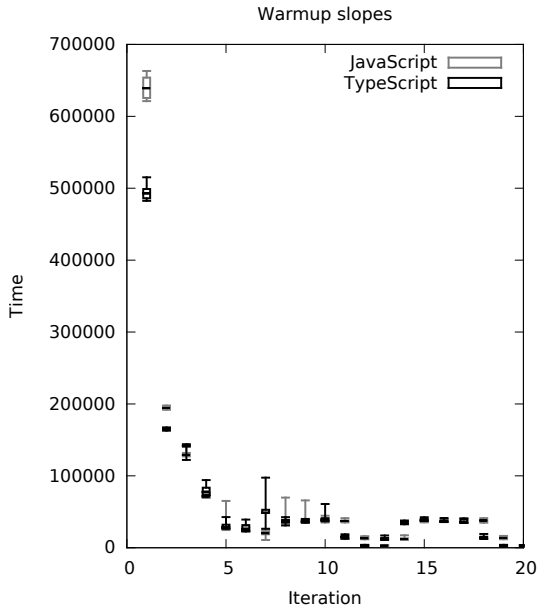
The warmup slopes for Crypto, DeltaBlue and Richards show similar patterns. The first iteration shows a big improvement in execution time for TypeScript compared to JavaScript. After that, JavaScript catches up and the difference disappears quickly. Some benchmarks show unstructured differences in later iterations. That could be caused by the garbage collector is slowing down the execution on different iterations for the two implementations.



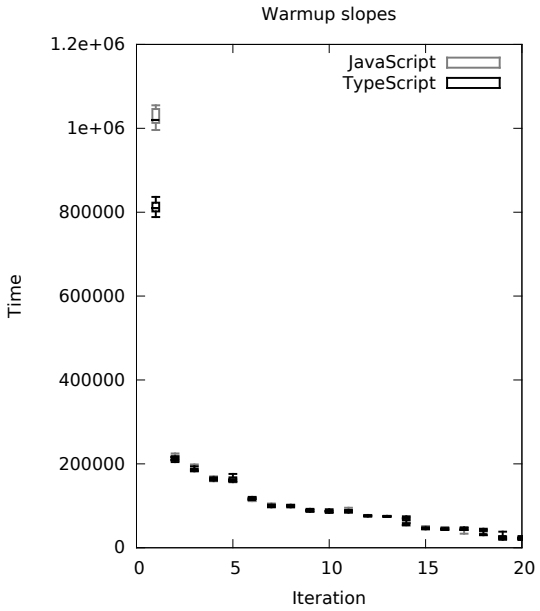
(a) Crypto



(b) NavierStokes



(c) Richards



(d) DeltaBlue

Figure 3.3: Warmup slopes for all benchmarks, the candlesticks shows the minimum, 1st quartile, median, 3rd quartile and maximum values for each iteration. One iteration is one execution of the whole benchmark. Iteration n is the nth time the benchmarks was executed on the same JVM. All benchmarks were executed for 60 iterations but only 20 are visualized in the figures because no interesting results can be seen after that.

Without optimistic type guessing the number intensive benchmarks showed a bigger performance gain than DeltaBlue which are not as number intensive. That is due to the fact that they can use narrower types. With optimistic type guessing the opposite is expected for warmup time. The warmup is caused by the fact that the optimistic type system initially assumes almost every expression to be an `int` until proven wrong. For every case where that assumption turns out to be false, warmup time increases. For number intensive, or actually `int` intensive, benchmarks the initial assumptions will not be false as often as for less number intensive benchmarks. Meaning that the JavaScript implementation will not cause as long warmup and making the difference between TypeScript and JavaScript smaller.

That could explain why NavierStokes shows practically no difference at all, not even in the first iteration. Both Crypto and Richards are also number intensive but they use JavaScript classes to hold the number which causes deoptimizing recompilations. NavierStokes, however, only constructs one object in the setup of the benchmark and that is not included in the measurement.

A general problem with all the four benchmarks is that their code bases are small, less than 2000 lines of code each. Since the effect optimistic type guessing has on warmup time grows linearly with the size of the code base (or actually the number of type assumptions, see Section 2.5) the warmup time reduction from using TypeScript would probably be bigger and the results more clear for bigger benchmarks. NavierStokes is the smallest of the four benchmarks which could further reduce the difference in warmup time between JavaScript and TypeScript.

3.2.4 Lessons learned from implementation

This thesis is not only about TypeScript performance but also about improving the architecture of Nashorn. When it comes to implementing TypeScript on top of Nashorn, it was a fairly easy job to do.

The runtime was easily adapted to be able to set boundary types to be used by Nashorn's type system. Actually not even a single line of code in either the optimistic type assignment phase or the local variable type calculation phase was changed, added or removed, all changes were made on the effected AST nodes. Only two methods needed to be changed in a few effected AST nodes (nodes such as property accesses, element accesses and function calls). The two methods can be seen in Listing 3.6, they return the optimistic and pessimistic boundary of the node. Previously the return values were hard coded in each node and the change that was made was to make it possible to override the hard coded type boundaries.

```
1  @Override
2  public Type getMostOptimisticType() {
3      if (optimismBoundary != null) {
4          return optimismBoundary;
5      } else {
6          return Type.INT;
7      }
8  }
9
10 @Override
11 public Type getMostPessimisticType() {
12     if (pessimismBoundary != null) {
13         return pessimismBoundary;
14     } else {
15         return Type.OBJECT;
16     }
17 }
```

Listing 3.6: Methods that were overridden

Of course there has been work put in to implement TypeScript than this, with the parser and all other compiler logic to handle all the new language constructs. But this is the only place where that hooks in to Nashorn's runtime environment. All other changes were quite time consuming but there were no major design flaws in Nashorn that caused any big problems. The fact that it was so easy to take advantage of the TypeScript types shows that the type system in Nashorn is well designed and that it is easy to reuse and even extend.

So TypeScript was easy to implement on top of Nashorn but TypeScript is in many ways the perfect fit since no runtime behaviour differs from JavaScript and most compile time processing is the same as well, the main difference is the type annotations. What about other languages? There are issues in Nashorn to overcome before it is suitable for implementation of other languages. More details on this is presented in Section 4.1.1

Chapter 4

Architecture

Oracle's intention is, as stated earlier, to turn Nashorn into a generic runtime library for dynamic languages. The main benefit from this is that Nashorn's optimistic type system and dynamic linking would be available for reuse by other dynamic language implementors. That would make it easier to implement dynamic languages with good performance on the JVM.

This chapter is about Nashorn's architecture and what changes that could be made to simplify reusing Nashorn's core concepts. The new architecture is partly based on the observations from the TypeScript implementation presented in Chapter 3.

4.1 Designing a new architecture

4.1.1 Current issues

In many ways Nashorn is well designed, an example of that was discussed in Section 3.2.4. Namely, that Nashorn's type system is fairly language agnostic and easily extended. The biggest problem with it is that the type assignment phases operate on a JavaScript AST, actually all compilation phases in Nashorn operate on a JavaScript AST. It is not that surprising since Nashorn was initially written as a runtime for JavaScript only. That is the underlying cause of most of the design issues mentioned in this section.

Some optimizations, like liveness analysis and control flow analysis, are not trivial to perform on an AST [1]. Söderberg et al. shows that it can be done in [21] but it is more complex to do. The AST representation is also structurally very different from bytecode, which is the target language. This results in a complicated bytecode generation phase in Nashorn. It consists of around 10 000 lines of code and is by far the most time consuming phase of them all. If Nashorn had a lower level representation in between the AST and bytecode representations it would open up the possibility to do more advanced optimizations and reduce the code size and the execution time of the bytecode generator.

Another problem with the AST is that it represents the *abstract syntax* of a programming language. An abstract syntax is derived from the *concrete syntax* which is the grammar of the programming language, hence the abstract syntax and the AST is language specific by definition [1]. Although it might be possible to create an AST that could be used to represent multiple languages one would probably run into problems at some point and it could be difficult to model some languages with a JavaScript AST. It would also not be trivial to do language specific optimizations without causing problems, since those optimizations would affect all languages that are implemented on top of Nashorn. For example, in Nashorn there are semantics built into the AST nodes. In the node representing binary operations, such as addition, the type of the result is inferred inside the AST node. To be able to reuse the AST for other languages all such usages of the AST would have to be moved elsewhere.

So if the AST cannot be reused easily, how to reuse the compilation phases that operate

on the AST? It's not possible and it would mean that each language implemented on top of Nashorn would need to implement its own version of for instance the type assignment phases and the splitting phase. To be able to fully reuse those phases they need to operate on a representation that is more language independent than an AST.

The AST issue is mostly a compile-time problem, what about Nashorn's runtime environment? It contains some JavaScript specific code. The linker links call sites according to the JavaScript specification. Even though linking is similar between languages in many cases, there are differences. Also, all builtin JavaScript variables and objects are part of the runtime.

4.1.2 A design suggestion

Based on the issues with the current design, the solution discussed in this thesis is an intermediate representation (IR) [1] for Nashorn. All language specific operations could then be performed on the AST and then it would be transformed into the intermediate representation for further optimization and type assignment. Apart from enabling reuse of Nashorn's solution it would also make the distinction between semantic analysis and other optimizations clearer. Actually the only operations that would be performed on AST level is the semantic analysis, quite similar to the compiler design example in [1]. Figure 4.1 shows how Nashorn's architecture could look like with an IR. One of the main benefits would be that recompilation could be handled inside Nashorn and would not need to concern the language implementors.

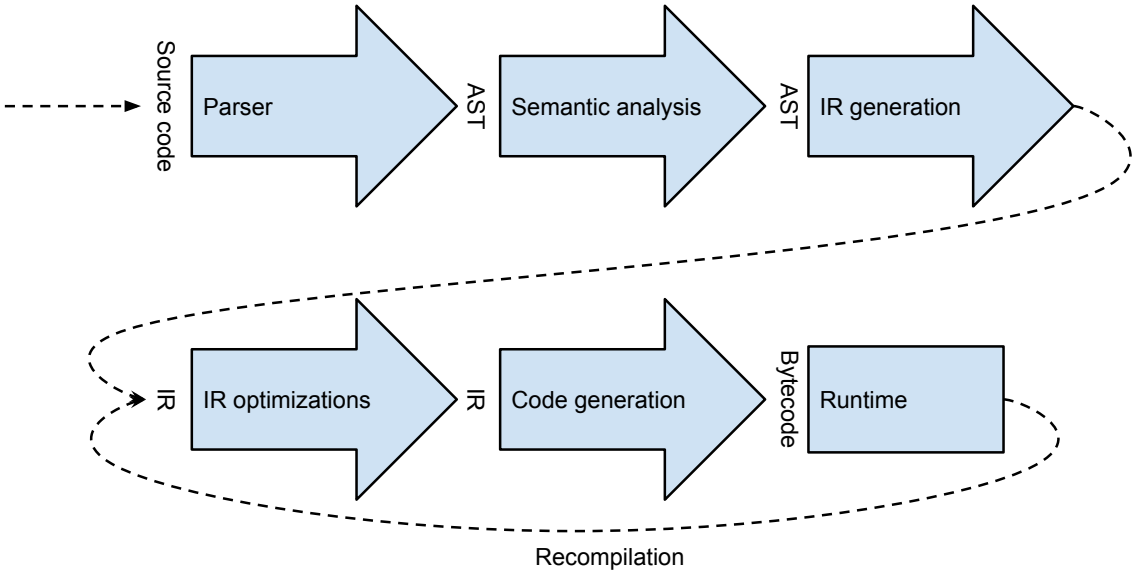


Figure 4.1: Overview of architecture with intermediate representation

What should be handled by Nashorn behind the intermediate representation and what should be left for the language implementor to implement in the compiler front-end? First of all, in terms of compilation phases that Nashorn has today, the type assignment phases, the function splitter and bytecode generation and installation phases are definitely bytecode specific and could perform on the IR rather than an AST.

In Section 1.1.2 a few concepts that are common to most dynamic languages are discussed. Those are dynamic typing, dynamic linking and closures. Dynamic typing will be handled in a similar way that Nashorn does today with the two type assignment phases except that they operate on the IR instead of the AST.

Support for closures in the IR would simplify implementing a dynamic language on Nashorn. That would require the IR to have support for lexical scopes in a way that the language implementor has full control of to be able to implement the scope rules of

the specific language. Although this has been considered, it is not a part of the design presented in this thesis, the reason for that decision is discussed in Section 4.1.6.

The dynamic linking will also need to be customizable by the language implementor. Each language has its own semantics and could require linking to be performed differently.

In the design suggestion presented in this thesis the IR is as similar to bytecode as possible but has additional untyped instructions. For every typed bytecode instruction, the IR has an equivalent but untyped operation which Nashorn assigns types to before emitting Java bytecode with correct types. For example an additional untyped addition operation is included and complements the already existing `iadd`, `ladd`, `dadd` etc. Because the similarities to Java bytecode, the IR is referred to as *Nashorn bytecode* in the rest of this thesis.

4.1.3 Operators

There is an interesting issue with the basic operations such as addition, multiplication and division etc. in Nashorn bytecode. The `+`-operator in JavaScript for instance can be used for multiple purposes. The most intuitive usage is to add two numbers but there are really no limits to which types the `+` operates on. For example `"2" + "2"` evaluates to `"22"`, `[] + []` to the empty string and `{} + {}` to `NaN`. This is specific to JavaScript and is different in other languages, for example in Ruby `[] + []` evaluates to the empty array.

```
1 function a(a, b) {
2     return a + b;
3 }
```

Listing 4.1: Add-operator example

It is common that operators behaves differently depending of the types of the operands, a compiler for a static language would solve this by emitting different code depending on the operand types. The problem for dynamic languages, however, is that the compiler front-end cannot know the types of the operands and can therefore not emit code based on the types.

Take the JavaScript code snippet in Listing 4.1 as an example. Function `a` on line 1 just returns the sum of the two variables, how would a Nashorn bytecode representation of that function look like? Without knowledge of the operand types there are only two options. The first one is to emit a conditional chain that performs different operations depending on the runtime type, Listing 4.2 shows parts of a pseudo Nashorn bytecode representation of that solution.

The code is not really valid Nashorn bytecode and most of the addition is left out since it would be too long, the code however shows the general problem with this approach. That amount of code would have to be generated for every add-operation in the program. It would kill the performance and totally defeat the purpose of both the optimistic type system and Nashorn bytecode. After all, the purpose of the optimistic type system is to be able to emit efficient bytecode with as narrow types as possible.

The other approach is to just use the Nashorn bytecode's `add` operation, like in Listing 4.3 and hope that it works. That seems like a naive approach, how would Nashorn know what bytecode to emit for the `add` operation if it ends up with an array and a string? Should it throw an error? In that case which error? Or should it do something else? If the Nashorn bytecode was intended for one language only, such as JavaScript, it could just do what the specification says but now there could be several semantic meanings of the same operation depending on the language. In some languages, like Ruby, it is even possible to overload operators for classes. That complicate the `add` operation in Nashorn bytecode even more.

```
    load 1              // parameter a
    instanceof Object
    ifgt OBJ
    instanceof int      // This is not valid bytecode since int is not a
        Class but used here for simplicity
    ifgt INT
    ...
OBJ:
    load 2              // parameter b
    instanceof Object
    ifgt OBJADD
    instanceof int
    ifgt MIXADD
INT:
    ...
OBJADD:
    load 1
    load 2
    invokestatic ScriptRuntime.add(Object,Object)Object
    areturn
MIXADD:
    load 1
    load 2
    invokestatic ScriptRuntime.add(Object,I)Object
    areturn
...
```

Listing 4.2: Pseudo Nashorn bytecode representation of the JavaScript function in Listing 4.1.

The solution is that while Nashorn bytecode provides the basic operations, the language implementor will have to tell Nashorn what to do with them depending on the type of the operands. Nashorn knows the types and the language implementor knows the semantics of the operator for the specific language depending on the types of the operands and provides that knowledge to Nashorn. If both arguments are `int`s Listing 4.3 could be compiled to the Java bytecode in Listing 4.4 and the addition would be a regular `int`-addition. If both arguments are Object types, code that handles addition of those types can be emitted. JRuby could for example emit a dynamic call site that links to the overloaded add-operator as in Listing 4.5.

```
load 1
load 2
add
return
```

Listing 4.3: Proper Nashorn bytecode representation of the JavaScript function in Listing 4.1.

To be able to utilize the type system fully, Nashorn has to know two things. First, to be able to do static type analysis of local variables, it has to know, depending on the operand types what the return type of the operator is. It could be an unknown type and that is then handled by the optimistic type system. Second, to be able to emit bytecode, it has to know how the operation should be executed.

That is the same information as the type system uses in Nashorn currently but it is slightly hard coded for JavaScript semantics.

```
1 iload 1
2 iload 2
3 iadd
4 ireturn
```

Listing 4.4: Possible Java bytecode representation of the Nashorn bytecode in Listing 4.3 when both arguments are ints.

```
1 aload 1
2 invokedynamic dyn:getMethod:add()Object
3 aload 1
4 aload 2
5 invokedynamic dyn:call(Object,Object)Object
6 areturn
```

Listing 4.5: Possible Java bytecode representation of the Nashorn bytecode in Listing 4.3 when both arguments are Objects. The addition is performed with a dynamic invokation to a method called add on the first argument.

4.1.4 Semantic analysis

The problem with operators described in the last section can actually be generalized. In dynamic languages some operations are performed in runtime that would in other, more static, languages be performed at compile time. One such operation is the semantic analysis which determines the meaning of each language construct.

For a dynamic language it is just not possible to perform a full semantic analysis at compile time because the information needed is not available until runtime. In fact, parts of the semantic analysis needs to be built into the runtime environment instead. The operator problem is just an instance of that general problem.

What Nashorn needs to achieve with its IR is to return full control of the semantics to the language implementors. This can not be achieved with a traditional semantic analysis in the front-end. Instead, the language implementor will have to plug in behavior to Nashorn's runtime environment to control the semantics of the language at all applicable places.

This does not mean that no traditional semantic analysis can be performed in the front-end. What can be done differs from language to language. For example in TypeScript the type checking could definitely be implemented as a traditional semantic analysis phase. But even for TypeScript parts the semantic analysis needs to be built in to the runtime since type safety is not guaranteed (a value of type `any` can be assigned to a variable of all types) and linking has to be done at runtime.

4.1.5 Bytecode analogy

In Section 4.1.2 it was stated that Nashorn bytecode should be *as similar to bytecode as possible but with untyped instructions*. Why is that and what does this really mean?

That question relates to what parts of Nashorn that should be made reusable. Nashorn's big advantage over other dynamic language implementation on the JVM is the type system which improves performance by using narrow types where possible. The reason to make the type system reusable is partly to simplify implementing new languages on the JVM with good performance but also to help existing implementations to increase their performance by making it easy for them to reuse Nashorn's type system.

JRuby for example already has a language specific intermediate representation designed for their uses which they perform several optimizations on. They would probably want to continue using that and do as few changes as possible to adapt to Nashorn bytecode. If Nashorn provided a library to build a bytecode-like untyped IR and a library to generate

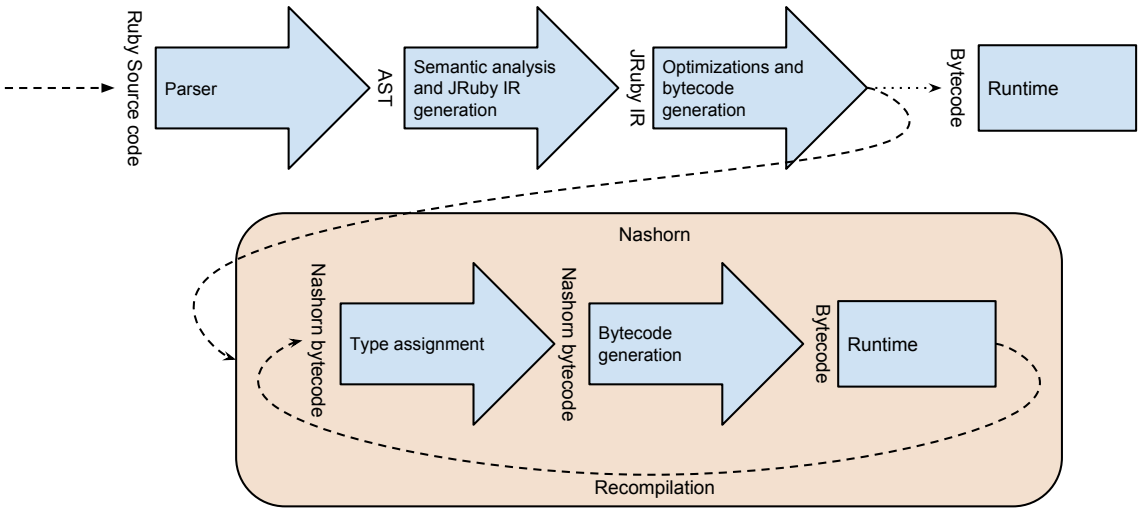


Figure 4.2: Possible design of JRuby when utilizing Nashorn's type system. They would get an alternative to generate Nashorn bytecode instead of Java bytecode.

it they would in many ways already know how to utilize it since the difference to what they are currently doing would not be that big. They can generate classes as they like, use their own object model, handle lexical scopes in a way that suits Ruby etc. The only difference would be that some instructions are replaced with untyped ones and that Nashorn does some intermediate processing before the actual Java bytecode is generated. Of course they would also have to plug in some language specific semantics for Nashorn to be able to process the Nashorn bytecode.

JRuby's design on top of Nashorn could look like in Figure 4.2. The big benefit is that JRuby's current solutions can be reused, they would not have to go through another major change in their design to utilize Nashorn's type system.

Nashorn bytecode is an alternative compilation target to Java bytecode to be able to execute code on the JVM that specifically targets dynamic languages.

The same reasoning applies to all other existing dynamic language implementations on the JVM as well, including Nashorn's JavaScript front-end. And that is arguably independent of their current design, just generate Nashorn bytecode instead of Java bytecode.

4.1.6 Closures

Closures and lexical scopes have been left out from the Nashorn bytecode. The main reason for that is that it would have enforced design decisions made in Nashorn to the implementing language.

The scope objects in Nashorn are initialized and attached to the function in the generated bytecode when the `ScriptFunction` object is instantiated. The scope object is stored inside the function throughout its lifetime. The initialization of the function and the scope object is executed in the parent function where the function is declared in the JavaScript source code. In that way the parent scope can be set in the created scope object to be able to access variables in that as well.

With Nashorn's current object model, that is necessary to be able to implement JavaScript's lexical scope rules. After all, the set of accessible variables in a function is dependant of where the function was declared lexically.

That way of handling scopes is quite general and could probably be adapted to support other languages lexical scope rules as well. One problem though is that Nashorn would be in full control of the scope objects with the consequence that the language implementor would not have that control. Control of the scope objects is needed to be able to add properties and built in functions to them. Also, existing language implementations might

have a completely different approach to scopes than Nashorn. Even if they could adapt to Nashorn's way of handling scopes and builtins they might choose not to since it might require them to rewrite a solution that already works for them.

What if the language wants different scopes to be handled differently? For example Ruby has several kinds of scopes (local scope, class scope, global scope etc.) while JavaScript has only one. Although that is solvable it would be difficult to do it in a way that fits all dynamic languages and especially languages that already have a working implementation on the JVM. Instead, the lexical scopes and closures are completely left to the language implementors to handle in the way that suits them and their needs.

4.1.7 Dynamic linking

Linking is one of the places where the language implementor will need to be able to plug in functionality in Nashorn. How a call site is linked is closely related to the semantics of the language.

Nashorn uses Dynalink to handle the dynamic linking for JavaScript. Dynalink was first implemented as a standalone library for dynamic linking with `invokedynamic` and is not at all written for JavaScript specifically [22]. As described in Section 2.3.2, Nashorn uses dynalink by implementing its own linker which is then used by the Dynalink framework to link call sites. The nice thing about this is that a language implementor for another dynamic language could do just the same.

To provide a good interface to dynalink's linking mechanisms Nashorn bytecode will have an operation for each of dynalink's operations, see Section 2.3.2. One other reason for that is that since `invokedynamic` is a typed instruction it can not be used directly for call sites where types are unknown. An untyped `invokedynamic` instruction could be exposed but since dynalink will be used nevertheless, having support for dynalink's operation in Nashorn bytecode is a good alternative that does not limit flexibility.

A problem with linking is that linking in Nashorn is closely related to typing since Nashorn must handle compilation of type specializations. In Nashorn today functions are represented as `ScriptFunction` objects, those objects are JavaScript specific. They keep track of both the lexical scope and the JavaScript source code. It also initiates type specialization compilation when needed and caches the compiled code for future use. The code part of `ScriptFunction` could be made language agnostic, just by keeping a reference to the Nashorn bytecode that was generated from it instead of the JavaScript source code. Nashorn could provide an object that wraps the Nashorn bytecode that has a simple interface, like the one in Listing 4.6 with just a single function that returns a representation of a compiled function of a certain type. That representation should also contain the information needed to link to it.

```
1 interface NashornBytecodeFunction {
2     CompiledFunction getTypeSpecialization(MethodType type);
3 }
```

Listing 4.6: The interface used to compile type specializations could look like this, here represented as a Java interface

The benefit of this approach is that the language implementor gets the freedom to represent functions as they want which means they have the freedom to link it as they like as well. They just need to keep track of the `NashornBytecodeFunction` representation of the function and connect it to their internal function representation. That is unfortunate but no solution that did not require that and still give the same freedom to the language implementor has been found. A solution where Nashorn keeps track of the Nashorn bytecode would involve to force the function representation to be the one Nashorn uses internally. That would be more complicated to work with and force existing language implementations to diverge from their current design and solutions.

For Nashorn's JavaScript implementation this would mean that the `ScriptFunction`

object would handle the lexical scope rules in the same way as today. The difference is that it keeps track of the code and compiles type specializations differently. It would keep track of Nashorn bytecode instead of JavaScript code and also let Nashorn bytecode handle type specialization compilation.

Being able to recompile code from Nashorn bytecode rather than the language source code is a benefit in itself. It gives a cleaner separation between the front-end and the back-end of the compiler. It also relieves the language implementor from having to do tricks to account for recompilation in, for example, the parser.

In the TypeScript implementation described in Section 3.1 a few problems occurred that were caused by recompiling from the source code. Method `a` on line 2 in Listing 4.7 is an example of that. How to parse the method declaration is contextual, when parsed eagerly it is no doubt there is a method declared on line 2 since it resides in a class. But when recompiled the parsing starts at the method declaration which puts the parser out of context. Without any special treatment the three first characters would actually be parsed as a function call instead of a method declaration. It is possible to work around such issues but it is indeed better to not have to.

```
1 class A {
2     a():number {
3         return 5;
4     }
5 }
```

Listing 4.7: A contextual class method.

4.1.8 Type hierarchy

The type hierarchy in the optimistic type system uses four different types, the types and the way they can be deoptimized is shown in Figure 4.3.

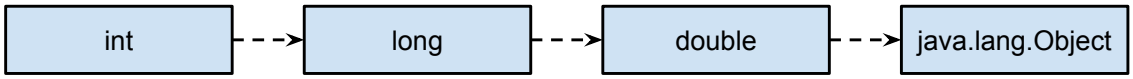


Figure 4.3: The type hierarchy in Nashorn, the arrows shows which type each type can be deoptimized to. Direct jumps from one node to another reachable node are implicit.

The types themselves can hold all values for every possible language since `java.lang.Object` is assignable by every Java object. The hierarchy for the numeric types is however not sufficient for all languages, currently in Nashorn for example an addition of two numeric types compiles to the bytecode in Listing 4.8 for each of the primitive types. The problem is that in the first two cases a dynamic invocation to a method that handles overflow is used while the double addition uses a regular `dadd` bytecode instruction. The reason for that is that a bytecode double can fit all JavaScript numbers so the overflow logic is not needed there [9]. A JavaScript number can however have the value `Inf` (infinity) just as a Java `double`.

```
1 invokedynamic iadd(II)I     // int addition
2 invokedynamic ladd(JJ)J     // long addition
3 dadd                        // double addition
```

Listing 4.8: Numeric additions compiled to bytecode

However, that is not true in all dynamic languages, Ruby for example has no limitations on how big a numeric value can become [5]. Ruby also has a separation between integer values and floating point values that JavaScript has not [5, 9]. To tackle the first problem it needs to be possible to fall back to a `java.lang.BigDecimal` or `java.lang.BigInteger`

representation of numeric values. The second problem is more related to the structure of the hierarchy, the hierarchy would have to be built up like in Figure 4.4 to be able to separate integer values from floating point values. Actually both `java.lang.BigInteger` and `java.lang.BigDecimal` can be assigned to `java.lang.Object` so the same bytecode could be used for the three of them. They are in the figure to emphasize that they need to be used and that they also might require special treatment to invoke the arithmetic operations.

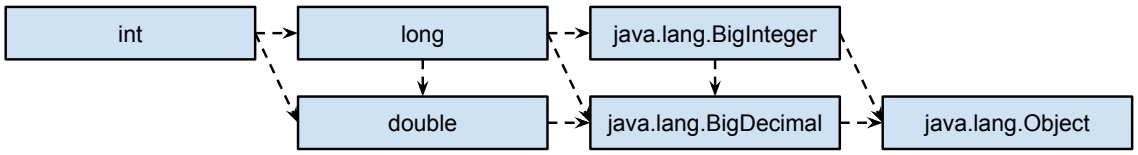


Figure 4.4: The type hierarchy needed to model Ruby, the arrows shows which type each type can be deoptimized to. Direct jumps from one node to another reachable node are implicit.

To make this possible the language implementor has to specify how the *type graph* looks like for the specific language. The nodes in the graph are the bytecode types and the edges are conditions that cause recompilation. Such conditions can for example be `int` or `long` overflows, `double`-values that become `Inf` or that references are assigned to places where the code is compiled with narrower types. This is already abstracted fairly well but the flexibility needs to be improved.

4.1.9 Type boundaries

Section 3.2 shows the results of how TypeScript performs on Nashorn compared to JavaScript. The performance and warmup time improvement is due to the fact that the TypeScript implementation sets type boundaries to expressions.

The question is, would it be worth supporting setting type boundaries on operations in Nashorn bytecode? There are three things to consider here:

1. How big is the performance gain?
2. How big of an effort would it be to implement support for type boundaries?
3. How useful would it be?

The first question is answered in Section 3.2. The results show that it does have an impact on the performance when optimistic types are disabled. The performance gain is almost up to 20% for some benchmarks. With optimistic types, however, the performance of the stable state is not really effected but that was expected. However, warmup time was effected, the execution time of the first iteration is significantly reduced for three out of the four benchmark. After that the difference fades away quickly but on the other hand, the biggest warmup issue is tied to the first iteration and for large applications the improvement could matter. Although improved, the first iteration is still magnitudes slower that the following ones so warmup is still a big issue.

The second question is also answered by the TypeScript implementation. It proved to be really easy to extend the type system to be able to set boundaries with just a few lines of code changed in each effected class, see Section 3.2.4.

The third question might also be the most relevant. Dynamic languages are dynamically typed and in many cases that means that there are simply no type information available to be able to set type boundaries. For JavaScript, Ruby [5] and Groovy [10] that is the case. Clojure however has support for type annotation and could utilize the type boundaries [8]. PHP has its type hints for function parameters that could make use of boundaries [7], but that is limited to function parameters only. TypeScript can also make use of the boundaries as shown by the TypeScript implementation in this thesis.

So the usefulness is limited and in the general case, dynamic languages will not be able to utilize the boundaries. The limited improvements in warmup time is also something that speaks against supporting type boundaries. This means that supporting type boundaries should not be first priority but given the small effort it would require it might still be worth implementing at some point.

4.1.10 Static support

There is no conflict between having untyped operations in Nashorn bytecode and still provide typed equivalents to the typed Java bytecode instructions. In most cases typed operations would not be very useful since the dynamic behaviour of a dynamic language cannot be modelled with them. But they can be useful for handling of internal objects.

Listing 4.9 shows a JavaScript construct that is modelled with `invokestatic` in Nashorn. The compiled bytecode can be seen in Listing 4.10. That shows that the static method `TYPEOF` on the `ScriptRuntime` class is invoked to execute JavaScript's `typeof`-operator.

```
1 var a = 5;
2 var aType = typeof a;
```

Listing 4.9: JavaScript's `typeof`-operator is implemented with an `invokestatic` invocation

```
1 invokedynamic dyn:getProp|getElem|getMethod:a(Object;)Object;
2 invokestatic ScriptRuntime.TYPEOF(Object;Object;)Object;
```

Listing 4.10: Bytecode compiled from the JavaScript code in Listing 4.9

There are other usages for `invokestatic` as well, there is no reason to remove the possibility to create such solutions and therefore those instructions are supported by Nashorn bytecode. Even direct use of `invokedynamic` is allowed but then the language implementor can not rely on Nashorn's type system to assign types to it.

4.1.11 Representation and API

Nashorn bytecode could be represented in a number of different ways. In this section three different representations is considered, binary representation, textual representation and Java object representation. Benefits and disadvantages of each of them are discussed.

Binary

Nashorn bytecode could be represented in a binary form just like Java bytecode. Since the instruction set would be similar between the two, most of the design choices done in Java bytecode could be replicated.

The main benefit with a binary representation would be that a program can be expressed in Nashorn bytecode in a compact way. Since the Nashorn bytecode will have to be kept in memory to be able to recompile it, that could actually have a big impact on Nashorn's memory usage.

Textual

The main benefit with a textual representation is that it is human readable and that could simplify for the language implementors that are generating Nashorn bytecode. A textual representation would not be as compact and memory efficient as a binary representation. In Nashorn today, the JavaScript source code is kept in memory textually so keeping a textual representation of Nashorn bytecode is feasible.

LLVM supports both textual and binary representation of its intermediate representation [13].

Java objects

The last option discussed is to represent the code as plain Java objects that are connected to each other in sequences. That is in fact a binary representation but is not as compact as the one discussed above. A Java object representation is the least compact of the three.

This representation would not be human readable either but has the benefit of not needing to be parsed. For Nashorn to be able to work with the Nashorn bytecode it would have to construct this kind of representation no matter how it is represented between compilations. In that sense it might be more efficient to use a Java object representation. The memory issue could be somewhat limited by serializing the objects between compilations.

API

Each representation has its advantages and disadvantages. One aspect to keep in mind though is that for the language implementor, what really matters is the interface Nashorn provides to develop against.

That API could, and probably should, be designed so that the internal representation in Nashorn does not matter. Given the previous design discussion Nashorn will have to provide an API with support for four tasks:

1. Construct Nashorn bytecode
2. Plug in logic for basic operations such as `add`, `mul` and `div`
3. Plug in a language specific type hierarchy
4. Define its own Dynalink linker

The first could be done by exposing a library that constructs Nashorn bytecode programmatically. Similar to what ASM Java bytecode construction library does [2] that Nashorn uses, except that it would generate Nashorn bytecode instead of Java bytecode. Whatever representation that produces in the end does not matter to the language implementor and could even be changed if the chosen one turns out to not fulfill the performance and/or memory requirements.

The second could be done either through the same API as the code generation or a separate one. Same reasoning applies here, what is important is that Nashorn provides a good API not how it is represented internally.

The third one could also be designed independent of representation and might be done with a separate API.

The last point already has an interface, namely the interface that dynalink provides and it could be used as is.

Given this discussion, the choice of how to represent Nashorn bytecode is left open to decide in the future since it does not really affect the design discussed in this thesis.

4.2 Results

In this Section the new architecture for Nashorn is presented, the architecture is based on the discussions in Section 4.1

4.2.1 Operations

Nashorn bytecode is an extension of Java bytecode and therefore supports all instructions listed in *The Java Virtual Machine Specification* [14] with the same syntax and semantics. Based on the discussion in Section 4.1.10, even though Nashorn bytecode adds support for untyped instructions it still supports typed instructions from Java bytecode.

Additional operations

Apart from the Java bytecode instructions, Nashorn bytecode has the untyped operations listed in Table 4.1 and the Dynalink equivalent operations in Table 4.2.

The operations in Table 4.1 are untyped versions of already existing Java bytecode instructions that lets Nashorn handle type assignment and generation of actual Java bytecode. Not all the operations listed are subject to failed optimistic assumptions. For example the compare instructions *cmpg* and *cmpl* only produces values -1, 0 or 1 and can therefore never overflow an `int`. The reason they are there is that since the compiler front-end does not know the types of the operands, it would not know which Java bytecode instructions to use if they were not present.

The dynalink operations in Table 4.2 are just equivalents to the already existing dynalink operations discussed in Section 2.3.2.

Table 4.1: Untyped equivalents to Java bytecode instructions

Operation	Stack	Description
add	..., *value*1, *value*2 → ..., *result*	Addition, $result = value1 + value2$
sub	..., *value*1, *value*2 → ..., *result*	Subtraction, $result = value1 - value2$
mul	..., *value*1, *value*2 → ..., *result*	Multiplication, $result = value1 \cdot value2$
div	..., *value*1, *value*2 → ..., *result*	Division, $result = value1 - value2$
rem	..., *value*1, *value*2 → ..., *result*	Modulo, $result = value1$ / $value2$
and	..., *value*1, *value*2 → ..., *result*	Bitwise and, $result = value1$ & $value2$
or	..., *value*1, *value*2 → ..., *result*	Bitwise or, $result = value1 \mid value2$
xor	..., *value*1, *value*2 → ..., *result*	Bitwise exclusive or, $result = value1 \hat{} value2$
shl	..., *value*1, *value*2 → ..., *result*	Bitshift *value*1 left by *value*2 bits
shr	..., *value*1, *value*2 → ..., *result*	Arithmetic bitshift left, shift *value*1 by *value*2 bits
ushr	..., *value*1, *value*2 → ..., *result*	Logical bitshift right, shift *value*1 by *value*2 bits
neg	..., *value*1 → ..., *result*	Negate *value*1
load	... → ..., *result*	Load a local variable to the stack
store	..., *value*1 → ...	Store *value*1 as a local variable
cmpg	..., *value*1, *value*2 → ..., *result*	Compare two numeric values, push 1 if *value*1 is greater than *value*2, 0 if equal and -1 if *value*1 is less than *value*2. If at least one of the two is NaN, push 1
cmpl	..., *value*1, *value*2 → ..., *result*	Same as cmpg but push -1 if at least one of *value*1 and *value*2 is NaN

4.2.2 Pluggable behaviour

This section covers the different behaviours of Nashorn bytecode that need to be plugged in by the language implementor. The description is biased towards the general design discussed in Section 4.1 rather than implementation specific details.

Table 4.2: Operations equivalent to Dynalink's operations.

Operation	Stack	Description
get_prop	..., *value* → ..., *result*	Get a property of *value*
get_elem	..., *value* → ..., *result*	Get an element of *value*
get_method	..., *value* → ..., *result*	Get a method of *value*
set_prop	..., *value* → ..., *result*	Set a property of *value*
set_elem	..., *value* → ..., *result*	Set an element of *value*
call	..., f, [$arg1$, $arg2$, ...] → ...	Invoke f
new	..., f, [$arg1$,$arg2$, ...] → ..., *result*	Invoke f as a constructor

Operators

The problem with operators is discussed in Section 4.1.3. It consists of the compiler front-end not being able to chose Nashorn bytecode operations depending on the type of the operands since that information is not available at compile time. That problem applies to all Nashorn bytecode operations listed in Table 4.1.

For each operation the language implementor has to specify two things:

1. Depending on the types of the operands, what is the return type of the operation.
2. Depending on the types of the operands, how to perform the operation.

How the API for this could look like is discussed shortly in Section 4.1.11, the details of the API is implementation specific and is left out of this thesis.

A suggestion however, provided that Nashorn bytecode is represented with Java objects, is to use Java class inheritance to implement language specific operators. That would mean that an abstract class or interface is provided by the API that requires the language implementor to specify what is required, the interface for the add-operation could, for instance, look like the one in Listing 4.11. Then the language specific operator class would be used when constructing the Nashorn bytecode representation.

```
1 public interface NashornAdd {
2     Type getReturnTypeFromOperandTypes(Type lhs, Type rhs);
3     BytecodeSnippet getEmittedBytecodeFromOperandTypes(Type lhs, Type rhs);
4     /*
5      * "BytecodeSnippet" is in this case just a container for bytecode,
6      * it is not a real class and any existing or new
7      * class with the same purpose could be used instead
8      */
9 }
```

Listing 4.11: Possible interface for the add-operation.

Type hierarchy

The type hierarchy needed for the optimistic type system might differ between dynamic languages and needs to be specified by the language implementor. The reason for that is discussed in more details in Section 4.1.8.

The type hierarchy is a graph where the nodes in the graph are the bytecode types and the edges are conditions that cause recompilation. Such conditions can for example be `int` or `long` overflows, `double`-values that become `Inf` or that references are assigned to places where the code is compiled with narrower types.

How the API for this should look like is also implementation specific and left out of this thesis.

Linking

Linking is handled with the dynalink equivalent operations listed in Table 4.2. To make use of those the language implementor will have to implement its own dynalink linker just like Nashorn does for JavaScript currently. How that is done is out of the scope of this thesis, the interested reader is referred to the dynalink documentation[1].

4.2.3 Construction

Nashorn will also have to provide a library for constructing Nashorn bytecode. The details of how that could be designed is out of the scope of this thesis. In Section 4.1.11 it is suggested that it can be designed similarly to ASM bytecode construction library but with support for the new Nashorn bytecode instructions.

[1]Dynalink's documentation: `https://github.com/szegedi/dynalink/wiki`

Chapter 5

Conclusions

5.1 TypeScript implementation

The performance and warmup measurements on the TypeScript implementation show that the type annotation indeed can be used to increase the performance of Nashorn. Without optimistic type guessing enabled there were improvements of up to almost 20% on some benchmarks, mainly the number intensive benchmarks. Others showed only a small performance increase.

With optimistic type guessing the performance of the stable state was not expected to increase but instead warmup time was expected to improve. While that was the case for most benchmarks the difference was not as big as hoped for and warmup time is still an issue. The execution time of each iteration for the JavaScript benchmarks catch up with the TypeScript performance after only a few iterations.

This could be a result of the benchmarks used for the measurements being too small, the warmup time grows approximately linearly with the code size. It could also be that the optimistic type system in Nashorn does a good job at deoptimizing functions with help of the runtime information available when an optimistic assumption fails, and by that reducing the need of further deoptimizations. The truth probably lies somewhere in between the two. Measurements on bigger benchmarks, such as Mandreel from the Octane suite [6], would probably give clearer results on which of the two reasons has the biggest impact.

Some observations from implementing TypeScript on Nashorn was used when designing the intermediate representation for Nashorn. Being able to set type boundaries is one such observation. Although it is not concluded that this should be a part of Nashorn bytecode it is discussed that it might be worth supporting despite the rather small performance gain because of the small effort it would require to implement it.

Another observation was that the optimistic type system is well designed and fits well for other dynamic languages as well. The only problems are that it operates on a JavaScript AST which is difficult to model other languages on and that the type hierarchy can not be used for all languages. With Nashorn bytecode it will operate on that representation instead to make it reusable for other dynamic languages.

5.2 Architecture

A new architecture for Nashorn has been presented that enables reuse of Nashorn's type system. The new design includes an intermediate representation that is similar to bytecode but with additional untyped instructions. It is therefore referred to as Nashorn bytecode in this thesis. Nashorn should provide an API that language implementors can use to construct Nashorn bytecode and execute it on Nashorn.

Initially it was not clear exactly what to handle in Nashorn behind the intermediate representation and what to leave for the language implementor to handle on their own.

Managing closures and lexical scopes was considered but was intentionally left out. There are two main reasons for that. First, scope rules differs a lot between languages and it would be difficult to model in a way that fits all languages. Second, there are a lot of existing dynamic language implementations on the JVM which could benefit from Nashorn's type system. By keeping Nashorn bytecode as close to Java bytecode as possible it gets easier for them to utilize Nashorn's type system. It does not require the same effort as if Nashorn forced a lot of solutions on them, instead they can use their current solutions when it comes to representing functions and classes, perform optimizations in the compiler etc. They would only have to rewrite the code generation to generate Nashorn bytecode instead of Java bytecode.

Section 4.1.4 mentions semantic analysis for dynamic languages and that it can not be fully performed in a traditional way at compile time since the information needed is often not available. Instead, parts of the semantic analysis need to be performed at runtime where there is more information available. Because of that, the new architecture supports plugging in language specific semantics in Nashorn's runtime environment in three situations. The first is basic operations where types are needed to be able to determine what the operation actually means. The second is dynamic linking, the language implementor gets full control on how to link call sites, and the third is the type hierarchy.

Implementation specific details are not discussed in great details in this thesis but rather it focuses on identifying problems and present possible solutions for them. When this new design gets implemented, new problems will probably arise that need to be solved but the biggest issues are hopefully identified and a general design that should work is presented in this thesis.

Nashorn bytecode differs from existing intermediate representations. JRuby's IR is the most similar in the sense that it is also designed for a dynamic language on the JVM. But that is written for Ruby only and has Ruby specific semantics built in. The representation is also on a somewhat higher level since it includes lexical scope rules for Ruby.

LLVM also provides a IR designed to be used by multiple languages. It also differs from Nashorn bytecode in many ways. First, it is not written for JVM only, LLVM has multiple back-ends that target different platforms. Second, it is not designed to handle problems specific for dynamic languages. It is, however, possible to use LLVM to implement a JIT-compiler for a dynamic language but it does not have an execution model with built in support for dynamic languages as Nashorn bytecode has.

The fact that Nashorn bytecode is intended for the JVM and for dynamic languages only had big impact on the final design. Since the JVM is its only target platform it has been designed to overcome issues that exist on that platform which are mainly dynamic typing and dynamic linking. That also ties to the fact that it is designed for dynamic languages only. Had it been intended for static languages also those two issues might not have been as prioritized. And of course, the fact that Nashorn bytecode is an extension of Java bytecode is a result of the JVM being the targeted platform as well.

The solution presented in this thesis is by no means the only possible way to achieve the same goals and there are probably an endless number of way it could be done. The design solutions are however motivated and discussed to make it clear why the chosen approach is a good one.

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, United Kingdom, 2nd edition, 2004.

[2] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[3] Thomas E. Enebo. JRuby 9000 project update. `http://rubykaigi.org/2014/presentation/S-ThomasE.Enebo`, 2014. RubyKaigi.

[4] Steve Fenton. *TypeScript for JavaScript programmers*. Swift Point Press, 2nd edition, April 2013.

[5] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language, everything you need to know*. O'Reily, San Fransisco, California, 1st edition, January 2008.

[6] Google. Octane benchmarks. `https://developers.google.com/octane/`. Google Developers.

[7] PHP Documentation Group. Php.net, type hinting. `http://php.net/manual/en/language.oop5.typehinting.php`.

[8] Stuart Halloway and Aaron Bedra. *Programming Clojure*. The Pragmatic Programmers, LLC, USA, 2nd edition, April 2012.

[9] ECMA International. Standard ECMA-262 5.1 edition. `http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf`, June 2011.

[10] Dierk König, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy In Action*. Manning Publications Co, 1st edition, 2007.

[11] Marcus Lagergren. The JVM as a multi language runtime. `http://vimeo.com/80848494`, 2013. GeekOut.

[12] Marcus Lagergren and Marcus Hirt. *Oracle JRockit, The Definitive Guide*. Packt publishing, 1st edition, June 2010.

[13] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The java virtual machine specification. `http://docs.oracle.com/javase/specs/jvms/se8/html/`, Mar 2014.

[15] Dan Maharry. *TypeScript Revealed, discover Anders Hejlsberg's new way to make JavaScript scalable and easier*. Apress, 1st edition, 2013.

[16] Microsoft. TypeScript language specification. http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf, April 2014.

[17] Charlie Nutter. A first taste of invokedynamic. http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html, September 2008.

[18] Oracle. Javadoc for java.lang.invoke http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html.

[19] Oracle. JMH benchmark tool. http://openjdk.java.net/projects/code-tools/jmh/. OpenJDK.

[20] Oracle. JSR 292: Supporting dynamically typed languages on the Java™ platform. https://jcp.org/en/jsr/detail?id=292, February 2006.

[21] Emma Söderberg, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.*, 78(10):1809–1827, 2013.

[22] Attila Szegedi. Dynalink - dynamic linker framework for JVM languages. http://medianetwork.oracle.com/video/player/1113272541001, 2011. JVMLS.