Compiling Dynamic Languages

Anders Schlichtkrull, Rasmus T. Tjalk-Bøggild

Kongens Lyngby 2013
Compute-B.Sc.-2013-15

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, Building 303 B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253031
compute@compute.dtu.dk
http://www.compute.dtu.dk/ Compute-B.Sc.-2013-15

Abstract

The purpose of this thesis is to examine if ahead-of-time compilation is a viable solution for executing programs written in Dynamic Languages when compared to interpretation and just-in-time compilation, and to find out how such a solution can be made.

We achieve this by first describing and classifying Dynamic Languages in terms of their type systems and programming concepts. Specifically we will show how the Dynamic Languages differ from Statically Typed Languages.

We then build an ahead-of-time compiler, called the project compiler, for a subset of JavaScript. The subset is large enough to constitute a Dynamic Language and large enough to run an industry standard benchmark. The ahead-of-time compiler can also do optimizations. Furthermore, we benchmark this compiler implementation and compare it to the benchmark results for two contemporary JavaScript implementations: Rhino which is a JavaScript interpreter and compiler, and V8 which is a JavaScript just-in-time compiler. We also profile the implementation to measure the effect of the implemented optimizations.

Based on these results we find that the performance of the project compiler is better than that of the interpreter, but the performance is worse than that of the two other compilers. The performance of the project compiler is influenced negatively by its implementation of JavaScript values in memory. We also find that the implemented optimizations give significant performance benefits for simple examples, but do not improve the performance significantly when running more complex benchmarks.

We conclude that ahead-of-time compilation is a viable alternative to interpretation of dynamic languages, but based on the collected results we are unable to make any generalizations with regards to ahead-of-time compilation compared to just-in-time compilation.

Contribution of work We have both contributed equal amounts of work to the project.

Acknowledgements

We would like to thank our supervisors Christian W. Probst and Sven Karlsson for the chance to work on this project as well for their valuable input and feedback.

We would also like to thank our families for their understanding and support throughout the project period.

Contents

CHAPTER 1

Introduction

Dynamic languages, and in particular JavaScript, are among the most widely used programming languages [Wel13] [Sof13] [Git13]. The dynamic languages have many features considered beneficial to the programmer such as a type system that does not impose restrictions on the types at compile time, easy access to data structures and concepts such as meta programming, higher-order functions and object oriented programming. Furthermore, the languages are used in many different contexts including web services, game scripting and rapid prototyping [OZPSG10].

Classically, dynamic languages have been interpreted, that is, evaluated at runtime by an interpreter program. In recent years, dynamic languages have also been complied, that is, translated to another language such as native code before being executed. In particular, the compilation model for translating the program to native machine code, known as just-in-time compilation has been used. In this model the compilation for a section of the program is performed just before it is used. This allows for faster start-up times than compiling the entire program ahead of time when a compiled version of the program is not available. These approaches especially make sense for executing JavaScript when used for client side web-programming, where it is important that the execution can start immediately.

However, Dynamic languages, including Javascript, are also used on servers, where the same code is run many times. For this purpose it would make sense to do the translation to native code only once, so that the run-time is only used on the actual execution of the program code. This technique is called ahead-of-time compilation.

The purpose of the project is to build an ahead-of-time compiler for the dynamic programming language JavaScript and measure the performance of this compiler against contemporary implementations. Specifically, the project will examine if ahead-of-time compilation is a viable solution when compared to both interpretation and just-in-time compilation.

To do this, an overview of the major contemporary dynamic languages is given first. The languages are introduced with a specific focus on how they differ from statically typed languages as well as the special requirements that the dynamic languages have during compilation and run-time. JavaScript is then placed in this framework to describe how it fits among the dynamic languages.

With JavaScript as the base, the major moving parts of a compiler is then analysed with a focus on how specific dynamic aspects of the language are translated.

The actual implementation of these aspects in the project compiler is then described as well as how the optimization of the code is performed.

Finally, the project compiler is benchmarked. Firstly, to measure the performance against contemporary implementations (both a JavaScript interpreter, a JavaScript ahead-of-time compiler and a JavaScript just-in-time compiler). Secondly, to measure the effect of the optimizations implemented.

Based on these benchmarks we will evaluate the viability of the compiler when compared to contemporary JavaScript implementations.

Chapter 2

Preliminaries and theory background

This chapter will provide the theoretical background needed to understand the report. The central terms of the report will be defined in a dictionary. Furthermore it will give an overview of the subject of *Dynamic Languages* and compiling.

2.1 Dictionary

The purpose of the dictionary is to establish unambiguous definitions of the central terms in this paper, to avoid the confusion sometimes associated with some of the terms.

Compiler A compiler is a computer program that takes a program written in a programming language called the source language and translates it to another language called the target language. The translation should not change the meaning of the program.

Interpreter An interpreter is a computer program that takes a program written in a programming language, and executes the instructions of the program one by one.

Many well known interpreters also include a form of compiling, because they start by taking the source language and translate it to an intermediate language, and then do the interpretation on that language. [ALSU86, p. 2]

Type A type can be seen as a set of values. An example is the integer type defined in many programming languages. The integer type usually represents the integers in the interval $[-2^{31}; 2^{31} - 1]$.

Run-time Anything that happens during the execution of the program is said to happen at run-time.

Compile-time Anything that happens before the execution of the program is said to happen at compile time.

Statically typed language A statically typed language is a language in which the types of expressions and variables are known at compile-time [JGB12, p. 41]. That a variable has a type, means that only values of that type can be assigned to the variable. That an expression has a type, mean that the expression will evaluate to a value of that type.

Dynamically typed language A language in which the types of expressions and variables are not, in general, known at compile time.

Note that there is a distinction between dynamically typed languages and dynamic languages.

Strongly typed language A language where the types of the operands limit the operators that can legally be applied to them. In Java, for instance, the input to a "+" operator is limited to strings and numbers, and the produced result depends on the types [JGB12, p. 41].

Weakly typed language A language where the types of the operands do not limit the allowed operators.

An example is perl language, where the input to an operator or function is implicitly converted if needed. The the "." operator takes two strings and concatenates them. This language will implicitly convert any input to strings before applying the operator. Conversely the "+" operator will convert operands to numbers. So $\$foo = "10" + "20"$; will yield the value 30, where as $\$foo = 1\ .\ 2$; will yield the value "12" [perl3c, p. 5]. The conversion is called type coercion.

Implicitly typed language A language where the programmer does not have to annotate variables or functions with types.

Explicitly typed language A language in which the variables and functions must be annotated with their type.

Boxed / Unboxed variables A variable is considered to be boxed, if it its value is stored at run-time along with type information. An unboxed variable stores only the value itself.

Meta-programming Meta-programming is the querying and manipulation of a program by the program itself [Tra09]. Often the definition is extended to include the creation of other programs, but for the purpose of this project, the definition is limited to programs that query and manipulate themselves.

Scope The scope at a point in the program is the set of variables that can be accessed at that point.

Closure The closure of a function consists of the body of the function and its environment. The environment is a mapping of the variables from their identifiers to the memory location of their values.

Classical inheritance The object oriented inheritance model used in languages such as Java or C++. In this inheritance model, object classes inherit all visible methods and fields from their *super class* [JGB12]. A class is allowed to *override* a method of a super class and thereby provide a new implementation for it.

Prototype inheritance In the prototype inheritance model, each object may have another object assigned as its *prototype*. This object may again have a prototype assigned - this is referred to as the prototype chain. An object may define its own methods and fields, but will also inherit any methods or fields defined in its prototype chain. To the outside user there is no difference between the objects own methods and fields and the methods and fields in the prototype chain. An object may provide its own definition for methods or fields defined higher in the prototype chain [Per13e].

2.2 Overview of Dynamic Languages

The purpose of this overview is to establish the common characteristics of the *Dynamic Languages*. We will do this by first examining a subset of the *Dynamic Languages* in detail and then from this, extract the common characteristics. These common characteristics rather than a formal definition will form our understanding and use of the term *Dynamic Language* in this paper.

2.2.1 Included languages

For the languages in this overview we wanted to include the dynamic languages most widely used today. We furthermore wanted to focus on contemporary languages, and avoid including domain specific languages like SQL or Matlab.

For meassuring use of the different programming languages, we used published lists of programming language usage [Wel13], [Pau07]. The languages included in the overview are the dynamic languages among the 10 most widely used languages in the lists.

2.2.2 JavaScript

JavaScript is a scripting language designed primarily to be used in a Web Browser context, but the language has seen some use in server side applications in recent years [nod13] [ecm11, p. 2].

JavaScript is a *Dynamically typed*, *Weakly Typed* and *Implicitly typed* language [ecm11, p. 42]. The language defines a small set of build in types (Boolean, Number, String, Object, Null and Undefined), but has a *Prototype-Inheritance* system which allows the user to define subtypes of the object type [ecm11, pp. 2-3].

The language supports a special syntax for initializing arrays and objects [ecm11, pp. 63 - 65]. Objects in JavaScript also function as dictionaries, a data structure that maps strings to other values. The arrays in JavaScript are resizable.

In JavaScript, functions are objects [ecm11, p. 7]. This means that anywhere where an object may be legally used in an expression or as an argument, a function may be used instead. That functions are proper objects means that the language supports *Higher-Order Functions* which are functions that take other functions as input and/or return a function as output. Functions are also used to create scope and functions form *closures* over the context in which they were created [ecm11, p. 42].

The language also supports *Meta Programming*, since all objects can be queried, that is inspected for what properties they contain and what type they have. Furthermore, object properties and functions can be overwritten, removed and added at run-time - with the exception of a few native functions [ecm11, p. 3-4].

JavaScript is specified in the ECMAScript Language Specification [ecm11].

2.2.3 Python

Python is a scripting language that has seen heavy use in the industry, with Google being one of its most prominent users. Among the uses are web programming, motion picture visual effects, game development and education [Fou13].

Python is a *Dynamically typed*, *Strongly Typed* and *Implicitly typed* language [pyt13a, 3.1]. Every type in the language is an object, including booleans, null objects, numbers, strings, functions and the classes themselves [pyt13a, 3.1]. This means that the language supports *Higher-Order Functions*. The language allows the user to extend the types by defining new classes [pyt13a, 3.1].

Python also supports a special syntax for initializing resizable arrays (lists) and diciontaries [pyt13a, 3.2].

Python, like Javascript, supports *Meta Programming*, since all objects can be queried at run-time, and properties and functions of classes can be overwritten, removed and added at run-time [pyt13b].

2.2.4 Ruby

Ruby is a scripting language used primarily for server side web applications. It is used extensively with the Ruby-On-Rails framework [abo13c].

Ruby is a *Dynamically typed*, *Strongly Typed* and *Implicitly typed* language [rub10, pp. 12, 17]. Like Python, the language treats all types as objects, including booleans, null objects, numbers, strings and the classes themselves [rub10, pp. 14, 150 - 156]. Functions are not objects, but *Higher-Order Functions* are allowed using the Block structure [rub10, pp. 13]. The language allows the user to extend the types by creating new classes.

Ruby has a simple build in syntax for initializing arrays and dictionaries [rub10, p. 104]. The arrays in Ruby are resizable [rub10, p. 194].

Ruby also supports *Meta Programming* in the form of querying and updating object properties and functions.

2.2.5 Perl

Perl is a scripting language that has a history of being widely used for text processing and server side web programming [abo13a].

Perl is a *Dynamically typed*, *Weakly Typed* and *Implicitly typed* language [per13a, p. 2]. The language requires the user only to define if a variable is a scalar, an array of scalars (list) or a hash of scalars (dictionary). A scalar can be of the type number, string or reference, where reference can be anything from an object reference to a file handle or database connection [per13a, p. 2]. The language supports a special syntax for initializing resizable arrays and dictionaries [per13a, pp. 7-9].

Originally the language did not allow the user to extend the type system, but recent versions of perl allow the user to define classes of objects [per13b]. The language also supports *Higher-Order Functions* [per13d].

Perl also supports *Meta Programming*, but does this through the use of the eval function [eva13] in the standard library, which can evaluate any string as part of the programs source code. This qualifies as *Meta Programming* since the program can inspect and manipulate the string.

2.2.6 PHP

PHP is a scripting language originally designed for serverside web development and is also primarily used for this. The language can be embedded directly in HTML code and contains many build in funtions for webdevelopment [MAso13, Chapter: Preface], [MAso13, Chapter: Features].

PHP is a *Dynamically typed*, *Weakly Typed* and *Implicitly typed* language. The build in types are booleans, integers, floating point numbers, strings and resizable arrays. Actually, the array type is a dictionary from integer or string values to values of any type [MAso13, Chapter: Language Reference.Types]. The language also supports *Higher-Order Functions* [MAso13, Chapter: Functions.Anonymous functions]

PHP has a simple build-in syntax for initializing arrays. [MAso13, Chapter: Language Reference.Types].

In addition to these types, the user of the language can make new types by making classes. These classes can then be instantiated to objects of that type. Furthermore the user can make a class structure with inheritance, abstract classes and object interfaces [MAso13, Chapter: Language Reference.Classes and Objects].

PHP also supports *Meta Programming* by allowing classes to be queried at run-time, and properties and functions of them to be modified at run-time [GMM].

2.3 Common characteristics for Dynamic Languages

Based on the classification of the 5 Dynamic Languages reviewed above, we want to extract common characteristics for Dynamic Languages. Based on the type systems we see that all the reviewed languages are *Dynamically typed* and *Implicitly typed*. *Dynamically typed* means that the types of expressions and variables are not known at compile time. Therefore it makes sense to make the language *Implicitly typed*, because it means the programmer is not required to annotate variables and functions with types that cannot be guaranteed at compile-time anyway.

However, as Python and Ruby show, Dynamic Languages can be *Strongly Typed*, which means that though the types of the operands provided to an operator are not known at compile-time, the program can still produce type errors when values of the wrong type are given to an operator. The error is just thrown at run-time instead of compile-time. Even in the *Weakly typed* languages, the values will have types, but the interpreter or compiler will ensure that they are automatically coerced to a valid type according to the operations performed on them.

Another defining characteristic for the *Dynamic Languages* reviewed here, is that they all support *Higher-Order Functions* and *Meta Programming.* The exact mechanism for *Meta Programming* varies a bit among the languages, but they all provide easy access to some way of querying and manipulating the program itself.

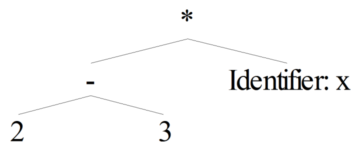
The reviewed languages also all support some special syntax for initializing common data structures like dynamic arrays and dictionaries. These data structures are a part of the language instead of being libraries. This gives the programmer easy and quick access to these data structures and in some cases, like JavaScript, form the basis for a concise data transfer format (JSON) [ecm11, p. 203].

2.4 A typical Compiler Design

A compiler traditionally has the following phases: Lexing, Parsing, Context Handling, Intermediate Representation Generation, Intermediate Representation Optimization and Target Code Generation. A compiler can be either broad or narrow. A broad compiler reads the entire source program to memory and works on this whereas a narrow compiler only reads a section of the program at a time and works on that. Broad compilers require more memory than the narrow compilers, but are usually faster and simpler to write. The broad compilers usually require only a single pass over each phase, whereas the narrow compilers require at least one pass over all phases for each section read.

Lexing and parsing The goal of lexing and parsing is to take the source code, which is a string of characters and translate it first to a stream of tokens (lexing) and then to an abstract syntax tree (parsing). The meaning of a program is defined from its syntactic structure. Therefore, it is this structure that the compiler needs to know, to translate the program to the target language. The abstract syntax tree is a concise tree representation of this structure [GBJL00][p. 52].

The AST for an expression such as `(2-3)*x` could for instance be:



Context handling Even if a program follows the syntax of the programming language, it might still be incorrect. For instance, many languages have restrictions on the names of the identifiers or want to ensure that the programmer does not use variables or functions that are not declared. Restrictions such as these are called context conditions [GBJL00][p. 194].

Context handling is also used for annotating the nodes of the syntax tree with semantical information about nodes such as if the expressions contain function calls. This information is used to help the intermediate representation generation, intermediate representation optimization and target code generation [GBJL00][p. 9].

Intermediate representation generation The goal of the intermediate representation generation is to take the abstract syntax tree and translate it to the intermediate representation. The intermediate representation lies between the source and target code, and a good intermediate representation will often be one that shares some properties with the source language and some properties with the target language.

Many compilers have multiple intermediate representations, to make the translations simpler, and because some optimizations are easier to do on a representation that is close to the source language, and others are easier to do on a represenation that is close to the target language.

Intermediate representation optimization The intermedate representation optimization changes the intermediate representation in such a way that it will produce the same results as before, but optimizing some parameter such as runtime or memory usage.

Although it is called optimization, the resulting code is usually not optimal, but rather improved along the optimzation metric.

Target code generation The target code generation will produce the target code from the intermediate representation. The closer the intermediate representation is to the target language, the simpler this part of the program will be.

2.5 Special requirements for Dynamic Languages

The purpose of this section is to introduce the special requirements that *Dynamic Languages* have with regards to their parse/compile and run-time phases. This section will focus on the differences between *Dynamic Languages* and *Statically typed languages.*

2.5.1 Delayed type checking

The major defining feature of the *Dynamic Languages* is the delayed type checking. Where a *Statically Typed Lanagues* will perform the type checks at compile-time the *Dynamic Languages* will have to perform these checks at run-time. This has the immediate effect that type information has to be integrated into the program - where purely *Statically Typed Languages* do not have to keep track of this information [GBJL00].

2.5.2 Type coercion

Although only relevant to *Weakly Typed Lanagues*, the type coercion in *Dynamic Languages* requires special attention. Where type coercions in *Statically Typed Languages* are solved at compile-time and by inserting the relevant casts into the outputted code [GBJL00, pp. 455-456], the *Dynamic Language* must resolve this at run-time, because the types of the operands may change during the course of the program.

2.5.3 Closures

Since the examined *Dynamic Languages* support *Higher-order functions*, closures need to be addressed. The presence of closures in a language complicates the function call logic, because functions are not only defined by their parameters and body, but also the environment in which they were declared. This means, for instance, that local variables cannot necessarily be discarded when returning from a function because there might exist a closure that keeps a reference to them.

2.5.4 Build-in data structures

The *dynamic languages* studied in this project all support dictionaries and resizable arrays. The build-in data structures are therefore more sophisticated than those of languages that only have arrays with a fixed size, and where libraries implement dictionaries and resizable arrays. When compiling *dynamic languages*, these build-in data structures need to be addressed.

2.5.5 Garbage Collection

Popular interpreters and compilers for the *dynamic languages* we have studied have Garbage Collection [Tea13c] [Lip03] [pyt13a, 2.1] [abo13b] [per13b, pp. 7-9] [MAso13, Chapter: Features.Garbage Collection]. This is implemented, so that the user of the language does not have to worry about removing objects from the memory. The deallocation of memory is therefore implicit in the sense that the languages do not expose any mechanism for forcing a value to be removed from memory. To fully support a *dynamic language* a compiler will have to implement a Garbage Collection of some sort.

2.5.6 Context

Some of the *Dynamic Languages*, like JavaScript and PHP, are tightly integrated into a special context. In PHP, the source code is allowed to contain regular HTML outside of the code blocks, that will be printed into the final output whenever it is encountered. If the HTML is in a loop or a function, it may actually be printed several times [MAso13, Basic Syntax - Escaping from HTML]. In JavaScript the context is typically provided by the browser and host web page (the global scope is actually the "window" object, which represents the host window in the browser). No output is possible in JavaScript without the browser or some similar context [ecm11, p. 2].

2.6 Related works

Trace-based Just-in-Time Type Specialization for Dynamic Languages Trace-based optimizations are based on the idea that just-in-time compilers do not have to compile code that supports all combinations of types, but rather just have to compile code to support the types observed [GES$^+$09]. The approach in this article is to start a JavaScript interpreter first (keep start-up time low) and compile "hot" functions (functions that are expensive, run-time wise) using the type information collected by the interpreter. This allows the system to have access to very fast native functions while at the same time maintaining full support of the JavaScript language.

The authors of this article later extended the work into the Mozilla Jäger-Monkey JavaScript trace-based compiler, used among others by the Firefox Browser [Man13].

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches In this 1991 paper Hölze, et al introduces the notion of Polymorphic Inline Caches [HCU91]. A polymorphic inline cache is essentially an optimized lookup cache. A lookup cache can be used in dynamic languages to cache the location of a methods that is called on an object. Since the object type is not known ahead-of-time in a dynamic language the object type and method name have to be resolved to a location at run-time. The lookup cache can speed this up, by storing the result of that computation.

An inline cache moves the location of the cache from a global position to the call site. In essence, the code that would look up the location of the method is replaced, at run-time, with code that calls the destination method directly. Since this might fail - the object might be of a different type next time the method is called - a guard is added to beginning of the method. If the guard finds that the type is incorrect, the lookup routine is called and the inline cache updated.

Since this might lead to a lot of updates back and forth if the cache misses a lot, the article proposes yet another improvement: the polymorphic cache. In essence this inserts a table of locations after the first cache-miss. This means that call site now calls a special stub method that contains checks for all cached types. If this cache missed, the table is extended and the program continues as before.

The result is that rather than having one global cache that contains all combinations of object types and method, each call site has a local cache that only contains the cache for the object types actually used at that call site. This keeps the caches as small as possible while still maintaining the benefit of the caches. The authors find significant improvements when implementing the Polymorphic Inline Caches in a Smalltalk context.

In relation to JavaScript Polymorphic Inline Caches have been implemented in the Google V8 JavaScript just-in-time compiler.

Google V8 JavaScript Just-In-Time compiler The V8 is a production quality, widely used implementation of a JavaScript Just-In-Time compiler, that uses the above concepts (along with several of its own improvements) [tea13b]. The V8 engine is used in the NodeJs server side JavaScript context and in the Google Chrome browser [nod13].

Mozilla Rhino compiler and interpreter Rhino is a production quality, ahead of time compiler that can compile JavaScript to Java classes. The project also includes a Javascript interpreter. The Rhino engine is used in RingoJs server side JavaScript context, and in other Java projects [Tea13f] [Dev13b].

Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs This paper approaches the problem of compiling dynamic languages from a different angle: Rather than building a completely new compiler, they extend and existing compiler for statically typed languages to support dynamic types [IOC+12].

The Java VM Just-in-time compiler is extended to support python byte code. The paper finds that significant speed-ups can be achieved with their modified just-in-time compiler compared to the python standard interpreter (cpython). These speedups are only achieved after introducing a set of new optimizations though.

CHAPTER 3

Analysis of the problem

This chapter will describe the different choices we have to make when writing a compiler for JavaScript. It will suggest solutions and show examples that illustrate some of the challenges that have to be overcome when compiling the language to native code.

3.1 Choice of output language

When making a compiler, an important task is to choose the output language. Since we are exploring the options of compiling the program to native machine instructions, an obvious option is to choose machine instructions as the output language. This has the advantage that we will be able to control exactly what the instructions the machine has to do to execute the program. Another advantage is that the language has no notion of types, because these concepts are to high level for a processor to handle. This means that we will not have to work around a static type system to implement the dynamic features of the source language. It is however also a disadvantage, that we are close to the hardware, because it makes it more difficult to implement features like the objects of the language. An obvious weakness to this approach is also that we would have to know many details of the instructions of the machine and operating system we are compiling for.

Another approach would be to compile the program to an assembler language. An assembler language is a language that is very close to actual machine code but leaves out details such as what numbers represent the instructions, and a jump to a line number can be written as a jump to a label in the code. To produce the native machine code from the assembler code we can just use one of the many assemblers that have been made. The advantage of compiling to an assembler language is that it is so close to machine code that we would gain the same advantages, however this also means that it has most of the same disadvantages.

We could also use a higher level language such as C. The advantage here is that the high level features of C could make it easier to implement the high level features of JavaScript. The disadvantage is, of course, that we do not have as much control over what the produced native machine instructions will be. However even though C is a high level language, it is not as high level as languages such as Java or C#. Furthermore there are many good C compilers that can produce efficient machine code. Another advantage is that we have experience with C programming from a course on embedded systems, so we could use time on making a good implementation instead of using the time learning a new language.

We could also choose to use a programing language on an even higher level such as the D programming language that can also be compiled to native code. The D programming language can, like C, be compiled to native code. An advantage over C is that it has direct support for many of the features such as nested functions and garbage collection. This is however also a disadvantage since we perhaps would not find out how these features are compiled to a native setting because we would just be using the features of that language. Another disadvantage is that we have no prior experience with the language.

We chose to use C as target language because we wanted to use a language that we were familiar with, and where it was relatively easy to implement the high level features such as objects.

3.2 Representing nested functions

In JavaScript, functions are objects, and can be declared as an expression anywhere in the code, where an object can be used. This has the consequence that functions can be nested, that is, a function can be declared inside another function declaration. It also means that functions can take functions as arguments and return functions as values.

For instance

```
function twice(f){
    function out(){
        f();
        f();
    }

    return out;
}
```

This means that we need to find a way to represent functions as values in C. The challenge is also to flatten the structure. This cannot simply be done as:

```
function twice(f){
    return out;
}
function out(){
    f();
    f();
}
```

because then we have the problem that the f in out has no connection to the f provided to twice as an argument. In other words, the challenge is to provide the function objects with the environments, that map variable identifiers to values in the memory.

This problem can also be illustated by the following example:

```
function makeMultiplier(c){
    return function multiplier(x){return x*c;};
}

var preserver = makeMultiplier(1);
var doubler = makeMultiplier(2);
```

In this example, when preserver is called, the c in multiplier should be bound to 1, but when doubler is called, the c in multiplier should be bound to 2.

Therefore preserver(10) will give 10, while doubler(10) will give 20. While the body of both functions are the same, their environments, that is, mapping of the variables to location in memory, are different.

3.3 Representing values and variables

Since JavaScript is dynammically typed, a variable can contain a value of any type. This means that a variable during the execution of the program can contain values of different types. Therefore, a variable cannot be said to have a distinct type.

```
var y = true;

for(var x = 0; x < 10; x++){
   if(x % 2){
       y = 42;
    }
    else{
        y = "blah";
    }
}
```

In the above code we see that the value of y will be true, 42 and "blah", respectively, during the execution, corresponding to the types boolean, numeric and string. This is a challenge when compiling JavaScript to C, because C restricts the variables to only contain values of one kind. Furthermore, in the JavaScript code

```
var x,y;
x = 1; y = 2;
console.log(x + y);
x = "a";
console.log(x + y);
```

the program will first write "3" on the console, and then "a2". This shows that the result from the addition depends on the types of the values of x and y. Depending on the types, the addition operator will either do addition or concatenation. Therefore it is also necessary to know the types of the values at run-time.

3.4 Representing objects

A JavaScript object is a storage unit for a set of properties that each map from a string to a JavaScript value. Exactly which properties an object contains can change over the duration of a program as well as the value type contained by each property.

```
var a = {};
//a has no properties
a.b=true;
//a has property b
if(x){
    a.cat="cat";
    //a has property b and cat
} else{
    a.dog="dog";
    //a has property b and dog which is a string
    a.dog=true;
    //a has property b and dog which is a boolean
}
```

Some properties further have a special meaning: If an object is created from a function with the "new" operator, it is the "prototype" property of the function that will be the prototype of the created object.

All property names in JavaScript are strings (even array indices): this means that `a[0]`, `a[0.0]` and `a["0"]` all access the same property, namely the property with the name "0". So, in order to represent a JavaScript object in memory, some sort of map from strings to JavaScript values is needed.

One obvious way to do this is to simply model each JavaScript object as a hash-map from strings to JavaScript values. This is simple to model, but has the potential disadvantage that two objects with the same fields contain the mapping twice.

Another way to model this is to make the mapping global and to represent each object instance with two pieces of information: 1) A link to the correct global mapping for the current properties 2) An array / list of JavaScript values that contain the actual values at the indices specified in the global map.

For this to work properly, an effective way to determine the correct global map is needed: Every time a new property is inserted or deleted, the object must point to a new global hash-map.

Combinations of these two extremes are also possible: A solution could allow each object to have a small number of properties in a map of its own, but when too many changes are made, a new global map is created.

Also, if one models objects in this way, special attention must be given to the representation of arrays, since these are, in principle, just objects with a special implementation of the "length" field. In JavaScript, they are often used as list structures, however, so it would probably not be beneficial to have a global map for each array currently in the program - at least it would be necessary to remove unused global maps to avoid filling the memory with global maps.

3.5 Meta programming issues

In addition to the issues outlined above with JavaScript values and objects that can change during the execution of the program, JavaScript has a number of meta-programming features that need to be addressed by a compiler:

- Functions can be redefined at run time

```
var f = function(x){return x+1;};
console.log(f(1)); //writes 2 on console
f= function(x){return x+2;};
console.log(f(1)); //writes 3 on console
```

- Functions can be used as constructors and as normal functions interchangeably: the value of the "this" object will change to reflect whether a new object is being created or not
- Every property (even of most native objects) can be replaced at run time

```
console.log("output") //writes output in the console
console.log = function(){};
console.log("output") //does not write in the console
```

- Inheritance: Prototype can be replaced: two object instances created with the same constructor function might not inherit the same properties via the prototype chain

```
var car = {wheels: 4, type: "car"};
var Suzuki = function(){this.brand="Suzuki"};
Suzuki.prototype = car;
var myCar = new Suzuki(); //the proto of myCar is car
var motorCycle =  {wheels: 2, type: "motorCycle"};
Suzuki.prototype = motorCycle;
var myMotorCycle = new Suzuki();
//The proto of myMotorCycle is motorCycle,
//though it was made from the Suzuki
//constructor. myCar was also made from Suzuki,
//but has car as proto.
```

These issues restrict the ways that JavaScript values can be modelled: Functions need to be modelled as pointers, the "this" variable needs to be looked up at run-time, most native objects will have to be made accessible as any other JavaScript object in the program and properties need to be looked up in the prototype chain at run-time.

3.6 Garbage Collection

In JavaScript, memory is implicitly managed: The language does not contain any constructs for explicitly allocating or de-allocating memory. The language contains constructs for creating new variables and objects, that will need to be stored in memory, but how this is done is not specified. In fact, the specification does not mention memory management or garbage collection at all [ecm11].

Since the specification does not mention this at all, a confirming implementation might simply ignore the issue and de-allocate only when the program ends. Since most JavaScript programs are executed either in the browser (running as long as the web page is open) or on the server (as a long running task) an implementation must provide some sort of automatic memory management to be useful.

Garbage Collection is the most common solution to the implicit memory management issue: memory is explicitly allocated to hold the variables needed and at regular intervals the garbage collector determines if any memory is no longer in use and releases it. Garbage Collectors differ in the way that they determine if memory is unused, but they all solve problem of automatic memory management [Tea13c].

3.7 Possible optimizations

Many of the optimizations available to compilers for statically typed languages are also available when compiling dynamic languages: Copy Propagation, Dead Code Elimination and Code Motion [ALSU86].

These optimizations provide a good trade-off between run-time in the compiler and the benefit obtained in the output code. There are many more optimizations described in the literature, but we will only consider these three "classic" optimizations to limit the scope of the project [ALSU86] [Cor13b].

When compiling dynamic languages, though, some additional optimizations might be beneficial: In particular calculating the type of different operands and expressions. This optimization is named "Type inference" for the action of statically analysing the JavaScript code to determine the type associated with each variable and expression. Since a variable can be used in such a way that it might hold different types at different times for the same expression, "any" is allowed as a type for an operand or expression to allow the optimization to be sound.

Chapter 4

Implementation work

This chapter will describe how we have implemented the different phases of the compiler. It will not go into detail with every aspect, but highlight the most interesting choices we had to make, and challenges we had to overcome.

4.1 Compiler structure

The overall compiler structure is a single-pass, multi-phase, broad compiler [GBJL00]. The project compiler completes one phase fully before handing control over to the next phase. This means that the compiler only needs a single pass to translate the program, but it also means that the compiler must keep all information for a phase in memory.

Rather than having a hierarchy of the phases, the Compiler class is the controller that calls each phase. Each phase is called with the data it needs (source file, AST, TAC tree, etc) and returns the output in the input format for the next phase. The lexer and parser, for instance, receives a handler object containing the source file and produces the Abstract Syntax Tree, while the optimizer both receives and produces a Three Address Code tree.

Some of the phases were build using external tools: Most of the lexer and parser was auto generated by a tool called ANTLR and the final build (from c code til native binary) is handled by the GCC compiler.

4.1.1 Phases included in the project compiler

The project compiler mostly follows the structure described in chapter 2. The exact structure is:

- Lexing and parsing the JavaScript to an AST
- Context handling on the AST
- Translation from AST to the TAC intermediate representation
- Optimization of the TAC.
- Translation from TAC to the IR intermediate representation
- Translation from IR to C-code
- Translation of the C-code to a native binary, by the GCC compiler.

Details of the choices we did when designing the different phases of the compiler will be described, as well as details in the implementation we have done.

4.1.2 Lexing and parsing

When we had to make the lexer and parser for the project, we considered handwriting them ourselves, using a tool to create them or to take a JavaScript lexer and parser from another project. The advantage of handwriting the lexer and parser would be that we would know the code better and we could implement the features we think are important, whether it be performance, good error messages or solving the problem of automatic semicolon insertion in an elegant way.

The advantage of using a tool to generate them is that we estimate that it takes much less time to make, because they are written in a syntax that is very close to the Backus-Naur Form grammar notation, that is also used in the ECMAScript Language Specification. There are many well tested tools for doing this, and many of them have build in error reporting.

It could also be a good idea to use the parser and lexer from another project, such as RingoJs which is also written in Java. The advantage is that if the code is easy to understand and use for another project it would require the minimal amount of work.

Since a compiler needs to implement the language specification correctly to be useful, we estimated that the best approach would be to make the lexer and parser with a tool directly from the Backus-Naur Form of the standard, so this is the approach we went with. We also thought that this was better than using the parser of RingoJs, because it would give us a good overview of the constructs of the language and not force us to use a pre-defined AST.

Because we had experience with the ANTLR lexer and parser generator from a course in computer science modelling, we chose to use that tool.

4.1.2.1 Building the AST

When we were done writing the grammar, and had made the lexer and parser with ANTLR, we had to write the Java code that would build the abstract syntax tree. The authors of ANTLR suggest the approach of letting ANTLR build a parsetree, and then make the abstract syntax tree builder as a Java class that extends the parse tree visitor, that the tool provides [Par12, 2.2].

The visitor pattern The visitor pattern is a design pattern that is used for adding behavior to tree traversals [GHJV95][pp. 330-344]. The pattern consist of an abstract class, that contains a method for each of the classes of nodes that the tree can have. So, if the tree can contain nodes of the type X, the visitor will have a method called visitX that takes an object of the type X as input. Each node type will have a method that takes the visitor as an argument and calls the correct method on the visitor.

Tree traversal can then be implemented by simply creating a concrete implementation of the visitor class and running the visitor on the root node. The idea is then that the function visitX does the computations for the nodes of the X class. If the computation depends on the subtrees that an X object can have, visitNode can be called recursively on these subtrees. visitNode will then call the correct visit function for the subtrees according to their type.

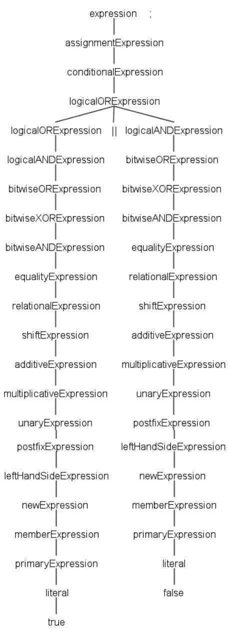
The abstract visitor class contains some default implementations for the methods. The default implementation for the visitX functions is that they should visit the child nodes recursively, and then aggregate the result, that is calculate a result from these values. The aggregation is made by the method aggregateResult which takes two results and give an aggregated result.

The defaultValue method gives the element that the aggregation should start from.

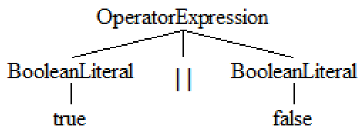
The default implementation of aggregateResult is to just give the right argument of aggregateResult, and the default implementation of defaultValue is to return null.

AST A parse-tree shows the syntactical structure of a program. The goal is to build the abstract syntax tree, which is very similar to the parse tree, but does not include syntactic details that are not important for the meaning of the program.

The parse tree of a simple JavaScript expression "`true || false`" is:



The reason that the parse tree is so deep is that the grammar takes into account the precedence of the different operators. The whole expression is first tried to be parsed as an assignment, because assignment should be done after all other operations, then as a conditional expression and lastly as a logical or expression, which is what it is. The same process is repeated for each of the literals. While it is practical that the parsing takes care of the operator precedence, the parse tree is obviously much too deep, and most of the nodes in the tree provide no semantical information. To make the tree more shallow, we have made the AST builder such that it, when it visits an Expression nodes, for instance a logicalANDExpression, checks if the node has one child or two children, and if it only has one child, the builder will simply return the result of visiting that child note. Therefore the AST for the above example is much shallower:



The illustration also shows that in the AST, we have a single class, Operator-Expression, that represents all types of binary operators, by containing a string that represents the operator.

4.1.3 Context handling

JavaScript has several context conditions, that should be respected for a JavaScript program to be correct.

4.1.3.1 Return statements

In JavaScript, all return statements must be inside functions [ecm11, p. 93]. Therefore the following JavaScript program is not correct:

```
function double(x){
    return x*2;
}
return;
```

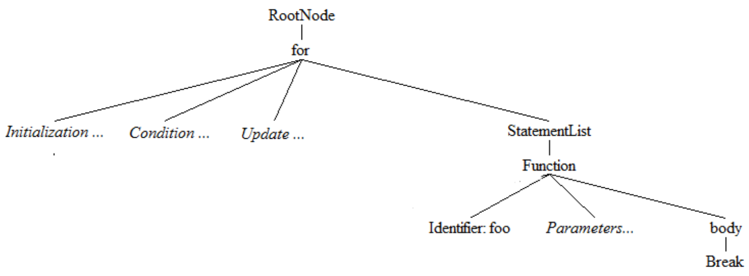
Checking this can be done as a tree-search for the return statement starting from the root-node. If a node is reached that represents a function, the search should not go in to the subtree starting at that node, because all return statements in that subtree are correct. The JavaScript program is correct with respect to the return statements, if the return-statement is not found by the search. We have implemented the tree search as a visitor.

4.1.3.2 Continues and breaks

The context conditions for continue and break statements are as follows [ecm11, pp. 92-93].

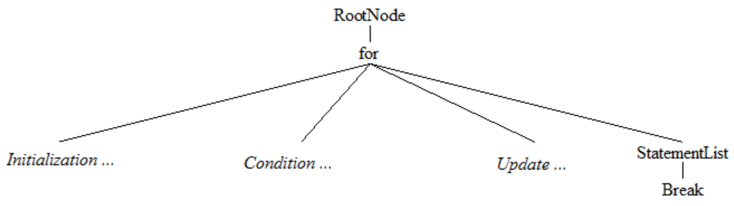
Unlabeled A continue or break statement should be inside a loop or switch, and there must not be a function scope between the statement and the loop or switch.

Figure 4.1



This means that the AST in figure 4.1 will be illegal because, while the break statement is in the subtree of a for node, there is a FunctionNode between the break and the loop. However, the AST in figure 4.2 will be legal.

Figure 4.2



Labeled For labeled continue or break statements, the rule is that they should be inside a loop or a block (code in a set of curly brackets), that has the same label as the continue or break statement, or the statement itself should have the label. It is still the case that there must not be a function scope between and the loop or block and the break/continue.

Algorithm Informally, the idea behind the algorithm for checking compliance with these conditions, is that each node asks all its subtrees if they comply. The node itself will comply if all the subtrees comply. For the subtrees to check for compliance, the node will tell them what labels are declared by and above the node, and if the subtrees are inside a loop or switch.

A switch or loop structure will therefore, in addition to the labels it know from its parent, tell ts subtrees that they are inside a switch or loop. If the structure has a label, it will also report that label to the subtrees. Functions cover all switches and loops, and will therefore tell their subtrees that they are not in a switch or loop and that there are no labels.

If an unlabeled continue or break is asked, it will answer yes, if it is told that it is inside a loop or switch, and no otherwise. If a labeled continue or break is asked, it will answer yes, if it is told the label was declared and no otherwise.

The algorithm is implemented as a visitor.

Alternative strategy Alternatively to this approach we could have ignored the condition, and just translated the code to C. In C, you will get a compile error when you try to make a jump to a label that is not declared, and therefore the condition would still be placed. The advantage of our approach is however that we do not rely on the error handling of the C compiler, and therefore make it more explicit how the conditions are derived from the ECMAScript Language specification. It also means that we can provide more meaningfull error messages than the C compiler, which, of course, does not know about JavaScript.

4.1.3.3 Getters and setters

Even though getters and setters are not supported by the compiler, we also check the context condition, that object literals, may only define each getter and setter once. This is implemented as a visitor.

4.1.3.4 Annotations

In the project compiler, the context handling is also used to annotate the Abstract Syntax Tree with information that will be practical to have in the later phases of the compilation.

4.1.3.5 Declarations and assignments

In JavaScript, there is a difference between function declarations and function expression.

```
function f(){
    console.log(g(5)); //g is bound to the function x*5, so 10 is written
    console.log(x); //x is bound to undefined, so undefined is written

    function g(x){
        return x*5;
    }

    var x = 5;
    console.log(x);//x is bound to 5, so 5 is written
}
f();
```

The code shows that function declarations bind the function value to the variable at the beginning of the outer function, while variable declarations bind them to undefined in the beginning, and then when the definition is reached, to the specified value.

To make the translation to C easy, we annotate the function nodes, with the sets of variables that are declared, and the sets of functions that are declared. We also annotate the root node with all the variables that were assigned to, but not declared, since they belong to the global scope as seen in the following example:

```
function f(){
    a = 5;
}
function g(){
    console.log(a);
}
if(x){
    f();
}
g();
```

If x is true, f will be run. That means f is run which has the consequence that 5 is assigned to a, in the global scope. When g is run it can then print a. On the other hand if x is false, a will not have any declaration or assignment, and g will produce a runtime error. This example shows that we also need to know what variables can belong to the global scope.

The annotating of this information is implemented as a visitor.

4.1.3.6 Alternative strategy

An alternative strategy to solving this problem with annotations, would be to make a translation from AST, to an intermediate language that was very similar, but had the restrictions that all variable and function declaration should be in the beginning of each body of a function.

We could then, for instance, take the AST representation equivalent of

```
function foo(){
    console.log(f(a));
    function f(x){
        return x*2;
    }
    var a = 5;
}
```

and turn it in to an intermediate representation that would be equivalent to

```
function foo(){
    var a = undefined;
    var f = function f(x){
        return x*2;
    }
    console.log(f(a));
    a = 5;
}
```

This solution is completely equivalent to what we do, but one could argue that it shows the meaning of what is done more explicitly, than adding annotations. Our solution, however, has the advantage that we do not need to build a new tree which would be more difficult than simply adding annotations, and we do not put a new tree in the memory of the computer.

4.1.3.7 Function IDs

Because there are not nested functions in C, all the function definitions have to be in the same outer scope. Therefore we have a problem if several functions in different scopes of JavaScript share the same name. To solve this problem, we give every function a unique ID, by visiting the tree and annotating all function nodes with a number.

4.1.4 Intermediate code generation

We have firstly made an intermediate representation that we call IR, which is very close to the syntax of the C programming language, albeit with some simplifications. We have chosen to have such a representation, because it makes it easy to make the C code that should be generated. When we want to translate from the AST to the IR, there are then two approaches:

Doing a direct translation or adding another (or more) intermediate language.

The direct translation can be done relatively easy, because many of the constructs of JavaScript are directly available in C, such as if-then-else, for-loops and while-loops. This also has the advantage that the outputted code will be very similar to the structure in the JavaScript code. Function calls can also be translated more or less directly once closures have been taken in to account. This has the problem, however, that the evaluation order of the arguments is defined to be from left to right in JavaScript, but in the C language the evaluation order of the arguments is not defined [Int11, p. 4] and is often right to left. If any of the arguments are the results of functions that have side-effects the compiled program might be incorrect due to this difference.

The operators in C, of course, do not work directly on the JavaScript values. This can be solved by implementing them as functions, but then they might also have the wrong evaluation order.

Another approach would be to make another intermediate language. Specifically we considered using an intermediate language, that we called TAC, because it used a three address code structure. The three address code structure is that the statements of the language are all on one of the three forms:

- $a_1 = a_2 \ op \ a_3$
- $a_1 = opa_2$

- $a_1 = a_2$

Here *op* is an operator, and the *a*s are addresses. The addresses that can be used as addresses in a TAC-statement represent variables, constants, properties and function calls.

This means that the equivalent to JavaScript `a=2+2;` is simply $a = 2 + 2$, while `a=2+4*5` can be translated as
$t_1 = 2$
$t_2 = 4 * 5$
$a = t_1 + t_2$
With this structure it is easy to make sure that the evaluation order is correct, and the order is also very clear from the representation. To be able to do loops, and conditionals, the TAC also has a jump statement, a conditional jump statement, and labels.

The TAC approach also has advantages over the direct translation when we want to do optimizations. Because there are so few types of statements, and they all are very simple, there are only few cases to consider when doing the optimizations. This makes it is easier to argue that a transformation that the optimizer will do does not change the produced result. The fact that the evaluation order is expressed explicitly in the representation also makes this much easier.

Furthermore, the classical dataflow algorithms uses control flow graphs to do their computations, and control flow graphs are easy to construct from three address code compared to IR.

During the development, we tried using both approaches. In the end we chose to use the TAC, however, because we wanted to get the evaluation order right, and because we wanted to do optimizations.

4.1.4.1 Our choice of TAC

We had to choose how close the TAC should be to the source and target languages. Since we already had a language, IR, that was close to the target language, we chose to make the TAC closer to the source language. We estimated that this would be a good idea, because it would mean that we could for instance infer JavaScript type information on variables, before translating the code to the IR.

Our TAC has statements of the following types:

- Assignments:
  - $a_1 = a_2$
  - $a_1 = op\, a_2$
  - $a_1 = a_1\ op\ a_2$
- Jumps:
  - $goto\, l_1$
  - $if\ a_1\ goto\, l_1$
  - $ifFalse\ a_1\ goto\, l_1$
- Labels:
  - l_1
- Returns:
  - $return\ a_1$

The meaning of the operators is the same as in JavaScript, that is, they do the necessary type coercion to produce the result. For the conditional jumps, the conversion of the condition address to a boolean is also left implicit. That is, $if\ a_1 goto\, l_1$ will only jump to the l_1 label if the value represented by a_1 can be coerced to $true$ and $ifFalse\ a_1\ goto\ l_1$ will only jump if it can be coersed to $false$. The coersion is done in the same way as in JavaScript.
The addresses that can be used in the statements are:

- Variables
- Temporary Values
- Property Addresses (e.g. $a_o[a_p]$)
- Constants
- Function calls ($a_f(a_1...a_n)$ for a call with n parameters)
- Constructor calls ($new\ a_f(a_1...a_n)$ for a call with n parameters)

Constants are used for two puporses: Constants, that represent the literals of JavaScript and Object literals / Function expressions. The function expressions and object literals are not really constants as they represent objects, but are modelled as such to simplify the translation. The function constant consists of its name, its parameters and its body as a list of TAC statements. Furthermore it contains all the annotations from the context handling.

The variables have the same meaning as in JavaScript, such as the semantics about being defined in closures, that is, they are available to be used in function declarations on the same or a deeper level.

The temporary values are used for storing the temporary results that are used when evaluating an expression. These values are therefore not part of the closures.

The PropertyAddresses are used for writing to and reading from properties. The semantics are the same as when reading from and writing to properties in Javascript. The conversion of the index to a string is also left implicit in the TAC code.

The values that the variables and temporary values represent have the same meaning as in JavaScript.

4.1.4.2 Translation to TAC

When a node from the AST is translated in the compiler, the translation is returned as an object of the type TacResult. The TacResult consists of a field, "statements", which consists of the statements that the node is translated to, and a field, "result", that contains the TAC address of the result that the statements compute.

We will not explain how every single statement and expression of JavaScript can be translated to TAC, but will show some of the most interesting transformations.

In the following, s_i will represent the statements of the TAC translation of the AST node i, and a_i will represent the address of the result.

Binary operation When we translate a binary expression such as $e_1 + e_2$, where e_1 and e_2 are expressions themselves, we do it as follows:

First we translate e_1 and e_2.

The TAC statements for entire operation are then:
s_1

s_2
$t = a_1 + a_2$

and the result is the address t, which is a fresh temporary value, that is a temporary value that has not yet been used in the translation of the program.

From this it follows clearly, that the left operand is evaluated before the right, because the statements of the left operand are added before the right hand side. This will also be the case when we have an expression with several operators, and perhaps parenthesises, because the left operand is always written first in a depth first fashion. Writing the nodes in this way is called a pre-order depth first traversal.

If then else-statement The if then else structure can be written relatively simply in the three address code. An if then else statement consists of a condition, an if-branch and an else-branch.

We translate the structure as follows:

First we translate the condition, c, the if-branch and $else$-branch.

The TAC statements for entire operation are then:
s_c
$ifFalse\ a_c\ goto\ l_{else}$
s_{if}
$goto\ l_{after}$
l_{else}
s_{else}
l_{after} , where the labels are fresh. For the result address we give null, because JavaScript statements do not give values back.

Short circuited logic The ECMAScript Language Specification specifies [ecm11][11.11] that when the the result of the logical or operator is evaluated, the $right$ operand should only be evaluated if the $left$ operand evaluates to false. The effect can be illustrated by:

```
function  getTrue(){console.log("true");  return true ;}
function getFalse(){console.log("false"); return false;}
getFalse() || getTrue();
```

Where the result should be that "false" and "true" should be written in the console, whereas for

```
getTrue() || getFalse();
```

would only write "true" in the console. The semantics for the logical and function and for the `:?` operator are similar.

In the TAC this is easy to translate, because the evaluation order can be controlled. It is therefore translated as follows:

```
$s_{left}$
$t_{result} = a_{left}$
$if\ t_{result}\ goto\ l_{shortcircuit}$
$s_{right}$
$t_{result} = a_{right}$
$l_{shortcircuit}$
```

where t_{result} is the result address.

Constants and functions The literals of the JavaScript language can be translated to the TAC constants described above. For the object literals, we also add property assingments to add the properties.

Function call We have added an address, FunctionCall, for doing function calls. The address contains the address of the Function that is called, and the addresses of the parameters. When we translate a JavaScript function call, we first write out the translations of the parameters, in order from left to right. Afterwards, we make the FunctionCall address, and sets its parameters to the result addresses from the translated Javascript parameters. Lastly we assign the FunctionCall address to a fresh temporary variable. The variable is returned as the result address of the functionCall.

```
$s_1$
...
$s_n$
$t = s_f(a_1 ... a_n)$
```

The reason that we do not simply return the function call as the result address is that then a JavaScript expression like $f(g(), h())$ would be translated as $f(g(), h())$ in the TAC, and then the evaluation order is not explicit. The reason is also that the call of the function has to be represented by a statement in the TAC if it is called as a statement in JavaScript. This is often done with functions that do not return a value, but change the state of the program. In TAC this can only be done as an assignment because that is the only type of statement, that does not make a jump.

4.1.5 Translating TAC to IR

The greatest challenges when translating TAC to IR, were implementing values, environments and operators. These challenges will be discussed further in later sections. Other than those difficulties, most of the statements in the three address code can be translated more or less directly to C, such as assignments to the temporary values, labels and jumps.

4.1.6 Output code generation

When doing the outputting of the IR as C code we did not have any challenges, because the IR was so close close to C code that it could almost be considered a syntax tree for the language.

4.1.7 Build native executable

The last phase of the compiler builds the native executable. This is done by letting the GCC compiler do the compilation from C-code to native code. To make our program work on both Unix and Windows systems, we use Cygwin to compile the executable on windows.

4.1.8 Environments and values

As we saw in the analysis, one of the reasons that JavaScript variables cannot be translated directly to C variables is that they can contain values of any type. A practical way to solve this would be to make a C-type that can represent all the JavaScript values. We also saw in the analysis that it is necessary to associate a type with the value. We have solved this by making the C-type of values as a "struct" that has a field containing the type, and a field content containing the value itself. The type of the content field is a "union" type of all the different

JavaScript types (booleans, numerics, undefined and so on).

The "union" type reserves enough space in memory to hold any of the enclosed types, but it can only hold one of the types at any given time. This type allows us to easily chance the type of a variable at run-time without having to reserve memory for every possible data type.

Content The primitive types of JavaScript, are boolean, numeric, string, null, undefined. In JavaScript, the numbers are defined as IEEE doubles [ecm11, p. 29], and this is exactly what the doubles of C are defined as [Int11, p. 507]. The boolean values are true and false, and they have equivalents in the stdbool.h library. A typical representation of strings in C is an array of chars. This is a little problematic because JavaScript strings can contain unicode characters. We chose this representation anyway because it was simple. In the limitation section we will consider solutions to this problem. The types null and undefined only contain one value each, namely null and undefined. Therefore we do not need to worry about having a representation of these values, because they are already represented by the type field of the value struct.

Objects We chose to implement Objects as hashmaps. This implementation is good because it is general, and simpler to implement than the other possibilities presented in the analysis. We chose to use a library implementation of HashMaps instead of doing one ourselves, so that we could use our time, for the other challenges of making a compiler. The specific implementation we chose was the UTHash, which had the advantage of being easy to implement in our program, and which is used by many other projects [Han13].

The struct that implements the Objects, therefore contains a HashMap. Since functions are also objects, the struct can also possibly contain reference to a closure. The closure is a structure consisting of a field containing a pointer to a C-function that represents its function body, and a field containing a pointer to a data structure that represents its environment.

Prototypes The object struct also contains a field, proto, representing the hidden prototype property, that all objects have. When an object is created using "new", we simply set the new objects proto to the value of the contructors "prototype" field (which the program get from the constructors hashmap).
To get the prototype chains to work, the lookup for properties should look in the object first. If the key is not found, it should do the lookup in the proto of the object, an so on. If nothing is found, undefined is returned.

4.1.8.1 Representing the environment itself

In the analysis we found out that we have to make an implementation of an environment, that is, a mapping from variables to the location of their values in the memory.

We have considered representing the maps by hashmaps, treemaps or arrays. Hashmaps and treemaps have the advantage that they can map the identifier string directly to the location. If an array is used, we first need to represent the identifiers as integers in the range from 0 to the number of identifiers. Arrays and Hashmaps have the advantage, that lookups and change of values can be done in constant time [Har08, 69] and amortizised constant time respectively [Har08, 88]. Copying them however takes linear time in case of the arrays and at least as much for the hashmaps. Immutable implementations of treemaps on the other hand can be "copied" in constant time, but looking up or changing values takes logarithmic time [Har08, 88]. The array data type also has the advantage of being built in to C, so it is fast to implement.

We estimated that variables will be accessed more often than environments will be created, so we chose the array representation, because this representation gives fast and easy access to the memory locations.

```
var i = "Irma";
function first(){
    var j = "Johanne";
    function second(){
        var k = "Kamilla";
    }
}
```

From the code we see that the outer scope is $\{i\}$ [1], the scope of first is $\{i, j\}$, the scope of second is $\{i, j, k\}$. The environment of the outer scope is therefore $[i \mapsto location\_of\_i]$, the environment of first is $[i \mapsto location\_of\_i, j \mapsto location\_of\_j]$ and the environment of second is $[i \mapsto location\_of\_i, j \mapsto location\_of\_j, k \mapsto location\_of\_k]$. We see that each environment is an extension of the environment outside it, and that we know the size and keys (the identifiers) of the map that represents the environment. We can therefore represent the environments as fixed size arrays, and then represent the identifiers by the index in the environment at which the memory location of their value is placed.

[1]For simplicity of the explanation we will ignore the function variables, that is, the scope is actually $\{i, first\}$. In the actual implementation, this is of course, done correctly.

Implementaion details

```
function first(){
    var j = "Johanne";
    function second(){
        //1
        var k = "Kamilla";
        var j = "Josefine";
        //2
    }
    //3
}
```

Here we see that the j in first will point to a location contain the string "Johanne". Second, however, declares its own j, which points to the location "Josefine". In the environment of second j should therefore point to another location than the environment of first. A solution to this could be to simply rename the inner j to j2. We have however solved the problem in the following way. When at runtime, the environment of a function is made, it is first made as a copy of the environment in which the function was created. Afterwards, all the declared variables are made to point new unique locations of undefined values. This way, these variables will point to new locations in the inner scope, but will not change in the outer scope. The variables that are not declared in the inner scope, will point to the same locations in both scopes. In our example at 1, k and j will therefore be undefined-value and at 2 they will be "Kamilla" and "Josefine" respectively. At 3, j will still be "Johanne", because j was changed in the environment-array of second, but not in the environment-array of first. The solution is good because it makes environments smaller, and hides locations that are out of the scope.

Furthermore, the example from the analysis:

```
function makeMultiplier(c){
    return function multiplier(x){return x*c;};
}

var preserver = makeMultiplier(1);
var doubler = makeMultiplier(2);
```

shows that each time a function is called, a new environment that puts the values on new locations must be created, such that the variables can be bound to different values in different contexts. This is also the way we have implemented our environments.

Reading and writing to the variables We can now read from and write to the variables in the closures by looking their location up in the environments.

4.1.9 Operators

In JavaScript one of the effects of the lack of explicit type information is that the operator specifications become quite complex. The "==" (non-strict equality operator) uses what is known as the "Abstract Equality Comparison Algorithm" [ecm11][11.9.3] that defines the equality operation for the related operators ("==", "!="). This algorithm is recursively defined with each recursive call "lowering" the type of one of the operands using either "ToNumber" or "ToPrimitive".

The effect of this is, that in the absence of any type information, any case described in the algorithm must be assumed to occur and therefore the code to handle it must be present.

If every operator call was inlined this could lead to a very, very large output file. Instead, the project compiler uses a solution where all operators are implemented in a run-time support library. So the above "==" operator translates in to c code similar to:

```
Value t_3 = equality_comparison_op(t_1, t_2);
```

This solution helps keep the output code file down, but comes at the price of a larger run-time overhead.

Logical operators There is a notable exception to the operators being defined as methods in the run-time support code. The logical operators are handled in the conversion to the TAC form. This is due to the short-circuit requirements of these operators: that is, these operators must be lazily evaluated [ecm11][11.11].

Type information As long as nothing is known about the operands, the general version must be included. If we are able to reason about the operands it is possible to reduce the complexity of the operands - sometimes to the point where they can be replaced with the standard c operator.

Example:

```
var c = a + b;
```

If nothing is known of a and b, the "addition_op" run-time support method has to be used to determine if any operand needs to be coerced and if the output is a string or a numeric value.

If it is known that, say, a is a string and still nothing is known of b, the operator can still be simplified. In this case the operator will always be a string concatenation and the unknown variable will always be coerced to a string. The operator can therefore be calculated with the following steps:

1. Convert b to a string using "To_String"
2. Concatenate the strings of a and b
3. Store the result in a JavaScript value at the location of c

This looks more complex, but is simpler, because the type information of the variables does not have be to queried before choosing the correct action. This also saves the overhead of the run-time call.

If it is known that a and b are both numeric, the operator can just be modelled as the c "+" operator - though the operands still have to be unboxed, and the result has to be boxed before storage in the implementation we have made.

In principle it is possible to have completely unboxed values, if it is known that they never change types. The current project compiler infrastructure does not support this though.

4.1.10 Garbage Collector

While it is possible to have an implementation of the full JavaScript specification without including a Garbage Collector, this solution would be very limited in the real world due to the long running nature of many applications of JavaScript (in the browser and on the server side). For the purpose of the project compiler a Garbage Collection system was needed to ensure a higher degree of fairness in the comparison to the contemporary implementations, that all have Garbage Collection systems.

At a high level, there are two solutions possible for including a Garbage Collection system in the project compiler:

- Write a dedicated Garbage Collection system for the implemented JavaScript memory model.
- Use and existing Garbage Collection system for the C language.

There are many ways to actually implement a Garbage Collection system: reference counting is one of the simpler ways. It has the problem, however, that if two objects reference each other, but are otherwise unreachable, it will not free their memory. Many common JavaScript patterns will create such cycles of memory. Therefore, Mark and Sweep-algorithms are the de-facto standard among browser implementations, since they do not have this weakness [Tea13c]. However, Garbage Collection systems are research subjects on their own and it would be outside the scope of the project to implement a dedicated Garbage Collection system.

Fully functional Garbage Collection systems for C are somewhat rare. One of the better known systems is the Boehms-Demers-Weiser conservative Garbage Collector [Boe13]. This GC is used, among other places, for the Mono project [Pro13].

The use of a ready Garbage Collection systems makes it easy to enable Garbage Collection in the project compiler, but limits the options for tailoring the Garbage Collection system to the project compiler's JavaScript memory model.

4.2 Implementation of optimizations

The project compiler implements the three "classic" optimizations described in the analysis ("Constant propagation", "Dead code elimination" and "Function inlining") along with the "Type inference" optimization specific to the dynamic language.

The optimizations implemented in the project compiler all use a data structure known as a control flow graph. The control flow graph is a directed graph, with nodes representing basic blocks and edges representing change of control from one basic block to another (a jump or simply from one line to another if the basic blocks are next to each other in the code) [ALSU86][9.2].

A basic block is a list of three-address code statements that contains no jump labels (with the possible exception of the first statement) and no jump statements (with the possible exception of the last statement). This effectively means that there is no way for program execution to start or end in the middle of a basic block.

As an example of the translation to three address code, consider the following fragment:

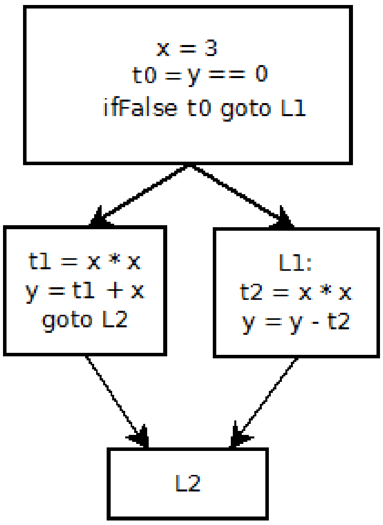
```
var x = 3;

if(y == 0){
    y = x * x + x;
}
else{
    y -= x * x;
}
```

This could be translated in to the following TAC statements:

```
x = 3
t0 = y == 0
ifFalse t0 goto L1
    t1 = x * x
    y = t1 + x
    goto L2
L1:
    t2 = x * x
    y = y - t2
L2:
```

These statements contain four basic blocks. The control flow for this fragment is the following:



The optimizations that are implemented in the project compiler are all global optimizations in the sense that they work on an entire control flow graph for a single function.

Program points The global optimizations considered share an overall algorithm known as a data-flow algorithm [ALSU86][9.2]. Members of this family of algorithms share the property that they examine one statement at a time in relation to two positions for that statement: "program point before" and "program point after".

The "program point before" is used to reason about the state of the variables just before the statement is executed and the "program point after" is used to reason about the state of the variables just after the statement is executed.

Let `I(s, x, in)` denote the information about the variable x, at the program point before the statement s, and let `I(s,x,out)` denote the information at the program point after s.

In addition to these positions, a set of predecessors and successors are maintained for each position. If the statement is in the middle of a basic block it will have only one predecessor and only one successor, but statements at the beginning or end of basic blocks can have more. For instance, in the illustration we see that statement $y = t1 + x$ has only one successor and predecessor, while $ifFalse\ t0\ goto\ L1$ has two successors.

The data flow algorithms work by updating the information at these positions iteratively until no more changes can be performed.

After the algorithm has finished, the compiler can use the information do the actual optimization of improving the program.

Constant propagation As a concrete example of a data-flow algorithm, the Constant Propagation optimization is considered.

In constant propagation we want to replace the variable "x" with the constant "k" in a statement in the program, if we can prove that x holds the constant value "k" for all paths in the program that leads to the statement.

Consider constant propagation for one variable. At any position in the control flow graph one of the following three information values is assigned to it:

- Bottom: meaning that the algorithm has not yet inferred the value for the variable.
- c: meaning a specific constant, c
- Top: meaning, the variable can hold more than one value at this point in the program

At the beginning of the algorithm, the value "Top" is assigned to the program point before the entry and the value "Bottom" is assigned to all other program points.

For Constant Propagation, the information is then updated according to the following rules [ALSU86][9.3] [Aik13b] [Aik13a] .

Rule 1: $I(s, x, in) = lub\{I(p, x, out) \mid \text{p is a predecessor}\}$

Where *lub* is the least upper bound, defined in the following way (where "c_k" is a specific constant):

$$lub(Bottom, c_1) = c_1$$
$$lub(c_1, c_1) = c_1$$
$$lub(c_1, c_2) = Top$$

$$lub(Top, ...) = Top$$

This means, that if we infer from the rule that $I(s, x, in) = c$, it is because on all paths going in to s, x is either "c" or we do not yet know what x can hold - that is, x is bottom. On the other hand, if we infer that $I(s, x, in) = Top$, it is because there is are two or more paths leading in to s, where x can hold different values, or because there is a path where x can hold multiple values - that is a path where x is Top.

Rule 2 $I(s, x, out) = Bottom$, if $I(s, x, in) = Bottom$

Rule 2 shows that if we have not yet inferred what x can hold at the program point before s, we should not yet try to infer what it can hold at the program point after.

Rule 3 $I(x := k, x, out) = k$, if k is a constant

If x is assigned to a contant we know that at the program point after the assignment x holds that constant as value. Note however, that rule 2 has higher priority, so this is ignored if $I(x := k, x, in) = Bottom$.

Rule 4 $I(x := f(...), x, out) = Top$ where f(...) is some expression such as an operation or a property lookup.

This means that if the right hand side is not a constant, then we assume that we do not know the value of x after this statement.

For JavaScript, this rule is interpreted a little bit differently if the right hand side is a function call. Since a function call can potentially change any variable in this scope, we, in this case, instead use the rule:

For all variables y: $I(x := f(...), y, out) = Top$ where f(...) is a function call or a constructor call.

Rule 5 $I(y := ..., x, out) = I(y := ..., x, in)$

This means that updates to another variable do not affect our information about x. The only exception is rule 4 which has higher priority.

Algorithm termination The algorithm continues until no updates can be performed according to the rules.

There will be a last update because there is a limit to how many times each rule can update the information of a statement:

- Rule 1: This rule can update the value at the program point before at most 2 times. One time assigning a constant, and one time assigning "Top".
- Rule 2: This rule never updates any values - it ensures that Rule 3 and 4 do not assign values prematurely
- Rule 3: Will only update the program point after the statement, after being applied at most once
- Rule 4: Will only update the program point after the statement, after being applied at most once
- Rule 5: Will only update the program point after the statement, after being applied at most once

Since there is no rule that can update the information indefinitely, the algorithm will eventually terminate.

Update statements Once the algorithm terminates, x is replaced with c in any statement in the code, where x was found to be that constant. When running the algorithm, it might be necessary to run the whole process several times. Consider the following code fragment:

```
var x = 12;
var y = x;
var z = y;
```

After the first run of the algorithm, we could get the following:

```
var x = 12;
var y = 12;
var z = y;
```

And after the second run:

```
var x = 12;
var y = 12;
var z = 12;
```

In the project compiler this means that an optimization will be run until it can no longer perform any updates on the statements. The project compiler runs the optimizations in a specific order: Constant Propagation, Inlining, Dead Code Elimination and Type Inference. This is viable as no updates that a later optimization performs would allow new updates for an earlier. The only exception is Inlining, but due to the limitations of the current implementation this is not actually an issue. If the project compiler included more optimizations (Constant Folding for instance) it might be beneficial to alternate between optimizations.

Type inference Type inference can be performed using the same algorithm, with the same set of rules, although this time the information values that can be assigned to the program points are:

- Bottom: meaning that the algorithm has not yet inferred the type for the variable
- type: where type is one of:
  - undefined,
  - numeric
  - string
  - boolean
  - null
  - object
- Top: meaning that the variable can hold different types depending on the flow of control.

The least upper bound is calculated as before, with "type" replacing the constant.

The actual type values are infered by a modification of rule 4, where assignments to constant values and from operators produce known output types. For instance:

```
var x = a - b;
```

Even if the types of a and b are not known, the type of x is always numeric due to the way that the subtraction operator is specified. Many of the JavaScript operators have a limited type space, but not all.

The type information is used in the translation from TAC to IR to make it possible to implement the operations directly with C operators.

Dead code elimination: This optimization removes unused code - defined as assignments to variables, where the assigned value is never read from the variable.

Works in much the same way as the other optmizations, but this time reads to a variable are measured. The possible information values are then:

- Bottom: meaning that the algorithm has not determined if the variable is dead or alive
- alive
  - true: meaning the variable is alive
  - false: meaning the variable is not alive, or in other words, that it is dead
- Top: meaning that the variable can be dead or alive depending on the flow of control.

The data in this data-flow algorithm flows "backwards" as seen in the following rules:

Rule 1:

$$I(s, x, out) = lub\{I(p, x, in) | p \text{ is a successor}\}$$

This rule works like the first rule of the other optimizations except that the data flows "backwards" (from the program points before the successors to the program point after for the current statement). The least upper bound is defined as before.

If we infer that x is alive, it is because it is alive or bottom on all successors. Likewise if it is dead. If we infer that x is top, it is because two successors say that it is dead and alive respectively, or because a successor says that it is top.

Rule 2:

$$I(.. := f(x), x, in) = true$$

This rule states that if x is used at all on the right hand side, then it is alive.

Rule 3: $I(x := e, x, in) = false$, if e does not refer to x

This rule states that x is dead before the statement where we assign to it, or more informally that the value of x is unimportant just before we assign x to something. This ensures that two successive writes without an intermediary read will mark the first write as dead.

Rule 4: $I(s, x, in) = I(s, x, out)$, if s does not refer to x at all

This rule simply states that if x is not used, the liveliness of x is copied.

After running this algorithm, any statement found to have alive = false, can safely be removed, with the following caveats for JavaScript:

- The scope of the variables needs to be taken in to account: Even if a write is dead in the local scope it might be alive in the a broader scope. We therefore model the last write (before a return statement) to every variable as being alive, to ensure that we do not accidentally remove a statement that was alive.
- Properties can be accessed in many ways: It is very difficult to prove that a property is not read at a later stage, since an object be stored in multiple variables - the safe approach is to leave all property writes untouched.

Inlining: The function inlining runs after the Constant Propagation has finished. It uses the information from that run to find any function call statements that calls a constant function node (meaning that the function is not altered between calls to it). If it finds such a node, it checks if the function node is a candidate for inlining. The project compiler is currently very restrictive in the rules for inlining:

- The function must not use the variable named arguments, which has a special meaning in JavaScript.

- All variables used by the function must be declared in its own scope. This restriction also applies to all inner functions.
- The function itself must not call any functions.

In practise this is so restrictive that the inlining only works for specifically constructed test examples, but it is included to as the basis for more elaborate inlining functions.

There is a further restriction in the way that the calculations are performed: To decorate a function as constant, it must not have any chance to be altered before it is called. Since Constant Propagations currently assumes that a function call changes all variables in scope, this has the undesirable effect that the project compiler can never inline a function that is called inside a loop, but defined outside it, since the call to the function itself marks it as being modified.

This could be changed by allowing the Constant Propagation to examine the function before deciding if it alters variables or not.

Algorithm implementation The order in which the statements are visited will influence the running time of the algorithm - however, the most efficient order depends on the whether the data flows "forward" or "backwards" [Win11]. For the project compiler, a naive solution was implemented that simply iterates over all statements to find the next statement where data can be updated. As with the rest of the project compiler implementation the decision was to use a simple solution and not to improve the implementation until a bottleneck was revealed.

The different optimizations in the project compiler that uses the data-flow algorithm all share the same implementation of the basic algorithm since they only differ in the rules applied to the different positions. The optimaztions provide these rules to the data-flow algorithm runner using a shared interface.

4.3 Limitations of our implementation

The project compiler does not implement the full JavaScript specification. The following is a detailed list of the features not included in the project compiler along with an outline of how they could have been included. The items in this list were excluded to limit the scope of the project compiler in a way that would allow a high level of comparability to the existing, fully compliant JavaScript implementations.

4.3.1 Lex and Parse level

JavaScripts "Automatic semi-colon insertion" feature is not supported in the project compiler. This feature of the language inserts tokens for ";" that are not found in the source file in certain circumstances when this would make the program parse. The exact rules for when to insert the token are given in the specification [ecm11][Section 7.9]. These rules are specified in terms of parse errors encountered that could have been avoided had there been an extra semi-colon.

This hints at an obvious solution: hook in to the error handling of the lexer - check if the error was due to a missing semi-colon. If it is, backtrack, insert it in the token stream and re-run the parser from the last known good position.

Another solution is to extend the grammar to include rules both with and without the semi-colon for all the places that could legally exclude a needed semi-colon. This means that we need to make a phase of the compiler where we visit the parsetree, to determine what the meaning of the nodes where a semicolon could be are. This could be done before, or as part of the AST generation. Also, this solution greatly increases the size of the grammar.

4.3.1.1 Regular expression literals

The JavaScript grammar actually contains two, separate grammars: One for where leading division operators are permitted ("/" and "/=") and one for where they are not. The reason for this separation is to allow the correct parsing of the following two statements: `var a = foo( b / 2, c /i);` `var a = /ab/i;` The first statement simply performs a division on both arguments for the function foo on the right hand side, whereas the next statement is a regular expression on the right hand side.

If only one grammar is used, one of the two statements will not be parsed correctly.

For the project compiler only one grammar is used, since Regular Expressions are not supported.

4.3.2 IR generation level

For the generation of intermediate representation there are some language constructs that have been excluded from the project compiler to limit the scope of the project. In addition, the compiler does not implement the entire native library specified in the standard. Specifically the following constructs and libraries are not supported:

4.3.2.1 Switch statements

The switch statement appears to be easy to model, when translating to c, simply by using the c "switch" statement. There is an added challenge, though, in the fact that in JavaScript you can switch on any type of variable and case any expression, like for instance:

```
switch(4){
    case 2 + 2: console.log("true"); break;
    case false: console.log("false");
}
```

will print "true".

The switch statement in C only supports integer values as the expression to switch on (that includes a single char, but not strings for instance) [Cor13a].

An implementation strategy in TAC could be the following:

- Associate each case block with a label
- Determine the destination label by evaluating the expressions associated with each case block one at a time. The first time there is a match, perform a jump.
- Break statements jump to the end of the switch statement - otherwise we just fall through

4.3.2.2 With statements

The JavaScript "with" statement changes way that variables are looked up by creating an object environment with the given object expression for all statements wrapped by the "with" block. This allows code, like the following:

```
var a = "bar";
with(console){
    log("foo");
    log(a);
}
```

which prints "foo" and then "bar" (if we assume that console is not modified).

The challenge with this statement is, that it is not actually possible to know what "log(a)" will produce - assuming that console contains a function named "log" and assuming that it doesn't contain a variable named "a" it will produce the above output, but due to the dynamic nature of JavaScript, this might change during run-time. This means that, at compile-time, it is not possible to know if an identifier inside the with statement is a property lookup or a variable lookup.

An implementation strategy might be the following:
For every variable lookup inside the "with" statement - try to look up in the object property list first - otherwise look up as normal variables

For the TAC, a solution could be to have a new type of address, that represents such a lookup. In C code, a solution might look like the following for the identifier "val_name":

```
Value val;
if(has_property(obj, "val_name")){
       val = get_property(obj, "val_name");
}
else{
       val = value_at(jsvar_val_name);
}
```

Where `get_property` looks up the provided property in the object, and where `value_at` looks up the value of the JavaScript variable jsvar_val_name in the environment.

The performance draw backs of the with statements are obvious, but since it is not possible to determine if the identifier is a variable or property, there is no general way around the extra lookup for all variables used inside the with block.

4.3.2.3 Exceptions

Exceptions in JavaScript are defined using the C++ / Java-like statements of "try", "catch", "finally" and "throw". The JavaScript exceptions are simpler than the exceptions in Java or C++ though, because the catch statement does not define a type. When an exception is thrown, the first try-catch block up the call stack receives the exception no matter what was actually in the thrown variable.

This simplification makes it easy to translate the JavaScript structures in to non-local jumps. In C, these can be achieved using the "longjmp" and "setjmp" methods [cpl13]. The "setjmp" function is both the function that sets the long jump pointer and the destination for the long jump pointer. This means that when the program returns from setjmp it is either because it gives a reference to itself or because a jump was performed to it. The setjmp returns a integer value to let the program know if the jump label was set or if a long jump was performed.

With access to dynamic, non-local jump pointers, the implementation strategy might be the following:

- Whenever a "try" statement is encountered, we add a long jump pointer to a stack. The pointer is associated with a jump to the "catch" section, and otherwise the execution just continues in the try block. This could be written as `if(setjump(top_of_stack_ptr)) goto CATCH;` - meaning that if a long jump was performed we perform a local jump to the catch section.
- If we encounter a "throw" statement, we follow the top-most long jump pointer to the appropriate catch section.
- If we reach the end of a "try" block - we pop the top-most long jump pointer

Since we can throw outside of a try-catch structure, we need to add a default pointer to the stack to a global "catch" function that will print the exception and terminate the program.

Once this structure is in place, we can add throw statements to the native library at the appropriate places. To do this we could implement statements for pushing and popping to the stack of long jump labels in the TAC code, as well as a statement for throwing an object.

4.3.2.4 Getters and Setters in objects

Object getter and setter functions are not supported. The getters and setters allow functions to be attached to a property to handle the updating and load of the property. Compared to a regular property the presence of getters and setters adds the complication that every object property can then be either a normal property or contain a getter and/or setter function.

This means that there are potentially two functions associated with one property.

One way to solve this is to have a special JavaScript value for the case with getters and setters, that can contain up to two functions. If, when reading a property, this value is found, the getter is executed and the result returned. If a setter is found, it is executed with the value as a parameter.

4.3.2.5 For-in loops

The JavaScript for-in loops iterates over the property names of the object (or in the case of arrays, over array indices previously set).

That means, that the for-in loops requires an iterator for object properties, but when this is available, the for(var foo in bar) can be thought of as a normal for loop.

So the following JavaScript loop

```
for(var foo in bar){
...
}
```

would become this pseudo-JavaScript loop

```
var itr = bar.getIterator();
for(var foo = itr.first();
    itr.hasNext();
    foo = itr.next()){
...
}
```

that would then be translated as a normal for loop.

Obviously we need a naming convention to avoid names to clash with JavaScript names and to allow nested for-in loops. And the iterator needs to be tagged in a way that signals to the compiler that the iterator is a native function and not a JavaScript function that can be overwritten.

4.3.2.6 Debugger statements

The debugger statement produces a break point, if a debugger is attached. Since we are not implementing a debugger, we are free to ignore this statement.

4.3.2.7 Regular expressions

Efficient regular expression evaluation is a research topic in its own right, and thus outside the scope of this project [Cox07].

To avoid re-implementing everything, a way to include regular expressions in the project compiler might simply use a regular expression library for C - however, special attention to the exact syntax used is required. Most likely a transformation of some sort will be required to ensure that the regular expressions are executed correctly.

4.3.2.8 instanceof operator

The JavaScript "instanceof" operator returns a boolean value indicating if the left hand side is an instance of the function on the right hand side that is if it has this function in its prototype chain.

The internal [[HasInstance]] [ecm11][8.6.2] function on Object is not implemented in the project compiler - when this is implemented the instanceof is straight forward to implement in the same manner that the rest of the operators were.

4.3.2.9 void operator

The "void" operator in JavaScript is used to evaluate an expression only for its side effects. The void operator always returns "undefined" no matter what the expression given returns. It is straight forward to implement with the rest of the operators.

4.3.2.10 in operator

The JavaScript "in" operator returns a boolean value to indicate whether the left hand side is a name for a property of the right hand side expression.

Could be implemented like the other operators using the internal "has_property" function, but was excluded to limit the scope.

4.3.2.11 Unicode strings and identifiers

The project compiler translates JavaScript initially to C - the identifiers are written in the output code to allow for easier debugging of the code. This means that the project compiler limits the accepted identifiers in the JavaScript code to the identifiers that c supports. This has the effect that the project compiler is limited to ASCII identifiers.

This is easy to change: simply give all variables a new unique name and don't append names to functions. This will make debugging the output code harder though.

To implement the Unicode strings, we could change the representation of strings from arrays of chars to an implementation from for instance the International Components for Unicode libraries of IBM [IBM13].

4.3.2.12 Native library functions

The project compiler implements a subset of the native library described in the JavaScript specification. The subset implemented is the following:

- Object: "toString" is supported, and a partial object constructor that does not support all cases defined in the ECMAScript Language Specification is supported.
- Function: "call" and "apply" are supported
- Array: Array constructor (both as function and constructor), "push", "pop", "shift", "unshift" and "concat" are supported
- Math: All the properties of the Math object are supported.
- Date: Date constructor, "getTime" and "valueOf" are supported
- console: "console.log" is supported

eval This specifically excludes "eval" and its aliases (the Function constructor among others).

The usage of eval in the wild is claimed to be quite widespread [RHBV11]. The usage measured includes aliases like "document.write('<script src=...>');" which are not really used on the server side - NodeJs, however, contains several aliases of its own for specific use server side: "runInNewContext", etc. [tea13h].

The eval function was excluded from the project to keep the scope limited.

An implementation strategy could be the following:

Since strings provided to the eval function can contain arbitrary code, the only general solution is to provide a callback to a JavaScript implementation such as the compiler itself with the string to be evaluated.

Solving the problem in this manner creates a few challenges:

- The newly compiled code must be loaded in to the already running program. This can be done with a technique called dynamic library load operation.
- The scope of the function must be correct. This can be solved by defining the new code as a function and calling it with the local environment.
- The compiler must be told that the code being compiled is eval code and the program and compiler must agree on a name for the function: By invoking the compiler with an argument to produce a libary with a named function both of these issues can be solved

- Due to the large overhead of invoking eval the compiled result should be cached.

No matter how this is solved, the use of ahead-of-time compilation will have the problem that the eval function will make the program stall, because it is forced to do compilation of the entire string. The only way to solve this problem in general would be to use the just-in-time technique or the interpretation technique.

4.4 Verification of the compiler

To verify that the JavaScript code translates into a functionally equivalent native binary a number of tests have been constructed.

These tests have known output and will produce error messages if anything is not computed as expected.

To ease the use of these tests the project compiler includes a TestSuite file that contains all the test in one JavaScript file. When executed the TestSuite will run all tests and report either "[OK]", if all tests passed or a message like "[1 of 30 failed]" if any failed. To ensure that the asserts really do show the error messages on errors one test is hardcoded to report an error.

The TestSuite tests the following aspects of the JavaScript language:

- Basic control flow
- Functions and environments
- Recursion
- Corner cases: a parameter named "arguments", a parameter named the same as the function itself, etc.
- Native library functions (Array, Date, Math, toString, valueOf)
- Object properties
- Object inheritance (prototype chain)
- Higher order functions
- Operators (unary, binary, precedence), order of execution

In addition to these tests, the benchmarks included also exercise a large portion of the language and report any calculation errors.

4.5 Description of the tool

The project compiler is delivered as a runnable jar file. When run on a JavaScript file it will produce a temporary .c file containing the translated JavaScript program. This file will then be compiled and linked against the run-time support library using GCC to produce the final, native binary.

To actually compile a JavaScript file, a command like the following is used:

```
java -jar compiler.jar input.js output/output.c [FLAGS]
```

where "input.js" is the input file, "output/output.c" is the output destination - the .c file is the name of the temporary c file that will be compiled by GCC.

Build scripts for GCC In addition to the temporary .c output file, the compiler will place a build script to run GCC as well as all the dependencies (run time support library and Garbage Collection code). The first time the build script is executing in a new folder, the Garbage Collection system will be build for the current platform.

Flags The compiler allows the following two sets of flags:

- Debug flag: -DEBUG
- Optimization flags:
  - -Oconstant_progagation: Enables constant propagation
  - -Oinlining: Enables inlining (and constant propagation)
  - -Odead_code_eliminiation: Enables Dead Code Elimination
  - -Otype_inference: Enables Type Inference
  - -Oall: Enables all optimzations mentioned above

Using the debug flag will print the AST, TAC code and IR tree to the console as well as enable GCC debug and warnings ("-wall -ggdb") and disable all GCC optimizations.

Chapter 5

Testing and performance analysis

This chapter will describe the benchmarking of the project compiler and three contemporary JavaScript implementations. We will explain what we want to measure and the exact setup for running the benchmarks. We will then present the results of the benchmarking and explain the reasons for the observed performance.

5.1 Benchmark Design

There are two overall goals for the benchmarks:

1. To compare the performance of the output programs from the project compiler to that of contemporary Javascript implementations.
2. To measure the effect on performance of each individual optimization implemented.

The performance will be measured in terms of run time as well as in terms of peak memory usage.

5.1.1 Comparison to contemporary implementations

The first goal of the benchmark is to compare the performance of a benchmark program when it is compiled by the project compiler to when it is run by a contemporary interpreter or compiler. Therefore, fair benchmark programs should be chosen. In this context we consider a benchmark program to be fair if it is a program that exercises features that would occur in real life production code, preferably uses a wide selection of language features, and that is also not constructed to favor any of the interpreter / compiler implementations.

This test program is to be run with and without optimizations for the different interpreters / compilers. This is done to investigate if the differences in performance of the implementations are the result of optimizations or other design choices.

5.1.2 Measure the effect of optimizations

The second goal is to measure the effect of the different optimizations implemented in the project compiler. The goal is to demonstrate the effect of each optimization in terms of total run-time reduction, and in terms of any differences made to the distribution of time spent in translated JavaScript code, the Garbage Collection subsystem, native libraries and the runtime support code. The runtime support code is the code that does operations, type coersion, property lookups and so on. The native libraries are the C-libraries that implement functions such as copying arrays, getting the length of a string and handling output.

5.2 Benchmark setup

In the following section we will describe the benchmark setup. This includes the programs included as the input to the benchmark, the flags given to the JavaScript implementations, the output measured and the test setup for running the tests.

5.2.1 Chosen Benchmarks

We have chosen 3 programs to use for the benchmark.

5.2.1.1 Delta Blue

Delta Blue is a one-way constraint solver originally written in Small Talk, but now used as a benchmark for a range of different languages (Java, Dart and JavaScript among others) [MW98] [Avr13] [Hem13]. The implementation that we have used is taken from the "Octane" benchmark suite used by Google, Mozilla and others to measure the performance of their JavaScript implementations [Tea12]. The Delta Blue benchmark is just one out of the 13 benchmarks that constitute the Octane benchmark suite.

We chose to remove the references to the "window" global object, to make the benchmark run in a non-browser context. Furthermore, we have changed the program in minor ways to have the test fit in the subset of Javascript that the project compiler supports:

1. The test runner of Octane and the Delta Blue benchmark are combined in one big file rather than using "require" to import Delta Blue.
2. "throw", "try" and "catch" are removed. "Throw" is replaced with a call to "console.log" to show the user if any exceptions would have been thrown.
3. "switch" statements are replaced with "if - else" constructs.
4. The number of iterations performed in each run of the benchmark function is reduced.

This also means that the scores produced in this report by the Delta Blue benchmark are not completely equivalent to scores produced by the standard Delta Blue benchmark, as available from the Octane website [Tea12].

The Delta Blue benchmark gives a numeric score calculated in the following way:

1) First the benchmark is run as many times as the implementation allows within a second.

2) Next the average time per run is calculated in microseconds.

3) Lastly, a score for the benchmark is calculated as $\frac{reference}{average\ time}$, where reference is a fixed reference runtime for the benchmark given by the benchmark authors. The reference score is 66118 ms for Delta Blue.

This means that if the implementation being tested uses less time per run compared to the reference, then the score will be greater than 1, otherwise it will be less than 1.

The score is multiplied by 100 before being reported. We can therefore interpret the score as a percentage improvement compared to the reference, that is a score of 110 means that the reference used 10 % more time per run than the version being benchmarked.

Therefore, if one implementation gets a score that is twice as big as another implementation it means that the time spent in each iteration of the benchmark function is half compared to the other.

5.2.1.2 Dijkstra

The Dijkstra test is an implementation of the single-source shortest path algorithm [Dij59] that uses only few memory allocations during the run of the algorithm. It is included to test the speed of the JavaScript implementation when stressing the Garbage Collector subsystem as little as possible.

The time measured is the run time of the algorithm only (the graph setup time is disregarded). The test loop is also run several times to allow the JIT implementations to perform their optimizations.

5.2.1.3 Loops

The last test is a simple loop structure that measures the time to add all the integers from 1 to one million (using a naive implementation). This is done 10 times.

The test is included mainly because it lends itself nicely to certain classes of optimizations. It can also be useful to explain fundamental differences in compilation strategies and for comparing different levels of optimizations within the same strategy.

The time measured is the total time to complete the test. The test is run only once.

5.2.2 Benchmarking against contemporary implementations

The benchmark will be done by running the benchmark programs in the project compiler, NodeJs (as a front-end to the Google V8 JavaScript implementation) and RingoJs (as a front-end to the Mozilla Rhino JavaScript implementation). All three implementations will be tested with and without optimizations. Both the interpreter and compiler of Rhino will be benchmarked.

Specifically, we will run the implementations using the following options:

The project compiler Each program will be run with:

- no optimizations, (abbreviated "No opt" in tables)
- only "Constant Propagation" enabled, (abbreviated "CP" in tables)
- only "Constant Propagation" and "Inlining" enabled (abbreviated "Inlining" in tables)
- only "Dead Code Elimination" enabled (abbreviated "DCE" in tables)
- all optimizations enabled, including type inference. (abbreviated "TI" in tables)

These optimizations are used before producing the C output code. For the translation from C to native binary we use GCC with all optimizations enabled.

RingoJs Each program will be run with:

- RingoJs in iterpreter mode, (abbreviated "interpret" in tables)
- RingoJs in compiled mode without optimizations, (abbreviated "No opt" in tables)
- RingoJs in compiled mode with all optimizations, (abbreviated "All opt" in tables)

For the benchmarks we will be using RingoJs version 0.9.0.

NodeJs NodeJs has options for activating or deactivating each of the individual optimizations. We measure the performance with:

- no optimizations (abbreviated "No opt" in tables)
- only "inline caching" (abbreviated "IC" in tables)
- all optimizations (abbreviated "All opt" in tables)

The decision to do a test with only "Inline Caching" was based on a test of the effects of each optimization flag on the Delta Blue benchmark. The inline caching is by far the most effective single optimization to enable. [1]

For the benchmarks we will be using NodeJs version 0.10.11

5.2.3 Measurements

We will make benchmark runs where we measure the following:

- The run-time results from the benchmark programs. We do this because we want to measure how the different implementations compare to each other as well as how optimization affect the run-time of the implementations.
- The peak memory usage of the benchmark runs. We want to compare the memory usage of the different implementations and also see if the optimizations affect the memory usage. The usage will be measured using the "memusg" tool [Shi13].
- The startup time for the implementations. This will be measured by making a run of the tests, where we measure the time from when we start the compilation to when the implementations have finished execution. This will be measured using the unix "time" program. From this we will subtract the time that is used on executing the benchmark. This will be measured using the Date objects of Javascript. The project compiler will report the actual compile time itself, and this is used instead.

Each run-time test will be run 50 times and the average calculated from those runs. For memory usage, 5 runs of each test will be used to calculate the

[1]The results for all NodeJS optimization flags are available in section F.1

average. The startup timings are run only once to avoid any caches in the implementations to affect the results.

The compilation is done just once per test.

5.2.4 Profiling the optimizations we have made

The above measurements will show how our optimizations will impact the run-time and memory use of our program. To find out how much time is used in the individual functions of our program, the garbage collection, native libraries and the run-time support code, we will also perform benchmarks where we use the Valgrind profiling tool [Dev13c]. The profiling tool will be run on the same binaries that the benchmark was performed with.

Since the memory usage is dramatically increased when profiling, some of the benchmarks will not be able to finish when profiling. For these tests, a fixed timeout is set - so rather than having profiling data for a complete run, we will have profiling data for the first 10 minutes of benchmark profile run. It would of course be better to have full runs, but this choice will at least give us some consistency when we compare the results before and after optimizations. It does however introduce some inconsistency, since the faster version will have executed a larger percentage of the program.

5.2.5 Machine

All measurements will be performed on an Amazon AMS virtual machine of the type "m1.medium" [Ser13b]. This is a virtual machine with the following specifications:

- 64 bit Amazon Linux Distribution
- 1 vCPU (virtual CPU)
- 2 ECU (EC2 Compute Unit, where 2 ECU roughly equals a processor that is twice as fast as a 1.2 GHz 2007 Intel Zeon processor [Ser13a])
- 3.75 GiB memory

The machine is set up to run only the benchmark process when measuring the performance, and we only do one benchmark at a time.

5.2.6 Limitations of the benchmark setup

We have designed the benchmark to produce as fair and consistent results as we could, however, we have identified some limitations of the benchmark design and setup, that we have not been able to fully remove or work around.

5.2.6.1 Language support

Since the project compiler does not support the full ECMAScript Language Specification, the project compiler will always have an unfair advantage over full implementations. Even if none of these features are actually used, the fully compliant implementations still have code to support them, and this could potentially slow these implementations down compared to an implementation that does not. Specifically, the project compiler implements a simplified object property model compared to the specification that could be significant in comparison to a fully compliant implementation. It also means that even if the programs have the desired property of using a wide selection of language features, there are language features that will not be exercised.

Partial mitigation We have removed all explicit use of the unsupported features from the Delta Blue benchmark to avoid giving the project compiler an unfair advantage and to make sure that it can compile and execute the program. For instance, all "try", "throw" and "catch" statements are removed to avoid forcing the overhead of exception handling to the other implementations - this, however, cannot fully be avoided as the there are still some operations that can throw exceptions and the other implementations will handle these cases, whereas we do not. The simplifications in the object mentioned above cannot avoid to give the project compiler an unfair advantage.

Other limitations, such as not supporting automatic semi-colon insertion, should not give our compiler an advantage over the other implementations.

5.2.6.2 Compile time

The benchmark run times from the output of the project compiler does not include the compile time. This is obviously a design choice because we have implemented an ahead of time compiler where all the compilation is finished before execution. However, since we compare the project compiler to interpreter and JIT compiling systems that do compilation during run time it needs to be addressed.

Partial mitigation The Dijkstra and Delta Blue benchmarks include a "dry-run" that does not count towards the measured run time. This gives the JIT compiler a chance to identify "hot" functions and optimize these. The benchmark also uses timings inside the program to report the run time rather than measure the run time of the process. This allows us to avoid including the parse timings for the interpreters and JIT compilers in the results.

5.2.6.3 Startup time

When we measure the startup time for the project compiler, all the time is used on doing the compilation. When we measure the startup time of NodeJs and RingoJs, some of the time may also come from starting up the server features, of these implementations. When we do the measurements of the startup time for the implementation, our compiler has a clear advantage over NodeJs and RingoJs because we have not implemented all the server features, that these implementations have.

Partial mitigation This is a flaw in our test we have not resolved. It means that even if our compiler uses less time to start up than the other systems, we can not conclude that it is a faster program. In the case that our compiler is slower, we will, however, be able to conclude that NodeJs and RingoJs can startup and are ready to begin execution faster than our compiler.

5.2.6.4 The benchmark suite used

The Delta Blue benchmark used is designed and developed by teams that design and implement JavaScript JIT compilers for use in a browser context. This raises the concern that the benchmark is focused for use in the browser and therefore does not give a good measurement of performance for a JavaScript implementation that is primarily designed to be used server side.

Partial mitigation Although the Octane benchmark suite is designed to test the JavaScript implementations of web browsers, there is no tests of the DOM interactions. We therefore believe that it is a fair tests to use. If anything, the benchmark will give NodeJs an advantage over our compiler, since it is optimized towards this benchmark as it is part of the continuous evaluation of the V8 performance.

The Dijkstra benchmark does not carry this concern because it does shortest path finding, which is an application that has seen use in applications run on servers such as Google Maps. The loops benchmark is so simple that it is not focused on any particular context.

5.2.6.5 The benchmark selected from the suite

The octane benchmark suite contains 13 different benchmarks, but we have included only one of them in our tests. This means that we do not get the coverage offered by the entire benchmark suite and one could raise the concern that we might have cherry picked a test that the project compiler does really well.

Partial mitigation The test that we have chosen tests primarily an object oriented programming style in JavaScript. The other tests of the suite benchmark features that we do not support (RegExp, for each, eval, etc.) or are simply too large for us to convert for use by the project compiler. To convert them we, among other things, must manually insert every missing semi-colon in the file - the Box2D test is 9000 lines of code.

5.2.6.6 Use of production code in the benchmarks

One of our goals when making the benchmarks was to use code that was close to production code, that could be of use in the industry. One could raise the concern that the benchmark programs do not have this property.

Partial mitigation We believe that the Delta Blue and Dijkstra benchmarks have this property to a high degree because they do constrains solving and shortest path finding, respectively. These are disciplines that have many practical applications. The coding style of Delta Blue is object oriented, which is a very popular style in the industry.
The loops benchmark, however, does not have the property. Even if the structure of the program does resemble some useful algorithms such as 2D folding, the program is extremely simple, and it does not compute anything that would

be of any value in real life. Therefore we will only use the loops benchmark to make conclusions on how well the compilers can optimize very simple program constructs.

5.3 Benchmark results

In this section we will present and discuss the results we measured when running the benchmarks described above. The raw results are presented in the tables in the appendices of the report.

5.3.1 Profiling the project compiler

First we will present the results we got when we profiled the project compiler, to find out where the time is being used when a benchmark is run. We will discuss if the distribution is reasonable or if it indicates that there is room for improvement in certain parts of the compiler.

5.3.1.1 DeltaBlue

The time spent in every function (where a non-zero amount of time was spent) was aggregated in to the following table:

Table 5.1: Profiling results for "DeltaBlue" in the project compiler

Category	Time (Optimized)	Time (Non-optimized)
Garbage Collection	54,41%	54,57%
Run-time support	28,89%	28,76%
Native libraries	14,54%	14,82%
"JavaScript"	1,25%	1,21%
Unknown	0,75%	0,8%

The average scores that the compiled executables got in the benchmark were:

A large amount of the total time is spent in the garbage collection in particular, and to a lesser degree the run-time support library. To get a clearer picture of

Table 5.2: Average benchmark results for "DeltaBlue" in the project compiler

	Optimized	Non-optimized
Score	718	708

where the time was being spent, we have included the top five most expensive functions in terms of run-time.

Table 5.3: 5 most expensive functions in the Non-optimized "DeltaBlue" benchmark

Function	Category	Time (self)
GC_mark_from	GC	20,23 %
assign_property	Runtime support	12,38 %
get_own_property	Runtime support	8,37 %
GC_reclaim_clear	GC	6,92 %
GC_malloc	GC	6,49 %

We observe that the single most expensive function in this benchmark is the garbage collection function responsible for marking the objects as being used or not, GC_mark_from. The fourth and fifth most expensive functions are also related to the garbage collection (GC_malloc which is acquiring memory from the operating system and GC_reclaim_clear which is internally reclaiming memory that is no longer being used).

The hash map (uthash) used to store the properties in objects is implemented using macros, which means that any time spent in the hash map will show up in the parent functions assign_property and get_own_property when profiling. It is therefore likely that the majority of the time spent in these two functions in particular are related to the hash map operations for storing and retrieving properties.

It seems reasonable that much of the time is used on reading and writing properties, because the Delta Blue benchmark is written in an object oriented style. This can also be the explanation of why a lot of time is used in the garbage collector, because the objects are put on the heap.

The code also contains a large number of function calls. Due to the way the project compiler is implemented, where environments are put on the heap, this will also cause a lot of garbage to be created.

That said, the amount of time spent in Garbage Collection and object property run-time support is much higher than what seems reasonable. This indicates that if we want to reduce the run time, garbage collection and object property run-time support are the most beneficial place to make optimizations. This also indicates that implementing further optimizations to run on the JavaScript code will most likely not produce significant performance gains before the issues with the Garbage Collection subsystem and memory layout of JavaScript objects are resolved.

If we compare the results without optimizations to the results with optimizations, we do not see much of a difference. Less time is, however, used in the support library for coercing variables [2]. As expected, however, this does not have much of an effect, if any, on the overall performance as can be seen in the benchmark.

5.3.1.2 Dijkstra

The profiling data included for the Dijkstra test is from the entire run of the program.

The time spent in every function (where a non-zero amount of time was spent) was aggregated in to the following table:

Table 5.4: Profiling results for "Dijkstra" in the project compiler

Category	**Time (Optimized)**	**Time (Non-optimized)**
Garbage Collection	7,07 %	6,44 %
Run-time support	51,03 %	54,93 %
Native libraries	34,96 %	31,99 %
Javascript	6,88 %	6,59 %

The average run time that the compiled executables reported in the benchmark were:

Table 5.5: Average benchmark results for "Dijkstra" in the project compiler

	Optimized	Non-optimized
Time	13674	15309

Compared to the DeltaBlue benchmark the time spent in the Garbage Collection subsystem is significantly lower. The time spent in the support code, however,

[2]See table E.1 and E.2 in Appendix E

is relatively large. Time spent in the five most expensive functions is shown in the table below:

Table 5.6: 5 most expensive functions in the Non-optimized "Dijkstra" benchmark

Function	**Type**	**Time (self)**
get_own_property	Runtime support	36,03 %
vfprintf	Native lib	12,19 %
jsfunc_5_Dijkstra	JS	6,88 %
get_property	Runtime support	6,09 %
__memcmp_sse4_1	Native lib	5,56 %

In the top five functions we also see two native functions: "vfprintf" is used to translate an index into a string, so that it can be used to look up a property (even arrays in JavaScript needs to support strings as property indices). "memcmp" is used to compare strings - probably as part of the hash map lookups here.

Again it seems reasonable that a lot of time is used on reading and writing properties, because nodes and vertices are stored as objects. Most of the time is used reading because the test is written in a way that requires a lot more object property look-up than writes (for comparison, 0,16% of the total time is spent in "assign_property" in this test). The benchmark does not create a lot of garbage because the temporary results of the algorithm are written in only a few variables, and the results are stored as simple values.

When we compare the distribution of time without optimizations to the distribution with optimization we see that there is not much of a difference. The explanation again is that too much time is used for reading. We do, however, see that much less time is used on conversion and operators [3]. From the benchmark we can furthermore conclude that this actually has made the program faster. This can be explained by the fact that in the code for Dijkstra, most of what is done is operations on simple values, so in this case, the optimizations are able to give an improvement.

5.3.1.3 Loops

The profiling data included for the Loops test is from the entire run of the program.

[3] See table E.3 and E.4 in Appendix E

The time spent in every function (where a non-zero amount of time was spent) was aggregated in to the following table:

Table 5.7: Profiling results for "Loops" in the project compiler

Category	**Time (Optimized)**	**Time (Non-optimized)**
GC	0 %	0 %
Runtime support	59,51 %	79,35 %
Native lib	0 %	0 %
JS	40,45 %	20,65 %

The average run time that the compiled executables reported in the benchmark were:

Table 5.8: Average benchmark results for "Loops" in the project compiler

	Optimized	Non-optimized
Time	1179	3548

Since there are only five variables in total in the Loops benchmark, it is expected that no time is being spent in the Garbage Collection sub system (the garbage collector will only start when a certain amount of memory has been allocated). The use of native libraries is also very limited (a few calls to "time.h" and one print to the standard out), so the time spent there is expected to be too low to measure. However, the fact that more than half of the time is being spent in the run-time support is still surprising. The optimizations clearly help in this regard, but even so, almost 60 percent of the time being spent in the support code is surprising. The top five most expensive functions for both the optimized and non-optimized versions are included below.

Table 5.9: 5 most expensive functions in the Non-optimized "Loops" benchmark

Function	**Type**	**Time (self)**
jsfunc_0_additionLoop	JS	20,65 %
addition_op	Runtime support	18,11 %
to_number	Runtime support	15,22 %
to_primitive	Runtime support	11,96 %
assign_variable	Runtime support	9,06 %

The single most expensive function is the function containing the main loop of the test. However for every iteration of the loop a new operand must be created, the addition operation must be run and the result must be written to the result variable. The effect of this is that these support functions each take almost as much time as the loop function itself.

Table 5.10: 5 most expensive functions in the Optimized "Loops" benchmark

Function	**Type**	**Time (self)**
jsfunc_0_additionLoop	JS	40,45 %
assign_variable	Runtime support	28,89 %
value_at	Runtime support	23,11 %
create_numeric_value	Runtime support	4,62 %
create_boolean_value	Runtime support	2,89 %

The optimized version gets rid of some of the support code. In particular the "+" operator can be translated as a C "+" operator rather than a call to `addition_op` because the type inference allowed the compiler to see that both operands would always be numeric. This also removes the calls to the type coersion. We still need to store the result, so the calls to `assign_variable` take time - a larger percentage, because the overall time is down. The numerical constants are still being initialized for every iteration of the loop, as the `create_numeric_calls` reveal.

Since we used GCC's optimizations in both runs, the results indicate that it can be a good idea to do optimizations on a level closer to the JavaScript code than on the level of the produced C code.

5.3.2 Results of optimizations

In terms of overall performance gains of the optimizations implemented in the project compiler, the differences in average run-times can be summed up in the following table:

Table 5.11: Average run-times for different optimization levels for the project compiler

Benchmark	No opt	CP	Inlining	DCE	TI
Delta Blue	708,6182	704,47	720,2938	716,4356	718,76
Dijkstra	15309,39	15212,73	15104,04	14993,8	13674,06
Loops	3548,26	3639,86	3631,56	3431,66	1179,5

The overall effect of optimizations on the Delta Blue score is close to non-existent. The test contains a lot of small functions and this limits the effects of the optimizations as implemented in the project compiler. Also, every time a function is called, we have to assume that it touched all variables unless we can explicitly prove that this is not the case. This further limits the effect of the optimizations in the project compiler.

For the tests that have a larger area for the optimizations to work, we do see some effect. Constant propagation has close to no effect on its own - which makes sense, since "Dead Code Elimination" is needed also to actually lower the number of statements executed. Inlining, as currently implemented, is very limited in the project compiler, meaning that it does not see any opportunities to inline functions. Inlining would provide a benefit on its own because it would get rid of the function call overhead (which is larger in the translated JavaScript compared to regular c code, because of the environment and arguments mapping).

Dead code elimination does provide a limited speed up, when executed in combination with constant propagation. Without any other optimizations than Dead code elimination there are (mostly) no dead statements to eliminate, so the it included here only in combination with Constant propagation. Finally when type inference is added a lot of the run-time support calls can be eliminated. The effect of this is significant for the Dijkstra and in particular the Loops test (with an 11 % and 67% reduction in run-time respectively).

5.3.3 Performance comparison between the project compiler and NodeJs

In comparing NodeJs to the project compiler we are not only comparing two different implementations of JavaScript compilers, but two different compiling strategies, namely just-in-time and ahead-of-time compilation. It is important to note, though, that the tests were specifically designed to disregard startup time, so in that regard it is more of a comparison between the compiled programs than a comparison between ahead-of-time and just-in-time compilation.

When comparing the benchmark results of the project compiler with the benchmark results of NodeJs, we see that the NodeJs performs better in every test we performed. There are, however, significant differences in how much better NodeJs performs in the different tests.

For the DeltaBlue test, we observed that NodeJs on average got a score that was 6 times better than that of the project compiler. In the table we have also calculated what average times the scores correspond to using the formula shown earlier. For the Loops test, NodeJs used, on average, about one eighth of the time that the project compiler did. The Dijkstra test stands out in this regard, because the project compiler "only" used twice as much time as the NodeJs did.

We also observe that the project compiler uses much more memory for those tests that contain object allocations. Especially when taking the NodeJs memory

Table 5.12: Average benchmark scores and memory usage for the un-optimized systems

Test	NodeJs	Project Compiler
Delta Blue (score):	4288	708
Delta Blue (avg ms):	15.4	93.4
Dijkstra (ms):	7494	15309
Loops (ms):	428	3548
Empty (Mb):	4,2	0
DeltaBlue (Mb):	9,8	342
Dijkstra (Mb):	11	83
Loops (Mb):	9,4	0,87

overhead in to account, as measured by the "empty" test, which is just an empty text file. For the Delta Blue test the project compiler uses about 60 times more memory than NodeJs uses for the program. Even for the Dijkstra test the project compiler uses about 12 times more memory to represent the same JavaScript objects.

It is interesting to note, that while the performance results of the project compiler are within the same order of magnitude, NodeJs still performs much better. This tells us, that the basic translation from JavaScript to native executable is significantly better in NodeJs compared to the project compiler - especially when taking in to account that the project compiler gets the benefit of all the optimizations available in GCC.

Compared to the NodeJs profiling results the time spent in the garbage collection sub system really stands out: Where the project compiler, as we saw in the profiling, spends almost 55% of the time on the garbage collection in Delta Blue, NodeJs spends around 2% [4]. This clearly indicates that the project compiler has a problem in either the garbage collection subsystem itself or in its memory usage pattern - possibly both. However, since the Boehm garbage collector used by the project compiler is also used in several widely used open source projects, it seems unlikely that the garbage collection implementation alone is responsible for the results observed. It is more likely that the memory usage pattern of the project compiler is a major contributor to the time spent in the garbage collection. Specifically that the project compiler allocates all variables and all environments on the heap regardless of whether these are local or not.

While the time spent in the garbage collection subsystem explains about half of the time spent by the project compiler, this does not fully explain why NodeJs

[4]See F.2 in Appendix F

is faster. If we disregard the garbage collection time completely, then NodeJs uses on average 15,4 ms · (1-0.02) = 15,1 ms and the project compiler uses on average 93,4 ms · (1-0,55) = 42,0 ms. That is, NodeJs still uses only about one third of the time of the project compiler. From the profiling of the project compiler we see that about 95 % of the non-garbage collection time is spent in either run-time support code or in native libraries. It turns out that 70% of the non-garbage collection time is actually spent dealing with object properties. The most expensive functions in this regard are `assign_property` and `get_own_properties`. These functions are responsible for assigning and retrieving the JavaScript values from the object property hash-map and include the time spent in the actual hash-map implementation. These two functions alone account for about 46% of the non-garbage collection time. We also see that the native string handling functions related to the use of object properties take up a significant amount of time.

The DeltaBlue benchmark makes heavy use of objects and properties, so it is not completely unexpected that object properties would take up a significant amount of time. However, since V8 spends about 0.5 % of its time defining properties and even less time retrieving properties[5], it shows that the project compiler's implementation of object properties is very inefficient compared to that of V8.

Garbage collection and object property handling account for roughly 86 % of the time spent by the project compiler for the DeltaBlue benchmark. If we could reduce this time to zero, we would have an average time of 93,4 ms · (1-0,86) =13,1 ms, which is almost the same as the 15.4 ms that NodeJs uses. However we cannot expect to be able to reduce the time spent on garbage collection and object properties to zero - especially when considering how much more memory the project compiler uses. This means that there must be further points where the project compiler is not as efficient as NodeJs.

To explain these differences, the compilation strategy for V8 is examined and compared to that of the project compiler.

V8 Compilation strategy The V8 JIT compiler has two different ways to compile JavaScript code internally: The "Full Codegen" compiler, that does the initial translation of JavaScript code to native code and the "Crankshaft" compiler, that does all the optimizing compilations. "Full Codegen" compiles one function at a time and does so lazily - that is, a function is only compiled

[5] Appendix F.3: The time spent in the function "DefineObjectProperty" is taken as a measure of the time spent defining properties. Since any function with less than 2% of the time is excluded from the profiling data we assume that less than 2% of the 27.4% for the support code is spent retrieving object properties.

when it is called for the first time. During this first compilation, profiling code is inserted that will notify "Crankshaft" when it is time to optimize a hot function [Win12a] [Win12b] [tea13a].

When disabling all optimizations, the only compiler active is "Full Codegen", so in that regard the unoptimized benchmarks are a direct comparison between the output code from "Full Codegen" and the output from the project compiler.

"Full Codegen" translates directly from the AST to the V8 assembly language - commands for a "high level" (platform independent) stack machine. "Full Codegen" does not have support for any real optimizations - by design, because it needs to be fast, but also because it only sees one function at a time, does not have any intermediate representation to perform the optimizations on and lastly because it does not have type information [Win12a].

"Crankshaft" on the other hand, has access to all those: it uses two intermediate representations internally: Helium (high level IR) and lithium (low level IR) before outputting the V8 assembly. These allow the compiler to execute the data flow algorithms necessary for the optimizations. Crankshaft also has access to type feedback from the un-optimized code when performing its compilation. The type feedback tells what the types seen so far are, and this allows Cranckshaft to compile a version that is optimized to these types, and to simply put a de-optimize instruction for when other types are seen. The V8 system can swap between optimized and de-optimized versions of the code without restarting any functions using a technique called on-stack replacement - that is, it allows the V8 system to replace the program code for a function that is currently being executed [Win12b].

V8 output vs the project compiler output When comparing the output assembly code generated by the un-optimized V8 engine[6] to the output of the project compiler, a few differences immediately become visible:

1. V8 can inline some of the support library calls - rather than call a function to calculate the result of an operation V8 can output the assembly instructions for that operator directly. V8 will sometimes output a code stub initially to be filled out when actually needed if the operand types are not known at compile time - this allows it to only emit the code actually needed and not for all cases of variable types, which can be quite complex, even for something as simple as the addition operator.

[6]See appendix F.4

2. To save compilation time during execution, the V8 compiler never produces text-based code for the later stages - the compiler simply calls methods directly in the assembler to get it to output the desired native code [Win12b]. The project compiler uses a number of stages (javascript -> c code -> o files -> native executable) - although this obviously happens before the program is executed.

3. V8 only forms a closure if any variables are read from outside the local scope (also, if the variables being accessed are directly up the call stack, it uses a clever mapping to avoid accessing via the closure) [Man10]. This means that most variables in V8 are local or stack variables. The project compiler on the other hand treats all variables as being in closures, so all variables are looked up on the heap.

Object representation The way that objects are represented in V8 is radically different compared to the project compiler. The project compiler uses a simple hash-map (from strings to JavaScript values) to store the object properties. V8 on the other hand tries to avoid that by using a flat memory layout (that is, all properties are placed next to each other in same way that a c struct might be layed out in memory). To allow access to the properties via their string names, a map is maintained from the name to the offset in the memory block. Every time a property is added to the object, the map needs to be replaced. V8 uses transitions for this purpose: a transition is a link from the previous map to a new map that contains all the old fields and the newly added field as well [Bak12], [Con13].

The important part, compared to a straight hash map solution, is that these maps are calculated only once and shared between all instances that hold the same properties.

The maps also allow V8 to store function pointers once per object type, rather than once per object instance, if they do not change.

Since this way of treating object properties could lead to a very large number of maps being generated, V8 also have some heuristics for when to change the object into a dictionary mode, that just uses hash maps to hold the properties [Con13].

It is important to note that V8 treats objects in this way regardless of the optimization level - it will, however, at higher optimization levels use information about the way that properties are accessed. For instance, if only integers are used to access properties in an array, it will just use an unboxed array behind the scenes if the optimization `--smi_only_arrays` is used.

5.3.3.1 Comparing the optimized versions

Table 5.13: Average benchmark scores and memory usage for the fully optimized systems

Test	NodeJs	Project Compiler
DeltaBlue (score):	221790	718
Dijkstra (ms):	225	13674
Loops (ms):	28	1179
Empty (Mb):	4,9	0
DeltaBlue (Mb):	13,5	344
Dijkstra (Mb):	12	82
Loops (Mb):	9,4	0,87

For the optimized versions of NodeJs and the project compiler the results are radically different: For the DeltaBlue benchmark, NodeJs gets a score that is 300 times better on average compared to the project compiler. The Dijkstra test now takes 60 times longer in the project compiler compared to NodeJs - and the loops is about 40 times slower in the project compiler. For the loops test we have to note, though, that timings on this scale (less than 100 milliseconds) are somewhat inaccurate - even so, there is no doubt that the NodeJs performs at least an order of magnitude better than the project compiler for this test. The memory usage, however, is not affected in any significant way by enabling the optimizations.

Based on the V8 options we see that V8 will do many different optimizations. The most significant one (at least in relation to the Delta Blue benchmark) appears to be the inline cache. This optimization alone improves the score by more than 2800% whereas all other optimizations alone degrade performance by a few percent. This suggests that in the Delta Blue test, function calls constitute a major part of the overall run-time. It also suggests that one of the reasons for the relatively poor effect of the project compiler's optimizations is that they do not really affect function calls. The inlining optimization could affect this, but it is too limited to have any significant effect on the performance. We also see that the type inference does improve performance a bit for the project compiler - the reason that we do not see a similar effect in V8, is that all type information is collected by the inline cache: without it, it cannot specialize any functions to the seen types, even with crankshaft enabled.

5.3.4 Performance comparison between the project compiler and RingoJs

Comparing the project compiler to RingoJs is interesting because it presents the chance to compare the project compiler to an interpreter and an ahead-of-time compiler that share almost all of the support code. The major difference between the two modes is whether or not the interpreter loop is used or class files are produced ahead of time. Every interpreter must contain some sort of loop or recursion over the instructions of the program being interpreted - and for each instruction it must inspect it to activate the correct function in order to execute the instruction. A compiler on the other hand simply outputs the correct function call in place of the instruction.

Table 5.14: Average benchmark scores and memory usage for the un-optimized systems

Test	RingoJs compiler	Project Compiler
Delta Blue (score):	2710	708
Dijkstra (ms):	6712	15309
Loops (ms):	1204	3548
Empty (Mb):	42,7	0
Delta Blue (Mb):	85,7	342
Dijkstra (Mb):	77,6	83
Loops (Mb):	56,6	0,87

The benchmark shows that the Java code produced by the RingoJs compiler performs better than the project compiler in all benchmarks. RingoJs' Delta Blue score is more than 3 times better than the project compiler's, and its run time in Dijkstra is "only" 44% of the project compiler, while that percentage is 34% in the loop benchmark.

While better garbage collection could explain a major part of the performance difference observed, it cannot explain the whole difference.

The object property memory model used by RingoJs for both interpreter and compiled code is similar to that of the project compiler, namely hash maps as the storage for all object properties [Tea13e] and no static access to the fields (that is, even if the string is known at compile time, it is still used as a string to lookup the correct field)[7]. However, the hash map implementation allows array indices to be used directly: that is, if a property name is actually a number, then that number is used as the hash [Tea13g]. This allows for a more efficient

[7]See appendix G

way of handling arrays, since the numeric indices can be used directly rather than transforming them to strings and then computing the hash. This does, however, mean that all strings must first be tested to see if they represent an integer value, since the hash of the string is different than the integer value it represents [Tea13d][l. 1644].

This difference could explain a significant part of the performance difference observed, especially in the Dijkstra benchmark, where the project compiler spent around 50% of the time on getting and putting data into object properties that are actually arrays.

This still cannot explain the whole difference since the loop benchmark also performs significantly better in Ringo compared to the project compiler. This benchmark is very light on the memory usage and contains no arrays. When the output code from Ringo is compared to the output code from the project compiler the results look very similar. There are a few differences though: the project compiler stores the result of the loop condition in a JavaScript value and then compares against this boxed variable. This introduces an extra value store for each iteration along with the boxing and unboxing of this value, compared to RingoJs, which just lets the comparison return a Java boolean directly. The project compiler also creates a new JavaScript value for the constant "10000" in every iteration whereas the Rhino compiler simply stores it as a class constant. Since the loop does these small operations many times in the benchmark, these differences are significant. There are also a some differences in the support code: The Rhino support code first checks for the special case of adding two numbers [Tea13d][l. 2556], before performing the general algorithm, where the project compiler only performs the general algorithm.

In terms of memory use, the RingoJs compiler performs better on Delta Blue. On the Dijkstra benchmark, the memory uses are very close. The project compiler performs better on Empty and Loops. The added memory overhead of the JVM compared with the C run-time environment explains why the project compiler uses less memory for these small tests.

We see from the benchmark that the RingoJs generally performs better than the project compiler, when they use optimizations. RingoJs gets a score that is almost 4 times better than the project compiler, and it uses only 0.48% of the time on Dijkstra. On the Loops benchmark the two compilers use almost the same amount of time to run the program.

With respect to memory, the results are very similar to the unoptimized compilers.

The optimizations that the Rhino engine can perform are quite limited. So

Table 5.15: Average benchmark scores and memory usage for the fully optimized systems

Test	RingoJs compiler	Project Compiler
Delta Blue (score):	2751	718
Dijkstra (ms):	6606	13674
Loops (ms):	1204	1179
Empty (Mb):	42,7	0
Delta Blue (Mb):	86,3	344
Dijkstra (Mb):	79,7	82
Loops (Mb):	56,7	0,87

the performance benchmark result is largely unchanged for the Rhino engine - however, the project compiler does gain a lot in the Dijkstra and Loops test. The project compiler is still not as fast for the either of the comparable tests, however.

That RingoJs still performs better than the project compiler when all optimizations are enabled for both system does not indicate that the RingoJs optimizations are better, but rather than the basic JavaScript compilation is better in RingoJs compared to the project compiler.

Table 5.16: Average benchmark scores and memory usage for the unoptimizing project compiler compared to the RingoJs interpreter

Test	RingoJs interpreter	Project Compiler
Delta Blue (score):	172	708
Dijkstra (ms):	21041	15309
Loops (ms):	20374	3548
Empty (Mb):	39	0
Delta Blue (Mb):	164	342
Dijkstra (Mb):	68	83
Loops (Mb):	129	0,87

Comparing the run-time performance of the project compiler to the interpreter reveals that the project compiler performs better in all tests than Rhino in interpreter mode.

Especially when comparing to the interpreter, the fact that we do not support the whole language might skew the results - the interpreter has to have code for supporting the whole language to be able to execute that as part of the interpreter loop. The project compiler does not and additionally only writes the instructions actually used in the source program. Furthermore, the JavaScript debugging facilities of the interpreter might also make it slow compared to the project compiler. Compared to the interpreter, the project compiler also has the advantage of having access to the GCC optimizer.

Memory-wise we observe that the project compiler uses less memory for the trivial example (Loops), but more memory for the tests that are closer to real-life production code. By examining the memory usage for the "empty" test, we see that RingoJs carries a memory overhead of about 40 Mb - although it is difficult to tell if all of that is being used or if the Java Virtual Machine simply pre-allocated some of it. Also, the interpreter has access to the JVM garbage collector - which is probably one of the most widely used garbage collection system in existence today. Given the amount of time spent in garbage collection by the project compiler this is actually a significant benefit, especially given their similar object memory models. It is, however, noteworthy that even the interpreter is able to store the same JavaScript objects as the project compiler in less space than the project compiler uses. This further indicates that the project compiler has a weak spot in the memory model used.

The benchmark shows that when RingoJs uses interpretation instead of compilation the project compiler can outperform it. It therefore seems resonable to think that it is the interpreter loop of the RingoJs interpreter which makes it slower than the project compiler. Due to the amount of extra time being used by the interpreter, the results strongly indicate that the project compiler could out-perform the interpreter even if the whole language was implemented as opposed to the current subset.

That both the project compiler and RingoJs's own compiler are able to significantly out-perform the RingoJs interpreter is a strong argument in favor of using compilers for executing JavaScript rather than interpreters if performance is important.

5.3.5 Compile timings

In order to fully compare the project compiler to NodeJs and RingoJs we need to take startup time in to account. This section will present the results we got when we meassured the startup times for the different JavaScript implementations.

5.3.5.1 The project compiler timings

The project compiler reports the time spent in each compilation phase after the whole compilation is complete. We have included the total time spent compiling and two individual phases: The lex and parse phase and the GCC compilation phase. The lex and parse phase is included for comparison with the NodeJs timings, since this is the phase that V8 needs to complete fully before any kind of partial compilation can take place. The GCC phase is included to get an idea about the time spent optimizing the final, native binary.

Table 5.17: The project compiler timings for the Delta Blue benchmark (in seconds)

	No opt	CP	Inlining	DCE	TI
Total	35,04	39,13	39,44	48,75	51,16
In GCC	3,65	3,71	3,69	3,71	4,30
Lex and Parse	28,86	29,02	29,31	29,59	28,41
Other	2,53	6,41	6,43	15,45	18,45

The Delta Blue is the largest JavaScript file of the benchmark programs, so it is not surprising that the compilation time is relatively large. The time spent in the lex and parse phase, is disproportionally large - especially when considering that the GCC phase includes a lex and parse phase of its own and still only uses about 10 % of the overall time. Almost all of the time spent in the lex and parse phase is in the auto-generated code produced by the tool ANTLR. The overall time increases when adding optimizations which is to be expected.

Table 5.18: The project compiler timings for the Dijkstra benchmark (in seconds)

	No opt	Cp	Inlining	DCE	TI
Total	8,824	9,955	10,698	13,388	14,362
In GCC	1,724	2,247	1,646	1,641	1,646
Lex and Parse	6,406	4,864	6,75	6,276	6,048
Other	0,694	2,844	2,302	5,471	6,668

Again we see that most of the time is spent in the lex and parse phase, but the results are less extreme compared to the Delta Blue test. Again the most expensive part of the rest of the compiler is the Optimizations, which, at the highest level, take almost as much time as the lex and parse phase. GCC, again, uses very little time compared to the rest of the compiler.

The loops program is tiny, so the fact that more than four seconds are spent in the lex and parse phase is very surprising. The size of the program also

Table 5.19: The project compiler timings for the Loops benchmark (in seconds)

	No opt	CP	Inlining	DCE	TI
Total	6,213	6,407	6,52	6,712	7,709
In GCC	1,335	1,345	1,343	1,342	1,335
Lex and Parse	4,106	4,111	4,215	4,071	4,384
Other	0,772	0,951	0,962	1,299	1,99

reflects on the time it takes for the project compiler to optimize it - even with all optimizations enabled, the optimization phase uses little more than a second.

Table 5.20: Distribution of time spent in compilation

	Time spent
In GCC	11%
Lex and Parse	65%
Other	24%

The distribution of time across all the compilations measured shows that the time spent in the lex and parse is disproportionally large. GCC uses a significant amount of time, but still not very much compared to the rest of the compiler.

5.3.5.2 NodeJs startup timings

For NodeJs we have timed how much of the run-time is being used outside of the test. This does not correlate exactly with the time being spent compiling. Since V8 is a JIT compiler, some of the compilation will be taking place during execution.

Due to the design of V8 the timings below are more accurately described as the time it spends on the lex and parse phase as well as compiling the outer-most function, since all other functions are not compiled untill they are called.

Table 5.21: NodeJs time spent outside the test

	No Opt	IC	All opt
Empty	0,045	0,045	0,046
Delta Blue	0,048	0,052	0,051
Dijkstra	0,129	0,064	0,068
Loops	0,047	0,049	0,052

The results do show that NodeJs spends very little time starting up. The overhead, as measured by the empty JavaScript program, is about 45 milliseconds. This indicates that V8 is able to lex and parse the Delta Blue test in about 3 - 7 milliseconds. The Dijkstra tests takes slightly longer for it to parse, but that might simply reflect that more code is in the outer-most function. The loops test is almost instantaneous with about 1-2 milliseconds spent lex, parsing and compiling the outer-most function.

5.3.5.3 RingoJs startup timings

For the RingoJs timings, the time spent outside the tests includes both the startup overhead as well as the compilation overhead. To try and differentiate the two, we have included timings for the empty JavaScript program. Rhino does ahead of time compilation, so the entire compilation overhead occurs before the test begins. We have included the interpreter as well to compare the time it takes to start up with the time it takes to first compile, and then start the program.

Table 5.22: RingoJs compile timings

	Interpret	No opt	All opt
Empty	1,971	1,005	1,005
Delta Blue	2,933	1,233	1,234
Dijkstra	3,944	3,462	3,461
Loops	3,434	1,046	1,057

It is interesting to note, that we see practically no difference between the time it takes Rhino to compile with and without optimizations enabled. Also quite surprising is that the interpreter actually takes longer to start than doing the compilation. The compilation times for the Delta Blue test are interesting when compared to the project compiler - the entire Rhino compilation is completed about two seconds faster than even just the GCC phase.

5.4 The project compiler performance bottlenecks

The project compiler only outperforms the interpreter - the other compilers have better performance than the project compiler. This indicates that there is a significant benefit in simply compiling the JavaScript code rather than interpreting it - even with the drawbacks of the project compiler, that we have seen in the results.

The results indicate that there are four major bottlenecks in the project compiler:

1. Garbage Collection: This is a major bottleneck with upwards of 55 % of the time spent in the garbage collector for some benchmarks, but there is no single fix for it. Although the garbage collector is not directly under our control, the number of heap allocations are. Since the garbage collector is activated only when additional memory is needed for a heap allocation, reducing the number of heap allocations would reduce the time spent in the garbage collector.
2. Object properties: the object property model is also a major bottleneck. The current implementation forces all indices to be strings, even for array indices. Introducing a mechanism for special casing integer indices, as seen in arrays, and cases where the property name is known at compile time could help to reduce this bottleneck.
3. Function calls: although function calls only account for a little over 2 % in the DeltaBlue benchmark, it is actually a significant part of the time not spent in the Garbage Collector or with the object properties. Especially since part of the overhead is introduced by creating the "arguments" object - which also adds to the memory allocation pressure. Improving the function call mechanism could be done by creating the "arguments" object only for those function that use them and using a different mechanism to pass a variable number of arguments to the functions.
4. Run-time support code: Native libraries also constitute a significant part of the overall run-time. Although some of it cannot be avoided (malloc for instance), some parts might be avoided. Especially the string handling functions take a disproportionally large amount of time. This might be avoided by using specific functions for the cases that we use.

5.4.1 Object properties

There are two major areas where the performance of object properties might be improved compared to the current project compiler implementation: handling "static" properties (properties, whose names we know at compile time) and handling integer indices.

With regards to the static property names, an improved approach might be to calculate the hash value at compile at then use this to look up the values rather than computing the hash at run-time.

However, the current hash map library used by the project compiler (and the other libraries surveyed for this project) does not support looking up a value with a pre-computed hash. Even if it did, there is the issue of having to reimplement the hash function in Java to allow the compiler to output the correct hash value.

Another approach might be to have two maps: one for the static properties and one for the properties looked up by strings. In this case, however, we need to ensure that the both maps are updated at the correct times. This is non-trivial as we still need to support the prototype chain properly, so it is not easy to know where the property is. A property might be defined for the local object at some later point in the program, but right now it might be found in the parent object.

With regards to the integer indices, the difficulties are the same, if it is implemented using a double index. We could potentially just update both indices whenever we write and then save some time when reading, but since "assign_property" is already the single-most expensive run-time support function for the DeltaBlue benchmark, this does not look like an attractive approach.

A different approach could be to have two hash functions: one for strings and one for integer values (like the solution found in Rhino). This ensures that there is only one map and that numeric values can be used to look up properties faster. This does, however, require that we recognize strings containing integer values and convert them to integer values before accessing the properties, since the hash of the string "1234" is most likely different from the hash of the number 1234. The current hash map library does not support having two different hash functions for the same map, so this approach would require us to write a new, custom hash map for the compiler.

5.4.2 Garbage collection

Since the garbage collection also used a lot of time, this is also a place were improvements would be very beneficial. A good way to make improvements would be to allocate more of the memory on the stack instead of on the heap. The stack is a cheap place to allocate memory, and has the advantage that, when a function returns, the memory that it allocated on the stack is automatically "freed". This also means, however, that when you put something on the stack, you have to make sure that you do not, somehow, return a pointer to the object.

The project compiler puts all lists of arguments, all environments and all JavaScript objects on the heap.

5.4.2.1 Arguments

The lists (arrays) of arguments could relatively easily be put on the stack because they are made in the caller of the function and the arguments are copied out of the list at the beginning of the callee, and it is the copies that are used. Therefore there might be references to copies of the values of the list, but not to the list itself. However, in addition to the list of arguments that we make, we also make the arguments object. This is a JavaScript object like any other, so even if we put the arguments list on to the stack, the arguments object will still be on the heap. Few functions use the arguments object, however, so we could look in the function to see if it is used, and only in that case create it. We do not need to worry about inner functions, because they will have their own arguments objects.

5.4.2.2 Environments

Environments are put on the heap and contain mappings for all the variables that they can access. This means that the values of variables are also on the heap. If a variable is declared by a function, and not used in inner functions, we could translate it to a temporary value instead of a variable in the TAC. The reason is that it in that case, it does not need to be in an environment. This would mean that that value is put on the stack instead of on the heap.

In the cases where we have to make the environment, however, it is difficult to tell when the environment can be removed again because it is bound to a function object.

5.4.2.3 Javascript objects

In general, objects cannot be put on the stack because there can be references to them from many places in the code. However, if we can determine that there are no references to the object when the function returns, we could put them on the stack. If the object is returned directly from the function, it therefore cannot be put on the stack. Also, references to the object can be made by adding the object as a property to another object that was declared outside the function. If this is the case, we also cannot put it on the stack. Lastly, if the object is added as a property to another object or is used in the environment of a function, and that object/function is returned, the object cannot be put on the stack. All these things can also happen if the object is passed as an argument to a function, and in this case we also have the difficulty of determining what function it was passed on to, because of the dynamic nature of Javascript function variables.

5.4.2.4 Other complications

The stack is often smaller than the heap. If we improve our model such that we allocate memory on the stack we should be careful not to run out of stack memory. This could happen in a program that used a deep recursion, because the stack would grow for each recursive call. To solve this problem we could then fall back to allocating memory on the heap, and then free it explicitly ourselves.

5.4.3 Run-time support

The run-time support code could be improved on its own, but we need to be careful that we do not overfit the run-time support for the exact benchmarks we run. For instance, it is possible to test for special cases in the operators before running the general algorithms, but we need to ensure that this would improve the run-time performance for most JavaScript programs and not only for the benchmarks included in this project.

We could also replace some calls to general library functions with dedicated functions, we implement ourselves. For instance, rather than using "sprintf" to convert a double to a string, we might implement a function that only supports this conversion.

The calls to other string specific functions such as strlen might also be avoided if we extend our strings to include the string length along with the actual chars. Since around 3.5 % of the time in the DeltaBlue benchmark was spent in this function, this improvement makes sense, but probably mostly after addressing the garbage collection and object property issues.

Chapter 6

Discussion

6.1 Compiler Implementation

The performance of the compiler itself has shown to be very dependent on the performance of the external tools used. In particular the performance of the ANTLR 4 tool has shown to be a significant bottleneck. The benchmark also showed that GCC used a significant amount of time, but not more than could be expected.

The ANTRL 4 tool allowed the use of the JavaScript grammar described in the ECMAScript Language Specification with very few changes - this allowed faster development at the cost of significant reductions in the run-time performance of the compiler itself. This tool is therefore a prime candidate for replacement if the run-time performance of the compiler itself should be optimized.

For the Java code of the project compiler, the optimizations code is by far the most expensive to run. Optimizations are expected to be among the most expensive parts of any compiler, but the current implementation in the project compiler does not make use of any of the improvements for the data-flow algorithms that exists. The naive approach to running the data-flow algorithm is one of the most obvious places to improve the performance of the compiler itself.

6.1.1 Compiler design

On the surface the different phases of the compiler are loosely coupled and only share the data model between them. In reality, however, the phases are rather tightly coupled due to the implicit assumptions for the output of previous phases. For instance, the TAC generator has implicit assumptions about the annotations produced by the context handling, the IR generator has implicit assumptions about the exact operators left after the translation to TAC form and so on.

While it is always desirable to reduce the coupling, it would be difficult to imagine a compiler design where each phase was completely decoupled from the previous phases. To ease maintainability, however, it might be desirable to make the assumptions more explicit. This could be done at the data model level: for instance, rather than having the context handling annotate the AST directly, it could create a new annotated AST - these classes could then enforce the assumptions about the annotations at the constructor level.

The compiler design does however allow for relatively easy extensions of the compiler. If for instance, a new phase was needed to perform optimizations directly on the AST, the current design would allow the construction of this without changing any of the existing phases. The only class affected would be the Compiler class itself.

6.2 Just-in-time compilation vs Ahead-of-time compilation in Dynamic Languages

The project compared two ahead-of-time compilers and one just-in-time compiler. Significant differences in performance were found among the different implementations both when comparing the un-optimized output code and the optimized output code.

Since the just-in-time performs better for the unoptimized code, it must be due to a better compiler implementation. At startup the just-in-time compiler does not have access to anything the ahead-of-time compiler does not also have - in fact, the ahead-of-time compiler should have been at an advantage because it compiles all functions ahead of time. In short, the examined just-in-time implementation was able to produce better performing code than the two ahead-of-time implementations using less time to do so.

Optimizations are where the difference in the two compilations really show:

ahead-of-time must gain all the information it needs from static analysis whereas the just-in-time compiler can use run-time type feedback.

Just-in-time can, in theory, perform everything that the ahead-of-time can in terms of type inference, but also has access to actual run-time type feedback - however, Just-in-time is severely limited in the time it is allowed to use since the compiler runs during the execution.

The project is unable to give any definite results on the comparison of ahead-of-time compilation and just-in-time compilation in general. Based on the differences observed in performance of the un-optimized code, it would not be reasonable to generalize the differences found in the concrete implementations to differences in performance for the two compilation models.

Of the concrete implementations considered, the just-in-time compiler performed significantly better than the two ahead-of-time compilers. The just-in-time compiler did, however, also have a much more advanced memory model for objects than the ahead-of-time compilers and the benchmarks revealed the overhead from this particular subsystem to be a significant bottleneck, at least for the project compiler.

6.2.1 Improving the project implementation

In addition to a changed object memory model, there are some of the techniques used in the just-in-time that could be adapted for the ahead of time compiler: Just-in-time compilers use an optimistic approach to optimizations: when optimizing a function, it compiles for only the types it has seen before. It then adds add guards to bail out to de-optimized code, if the assumptions of types are no longer true.

An ahead-of-time compiler might follow a similar approach and compile several versions of a function for different combinations of types found. This strategy essentially makes a trade-off between the size of the binary and the run-time overhead. A different way to make the same trade-off is to implement more specific versions of the run-time support methods, that have partial type information (for instance, a addition operator, that knows that one operand never a string).

6.2.2 Testing just-in-time vs ahead-of-time compilation models

To perform a test of the just-in-time compilation model vs the ahead-of-time compilation model rather than testing the performance of concrete implementations that happen to use different compilation models, a number of changes would have to be made:

A similar model would have to implemented in ahead-of-time compilers to perform a more realistic comparison between the two compilation models. Once the performance of the unoptimized code from the two compilers compared are the same a realistic comparison of the compilation models could be performed. Even so, the end result would be very dependent on the quality of the implementation, the number of optimizations implemented. Since the compilation time overhead are paid at two different times a fair number of iterations must also be considered: if the program in question is run on the server side of a busy website, it might be invoked thousands of times per minute - and even a slight reduction in run-time might justify long compilation times. On the other hand the just-in-time compiler will always have to produce the code quickly, since it is not invoked until the user needs it.

Chapter 7

Conclusions

The project first described the characteristics of a dynamic language and showed that JavaScript fits in this definition due to its type system and programming concepts such as higher order functions and build-in data structures.

The project then showed that it is possible to compile this dynamic language ahead-of-time to a native binary, using a statically typed language as an intermediate step.

The performance of the output code from the project compiler was measured to perform significantly better than a contemporary, widely used interpreter, but the performance was lacking when compared to another ahead-of-time compiler and a just-in-time compiler.

The un-optimized output code was found to be less efficient than that of the other compilers. This result appeared to be driven largely by the inefficient modelling of object properties and garbage collection in the project compiler.

For the optimized output code, the just-in-time compiler performed several orders of magnitude better than the output from the project compiler. The project compiler's optimization improved the performance significantly for small test examples, but did not improve the performance for the larger benchmarks.

Overall the project found that compiling a dynamic language ahead-of-time is a viable alternative to interpreting the language, but was unable to make any generalizations with regards to ahead-of-time compilation compared to just-in-time compilation.

7.1 Future work

To limit the scope of the project compiler, a subset of JavaScript was selected. To make the compiler useful for real-life projects the whole JavaScript specification would have to be implemented. In addition, a native libary similar to that of RingoJs and NodeJs, that maps server specific functions would have to be implemented for the project compiler to be used in its intended setting.

7.1.1 Performance improvements

In addition, the performance of the compiler could be improved in several ways. Specifically by addressing the bottlenecks identified.

Improved memory model The memory model for objects and properties was found to be one of the major performance bottlenecks in the output code of the project compiler. To remove this bottleneck, one or more of the following strategies might be applied:

- Use a custom hash-map implementation to support numeric indicies along with string indicies and also support pre-computed hashes for properties, whose names are known at compile-time.
- Use global maps rather than object-local maps. This strategy requires a solution similar to the map transitions of V8, but would allow all object instances that contain the same fields to share a single map. This would reduce memory pressure and thereby reduce the load on the Garbage Collection subsystem, and make lookups faster.

Usage of stack We have suggested solutions for allocating memory on the stack in chapter 5. Implementing these suggestions could improve performance.

Improved JavaScript optimizations The reach of the optimizations could be improved by allowing the data-flow algorithms to share information across function calls. This requires some back and forth sharing of data: Whenever a function call node is encountered - if the function node is known at that point - the data-flow algorithm can hand the variable state down to the called function node as the initial conditions and run the algorithm there. Once that recursive call is completed, it can then continue on the other side of the function call with the output from the recursive call as the variable state. The draw-back of this approach is that the state of the variables before the function call might change, forcing a new recursive call to be made later in the optimization.

Extends the use of type information The type information collected by the Type Inference optimization might be used to allow further performance improvements. One way could be to build a framework for fully unboxed variables: If it is detected that a variable is always a single type, then it would be beneficial to use that information to skip the JavaScript value completely and store the variable as its native C type instead. The memory model of the project compiler would have to be changed to allow this type of variables in the environments though.

Another way to use the type information more could be to collect information about the different types that parameters for a function could have and then compile several versions of the function for those different type combinations. This would allow the compiler to optimize each function fully, in the same way that the just-in-time compilers do, to be useful only for the specific types of parameters considered, but still allow the types to change at run-time.

Inline caches Inline caches have shown significant performance improvements for the V8 just-in-time compiler. Implementing a version for the ahead-of-time compiler could give the same benefits to the project compiler.

7.1.2 Compile timings

We found out that the phases that used the most time when doing the compilation was the parsing, lexing and optimizations. To make the parsing and lexing faster, an obvious solution would be to try to use another tool than ANTLR.

Our implementation of the data flow algorithm could perhaps be improved by using another strategy for adding information to the statements. Our current strategy is to iterate over the statements, until nothing more can be infered. When information is added to a node, the next statement where this can be done will, however, be either the ancestors or predecessors of the node. This could perhaps be exploited by adding these nodes to a queue of the next nodes to add information to.

Appendix A

Appendix A - Benefits of Dynamic Languages

According to many authors *Dynamic Languages* offer shorter development time and at least as high quality code as *Statically typed languages* [Tra09, pp. 1-2] [Pau07, p. 13] [Ous98, p. 5] [NBD$^+$05, p. 1]. Although only a few systematic comparisons of dynamic and static progamming languages have been published, there appear to be some evidence for their claims, at least for smaller systems [Pre00, pp. 28-29] [SH11, pp. 102-104] [Han10, pp. 301-303].

The purpose of this section is to describe the primary arguments for using *Dynamic Languages* in software development and to give concrete examples of problems that are well suited for being solved using these languages.

A.1 Primary arguments for the benefits of Dynamic Languages

The arguments for the benefits of *Dynamic Languages* generally fall into two categories:

Dynamic Languages are more *expressive*

Static typing doesn't solve all type problems anyway

The term *expressiveness* in this context means the ability to express the same functionality in less code and in a more convenient way [Pau07, p. 8] [Tra09, p. 15].

Remove unnecessary details *Dynamic Languages* are attributed with removing a lot of unnecessary details. As an example the classic "Hello, world!" program is often mentioned. In Python it is a simple one liner: `print "Hello, World!"` where Java requires at least 5 lines of code (class definition, main function defintion and the call to `System.out.println`) [Tra09, p. 15].

Build-in data types The build in data types often allow the programmer to express the intent of the code more directly [Tra09, p. 15]. As an example, compare the assignment of three numbers to a dynamic array in Java and JavaScript. The JavaScript version is a single line: `var arr = [1,2,3];` whereas the Java version requires four lines.

```
ArrayList<int> arr = new ArrayList<int>();
arr.add(1);
arr.add(2);
arr.add(3);
```

Inadequacies of static type systems The other main argument given for the benefits of the *Dynamic Languages* compared to *Statically typed languages* is that the static type systems fail to solve all the type related errors - either by disallowing code, that would be valid at run-time or by allowing code that is not valid at run-time.

Valid at run-time The type systems of *Statically typed languaes* will some times reject code, that would be valid at run-time [Rus04, p. 10]. An example in Java, is the following code:

```
public int length(Object foo){
    if(foo instanceof String){
            return foo.length();
        } else {
            return -1;
        }
    }
```

The object foo, is guaranteed to be of the type `String`, when the function "length" is called, but the static type system will see foo as an instance of `Object` rather than `String` and reject the code.

Invalid at run-time The type systems of *Statically typed languaes* will under certain conditions fail to detect type related errors at compile-time [Tra09, p. 8]. An example in Java is the following code:

```
    String [] ar1 = new String[1];
    Object [] ar2 = ar1;
    ar2[0] = new Object();
```

This is legal Java code, and will compile, but will fail at run-time because the code is assigning an instance of `Object` to an array that can only contain instances of `String`.

A.2 Examples of problems suitable for Dynamic Languages

As our overview of the different languages have shown, the dynamic languages give the programmer the posibility to express solutions to problems using objects, higher order functions and metaprogramming. This means that the programmer has the possibility to express the solution in the way he finds the most convenient. To give a concrete example of this, we explored the design patterns described in "Design Pattern" book by Gamma, Helm, Johnson and Vlissides [GHJV95]. Although any pattern in this book can be implemented in any language that supports objects, using a *Dynamic Language* has some advantages. We found this to be true for most patterns in the book, but chose the decorator pattern as an example, because it clearly shows how the *Dynamic Languages* remove unnecessary details to create shorter and clearer code. In the example, we have a "window" object that we want to decorate with horizontal and vertical scroll bars. The example is given in JavaScript code.

```
function SimpleWindow()
{
    this.draw = function(){...};
    this.getDescription = function(){ return "simple window"; };
}

function HorizontalScrollbarDecorator(window){
    var addHorizontalScrollbars = function(){...};

    this.draw = function(){
        window.draw();
        addHorizontalScrollbars();
    };

    this.getDescription = function(){
            window.getDescription() + ", including horizontal scrollbars";
    };
}

function VerticalScrollbarDecorator(window){
    var addVerticalScrollbars = function(){...};

    this.draw = function(){
        window.draw();
        addVerticalScrollbars();
    };

    this.getDescription = function(){
             window.getDescription() + ", including vertical scrollbars";
    };
}

var window = new SimpleWindow();

// "decorate" the window with scroll bars
window = new VerticalScrollbarDecorator(
           new HorizontalScrollbarDecorator(window));
```

Compared to a Java implementation we can ommit an abstract super class (Window), that would define the interface (`draw` and `getDescription`) as well as an abstract decorator super class (WindowDecoratoer) that would implement the Window interface by delegating calls to a concrete implementation of Window.

A.3 Context

Besides the features of the *Dynamic Languages* themselves, the contexts that they are used in have also contributed to their popularity.

The languages also have benefits in that some of them are made to have features that are particularly suitable in certain areas. JavaScript is supported as a scripting language that can be used for making client-side scripting in all major web browsers [dev13a]. If you want scripts in other languages to run in the browsers, they must be compiled to JavaScript [goo13].

PHP and Ruby are both languages that have features suitable for sever-side webdevelopment [MAso13, Chapter: Preface] [abo13c] and are available from many webhosts.

Python has the benefit of being able to serve as a middle layer between different programs, because it can easily call functions in programs written in C, for instance [ext13].

Appendix B

Appendix B - Run-time Data

Table B.1: Project Compiler run-time benchmark score: DeltaBlue

Run #	No opt	CP	Inlining	DCE	TI
1	559,21	552,63	562,98	738,31	747,76
2	734,64	725,12	733,91	717,1	744,16
3	732,44	730,99	750	737,57	717,81
4	724,4	721,53	549,89	541,63	544,96
5	535,56	688,73	744,16	740,52	730,26
6	722,24	730,26	740,52	743,42	749,25
7	733,91	721,53	727,3	741,94	741,94
8	727,3	714,07	733,18	727,3	746,39
9	723,68	728,81	736,84	737,57	747,76
10	728,09	717,81	739,78	735,37	750,74
11	739,04	731,71	749,25	751,49	740,52
12	726,57	722,96	736,84	736,84	747,76
13	733,91	717,81	740,52	738,31	739,78
14	732,44	721,53	739,04	725,12	738,31
15	726,57	722,24	739,78	730,26	749,25
16	721,53	717,81	747,13	733,91	744,9
17	717,81	727,3	740,52	736,1	744,16
18	732,44	729,53	739,78	734,64	736,84
19	714,97	728,09	748,51	738,31	741,94
20	729,53	725,85	743,42	733,18	736,84
21	728,09	729,53	743,42	735,37	746,39
22	727,3	722,96	744,9	737,57	741,94
23	725,85	728,81	744,9	535,02	560,32
24	548,78	516,14	647,96	736,1	752,24
25	718,53	733,18	736,84	737,57	730,26
26	730,99	725,12	746,39	741,2	733,91
27	736,84	742,68	750,74	736,84	747,76
28	743,42	721,53	744,9	542,81	558,1
29	549,35	548,23	603,46	736,84	748,51
30	724,4	711,94	716,39	727,3	714,07
31	712,65	718,53	737,57	731,71	739,78
32	736,84	722,96	739,78	725,12	742,68
33	715,68	710,52	754,32	723,68	715,68
34	732,44	727,3	750	720,69	741,2
35	730,99	720,69	746,39	736,84	731,71
36	730,26	552,63	564,1	536,8	540,55
37	700,15	719,25	733,18	733,91	738,31
38	734,64	727,3	753,75	695,98	535,02
39	543,35	540,55	579,52	733,91	725,85
40	724,4	736,1	720,81	722,96	724,4
41	721,53	706,05	734,64	736,1	738,31
42	733,91	716,39	717,81	703,24	714,07
43	698,06	713,36	733,18	732,44	728,09
44	723,68	725,85	723,68	736,1	739,04
45	722,96	716,39	741,2	736,1	736,1
46	733,91	714,07	725,12	717,81	720,69
47	727,3	717,1	738,31	738,31	741,2
48	717,81	725,85	733,91	713,36	736,84
49	730,99	728,09	729,53	733,91	731,71
50	729,53	728,09	734,64	727,3	741,94

Table B.2: Project Compiler run-time benchmark time (ms): Dijkstra

Run #	No opt	CP	Inlining	DCE	TI
1	15199	15361	15377	14374	13535
2	14285	14403	15454	14143	12818
3	18025	15367	15431	14456	13752
4	14269	14555	15146	17089	13614
5	15911	14796	15522	14734	15872
6	16432	14719	14881	15327	13771
7	15837	15542	14203	15578	12957
8	15083	15668	14874	15557	13419
9	15484	15592	15205	14962	13974
10	15664	16409	16061	15076	14090
11	15450	15341	14652	14313	12966
12	14648	14904	14709	14435	13381
13	14754	14740	14764	14788	13322
14	14679	15959	15417	15124	13098
15	14389	16787	14624	15052	13375
16	15370	14875	15153	14884	13686
17	18292	15086	15200	14119	13100
18	14469	14585	14441	14490	13091
19	15931	15227	14781	17517	13202
20	15864	15769	14507	14625	14318
21	17529	14853	15789	14914	14272
22	16216	15656	15737	15494	14584
23	15819	16018	15431	15154	14166
24	15077	17851	15231	15600	13496
25	15684	15563	15678	15831	13219
26	14664	14996	14722	17726	14823
27	15645	15045	14474	14694	13309
28	15318	15294	15306	15938	13780
29	15721	14606	14681	14266	13151
30	15626	15245	14400	14342	13417
31	15550	15254	15420	14467	13014
32	15658	15580	15307	15586	13950
33	15769	15660	18178	14644	15715
34	14608	15879	14848	14909	13585
35	14658	14131	14207	15110	14015
36	14354	14267	15260	17735	13616
37	15035	15142	17448	15541	13761
38	15977	15808	15424	14468	14577
39	15279	14968	14828	14453	12874
40	14361	14968	15491	14924	13940
41	15803	16091	14492	14513	13209
42	14783	15697	15592	14325	13185
43	14768	15323	16435	14442	15045
44	14440	16136	15691	14235	16236
45	14622	14469	14826	14827	13110
46	14839	13834	13896	13741	12806
47	14501	13982	13544	13858	12381
48	13810	13579	13681	13721	12395
49	14011	13844	13679	14595	13057
50	13904	13843	13732	13546	12324

Table B.3: Project Compiler run-time benchmark time (ms): Loop

Run #	No opt	CP	Inlining	DCE	TI
1	3537	3617	3613	3420	1173
2	3534	3628	3622	3430	1169
3	3541	3689	3623	3422	1174
4	3551	3621	3632	3428	1169
5	3529	3643	3618	3412	1174
6	3580	3638	3639	3423	1174
7	3525	3635	3610	3410	1169
8	3532	3607	3610	3412	1168
9	3519	3614	3615	3415	1169
10	3525	3634	3683	3420	1168
11	3516	3631	3609	3414	1193
12	3551	3612	3615	3409	1175
13	3515	3640	3612	3410	1179
14	3543	3616	3642	3411	1168
15	3524	3612	3615	3467	1181
16	3521	3614	3622	3409	1180
17	3533	3624	3627	3420	1170
18	3554	3631	3634	3419	1186
19	3548	3643	3644	3433	1179
20	3531	3620	3619	3474	1170
21	3528	3627	3641	3413	1167
22	3523	3658	3627	3448	1179
23	3537	3636	3677	3461	1179
24	3545	3620	3637	3421	1170
25	3525	3613	3624	3420	1169
26	3532	3623	3620	3429	1170
27	3534	3671	3654	3426	1171
28	3520	3616	3635	3476	1223
29	3530	3638	3617	3410	1169
30	3533	3642	3617	3420	1168
31	3526	3622	3637	3415	1170
32	3532	3620	3618	3477	1170
33	3519	3626	3663	3552	1409
34	4183	3977	3631	3441	1172
35	3530	3706	3682	3479	1181
36	3562	3666	3636	3431	1172
37	3538	3621	3618	3439	1179
38	3546	3635	3639	3436	1174
39	3571	3641	3628	3423	1176
40	3548	3662	3644	3439	1184
41	3558	3642	3642	3434	1179
42	3542	3634	3680	3434	1177
43	3544	3629	3634	3421	1169
44	3528	3659	3644	3427	1171
45	3527	3641	3631	3421	1173
46	3538	3632	3620	3435	1174
47	3539	3615	3615	3422	1171
48	3525	3611	3612	3416	1178
49	3518	3620	3611	3412	1170
50	3523	3621	3640	3447	1173

Table B.4: Node run-time benchmark score: DeltaBlue

Run #	No Opt	IC	All opt
1	4310,894	70924,78	220887
2	4383,623	72405,82	228384,8
3	4099,316	69913,17	217270,4
4	4357,176	71308,26	221753,2
5	4304,282	73430,65	223267,3
6	4346,218	73252,13	223214,4
7	4264,611	70957,84	217686,9
8	4286,771	72432,27	221085,4
9	4330,729	72339,7	228034,4
10	4251,387	73278,58	227809,6
11	4317,505	73800,91	220014,3
12	4291,058	72498,39	221336,6
13	4363,788	74191,01	228199,7
14	4313,192	73278,58	218777,9
15	4346,218	73939,76	219353,1
16	4343,953	73556,27	225488,8
17	4326,403	71870,27	215703,4
18	4317,505	74243,9	222916,8
19	4299,982	73086,84	221045,7
20	4304,282	71632,24	219769,6
21	4310,894	72723,19	224265,6
22	4330,729	73298,41	220483,7
23	4253,745	72881,87	219207,6
24	4363,788	73714,96	221495,3
25	4319,797	71281,82	221191,2
26	4304,282	74045,55	215696,8
27	4244,776	71830,6	223816
28	4277,835	73952,98	223240,8
29	4370,4	74111,67	223921,8
30	4333,008	73529,83	222731,7
31	3054,652	54805,21	161096,5
32	4337,341	73714,96	223981,3
33	4286,771	73576,11	223822,7
34	4280,166	73840,58	221006
35	4363,788	74600,94	226745,1
36	4310,894	73073,61	220450,6
37	4257,999	73893,48	220979,6
38	4350,564	74270,35	229753,4
39	4306,587	73820,75	228960
40	4299,982	74508,37	224126,8
41	4350,564	74197,62	221105,2
42	4357,176	74581,1	228298,8
43	4310,894	74475,32	225627,7
44	4363,788	73986,04	225892,1
45	4352,823	73741,41	227194,7
46	4304,282	74508,37	226936,8
47	4297,67	73523,22	226427,7
48	4297,67	73172,79	221270,5
49	4326,403	74032,32	220325
50	4370,4	74138,11	227465,8

Table B.5: Node run-time benchmark time (ms): Dijkstra

Run #	No Opt	IC	All opt
1	7263	425	214
2	7486	428	219
3	7386	432	239
4	7450	430	231
5	7246	417	220
6	7259	419	224
7	8550	543	245
8	7687	419	222
9	7332	419	222
10	7284	421	221
11	7523	421	221
12	7189	417	221
13	7186	417	221
14	7166	425	221
15	7264	417	223
16	7399	422	221
17	7235	424	221
18	7309	419	221
19	7212	417	221
20	7943	424	221
21	7203	422	221
22	9239	421	221
23	7310	418	220
24	7159	420	222
25	7167	417	222
26	7149	423	227
27	7241	418	224
28	7357	424	224
29	7179	417	222
30	7072	538	281
31	7306	421	223
32	7070	419	221
33	7118	421	225
34	7041	420	220
35	7063	417	221
36	7185	423	220
37	7142	418	221
38	7070	417	221
39	7084	418	221
40	7143	416	221
41	7113	417	221
42	7110	424	223
43	7081	419	220
44	7094	417	221
45	7049	417	221
46	7151	425	221
47	7135	418	223
48	8160	523	279
49	7060	417	222
50	7051	419	221

Table B.6: Node run-time benchmark time (ms): Loops

Run #	No Opt	IC	All opt
1	426	425	35
2	427	427	35
3	444	435	28
4	419	429	28
5	418	419	28
6	428	419	27
7	425	422	28
8	425	421	28
9	436	421	27
10	422	421	27
11	430	422	28
12	421	422	27
13	420	421	28
14	422	421	28
15	422	422	28
16	422	421	28
17	422	421	27
18	427	421	28
19	421	424	27
20	419	424	27
21	420	419	27
22	422	440	28
23	421	420	28
24	419	421	28
25	438	425	28
26	428	427	27
27	423	423	27
28	424	429	27
29	417	421	27
30	553	555	33
31	420	423	27
32	421	423	28
33	425	426	28
34	419	418	27
35	420	419	27
36	420	422	27
37	433	419	27
38	418	419	27
39	423	424	27
40	421	421	28
41	419	424	28
42	417	427	27
43	419	421	28
44	419	424	27
45	420	421	27
46	419	418	27
47	425	427	29
48	546	546	33
49	419	421	27
50	429	419	27

Table B.7: RingoJs run-time benchmark score: DeltaBlue

Run #	Interpret	No opt	All opt
1	179,14	2923,18	3048,04
2	170,88	2865,55	2923,18
3	176,36	2924,70	2862,91
4	173,22	2816,63	1935,32
5	176,79	3219,95	2942,25
6	176,84	2796,03	2968,70
7	173,22	2816,63	3160,44
8	179,31	3021,59	2776,96
9	172,54	2862,91	3061,26
10	172,20	3100,93	2853,44
11	176,61	2743,90	2884,43
12	177,04	2929,03	2810,02
13	170,54	2968,70	2790,18
14	178,79	2291,22	2426,53
15	159,60	2283,12	2624,88
16	174,17	2902,58	2281,07
17	173,07	2048,08	1836,24
18	170,27	2538,93	2367,02
19	117,43	2824,20	2071,96
20	179,67	2895,97	2849,57
21	177,30	2910,77	3167,05
22	171,02	2677,78	2644,72
23	177,04	2968,70	2834,57
24	180,18	2893,08	2948,86
25	170,63	2810,02	2899,68
26	175,79	780,19	1935,32
27	175,28	1930,65	2854,35
28	172,03	2558,77	2810,02
29	166,52	2790,18	2279,93
30	167,18	2832,69	2208,34
31	168,62	2912,89	2624,88
32	176,87	3067,88	2850,60
33	137,68	1359,31	2380,25
34	177,45	2902,58	2902,58
35	173,91	2915,80	3160,44
36	175,28	2942,25	2935,64
37	178,52	2922,42	3100,93
38	178,00	2118,41	2426,53
39	179,23	3008,37	2981,92
40	174,95	2895,97	2988,53
41	177,82	2903,39	3451,36
42	181,68	2877,85	2823,24
43	179,14	3034,82	3180,28
44	175,71	2893,08	2671,17
45	176,58	2267,85	2866,65
46	175,28	2320,74	2697,61
47	169,70	2281,07	2909,19
48	171,39	2935,64	2783,57
49	174,50	2929,03	2882,74
50	180,79	3087,71	2869,52

Table B.8: RingoJs run-time benchmark time (ms): Dijkstra

Run #	Interpret	No opt	All opt
1	21195	8043	6517
2	20516	6564	6544
3	20852	6921	6752
4	20998	6718	6599
5	21207	6778	6657
6	21250	6605	6585
7	20964	6795	6582
8	20768	6479	6569
9	20929	6644	6578
10	21735	5573	6678
11	20438	6616	6553
12	21122	6677	5115
13	21288	6650	6895
14	21420	7121	5749
15	23635	6767	6685
16	20547	5282	8703
17	21656	6884	6784
18	20989	6670	6830
19	20775	6518	6615
20	20944	6612	6849
21	21008	6652	6558
22	20489	5908	6658
23	21266	6805	6621
24	20902	6665	6565
25	20559	6531	6764
26	20413	6854	6736
27	20545	6534	5481
28	20488	5384	6645
29	21021	6567	5396
30	20809	6601	6849
31	21856	6570	6546
32	21361	6718	6616
33	21207	5504	6585
34	21054	6591	5228
35	20850	6672	6863
36	20463	6496	5141
37	20527	6546	6663
38	21310	4928	7359
39	22187	6591	6671
40	21629	6631	6668
41	20673	5558	6615
42	21266	5588	5302
43	20762	6573	5316
44	21088	6671	6593
45	21492	6994	6917
46	20525	6613	6572
47	21076	6746	6815
48	20876	5286	6829
49	20653	5188	6793
50	21023	6662	6663

Table B.9: RingoJs run-time benchmark time (ms): Loops

Run #	Interpret	No opt	All opt
1	20284	1239	1190
2	20076	1191	1193
3	19940	1186	1200
4	20306	1189	1221
5	20184	1193	1194
6	20359	1201	1202
7	20350	1209	1218
8	20238	1239	1193
9	20690	1184	1186
10	19973	1194	1178
11	20071	1194	1201
12	20523	1211	1216
13	20189	1186	1193
14	20004	1191	1191
15	19986	1174	1185
16	20271	1184	1183
17	20118	1195	1187
18	20153	1186	1183
19	20121	1190	1194
20	20052	1250	1223
21	20351	1210	1262
22	20510	1202	1210
23	20758	1202	1200
24	20372	1201	1206
25	20161	1202	1210
26	20621	1197	1202
27	21910	1195	1195
28	20428	1202	1207
29	20615	1211	1200
30	20322	1212	1192
31	20265	1191	1216
32	20086	1194	1196
33	20105	1185	1211
34	20081	1211	1203
35	20285	1208	1202
36	21767	1205	1198
37	20136	1200	1206
38	20717	1201	1205
39	20421	1214	1200
40	20121	1188	1205
41	20360	1204	1210
42	20709	1211	1220
43	20418	1218	1223
44	20552	1260	1242
45	20905	1209	1204
46	20228	1221	1224
47	20461	1206	1203
48	20607	1196	1192
49	20456	1259	1192
50	20080	1205	1215

APPENDIX C

Appendix C - Memory usage Data

Table C.1: Project Compiler memory usage (Kb): DeltaBlue

Run #	No opt	CP	Inlining	DCE	TI
1	347264	340664	360200	355972	357036
2	351216	349632	357296	350164	352816
3	350168	350424	353332	353860	352020
4	350956	348580	361252	355704	347532
5	351484	342772	357556	353328	354924

Table C.2: Project Compiler memory usage (Kb): Dijkstra

Run #	No opt	CP	Inlining	DCE	TI
1	86960	78512	86960	86964	78512
2	86960	86960	86960	78516	78516
3	78516	86960	86960	78512	86956
4	86960	86956	86956	86964	86964
5	86960	86960	86960	78512	86964

Table C.3: Project Compiler memory usage (Kb): Loop

Run #	No opt	CP	Inlining	DCE	TI
1	896	888	896	888	892
2	900	892	896	892	892
3	900	892	892	892	888
4	896	892	892	896	892
5	900	892	888	896	896

Table C.4: NodeJs memory usage (Kb): Empty

Run #	No opt	IC	All opt
1	1400	5296	5064
2	4540	5036	4780
3	4960	5296	5316
4	5220	5296	5052
5	5220	5036	5056

Table C.5: NodeJs memory usage (Kb): DeltaBlue

Run #	No opt	IC	All opt
1	10076	10276	13876
2	10072	10276	13876
3	10076	10280	13876
4	10072	10276	13884
5	10072	10276	13880

Table C.6: NodeJs memory usage (Kb): Dijkstra

Run #	No opt	IC	All opt
1	11300	11456	12100
2	11300	11452	12356
3	11300	11452	12356
4	11300	11452	12356
5	11304	11456	12356

Table C.7: NodeJs memory usage (Kb): Loops

Run #	No opt	IC	All opt
1	9336	9460	5052
2	9788	9940	5052
3	9788	9460	4776
4	9524	9460	5320
5	9784	9944	5064

Table C.8: RingoJs memory usage (Kb): Empty

Run #	Interpret	No opt	All opt
1	39340	43736	43396
2	40088	43996	43720
3	39744	43736	43772
4	40008	43692	43712
5	39084	43480	43788

Table C.9: RingoJs memory usage (Kb): DeltaBlue

Run #	Interpret	No opt	All opt
1	68964	79192	79448
2	69908	82172	81940
3	69488	78484	82420
4	70368	79064	81972
5	69244	78540	82396

Table C.10: RingoJs memory usage (Kb): Dijkstra

Run #	Interpret	No opt	All opt
1	168020	85916	86672
2	165212	86888	89768
3	168372	86564	89948
4	169900	88468	88576
5	167700	91172	86784

Table C.11: RingoJs memory usage (Kb): Loops

Run #	Interpret	No opt	All opt
1	130984	58028	56684
2	134464	57872	59576
3	131100	57056	57608
4	133944	57744	58540
5	132364	59220	57700

Appendix D

Appendix D - Compile time data

Table D.1: Project Compiler - compile timings (seconds): DeltaBlue

	No opt	CP	Inlining	DCE	TI
Total	35,04	39,13	39,44	48,75	51,16
In GCC	3,65	3,71	3,69	3,71	4,30
In Antlr	28,86	29,02	29,31	29,59	28,41
Our code	2,53	6,41	6,43	15,45	18,45

Table D.2: Project Compiler - compile timings (seconds): Dijkstra

	No opt	CP	Inlining	DCE	TI
Total	8,82	9,96	10,70	13,39	14,36
In GCC	1,72	2,25	1,65	1,64	1,65
In Antlr	6,41	4,86	6,75	6,28	6,05
Our code	0,69	2,84	2,30	5,47	6,67

Table D.3: Project Compiler - compile timings (seconds): Loops

	No opt	CP	Inlining	DCE	TI
Total	6,21	6,41	6,52	6,71	7,71
In GCC	1,34	1,35	1,34	1,34	1,34
In Antlr	4,11	4,11	4,22	4,07	4,38
Our code	0,77	0,95	0,96	1,30	1,99

Table D.4: NodeJs startup overhead (seconds)

	No Opt	IC	All opt
Empty	0,045	0,045	0,046
DeltaBlue	0,048	0,052	0,051
Dijkstra	0,129	0,064	0,068
Loops	0,047	0,049	0,052

Table D.5: RingoJs startup overhead (seconds)

	Interpret	No opt	All opt
Empty	1,971	1,005	1,005
DeltaBlue	2,933	1,233	1,234
Dijkstra	3,944	3,462	3,461
Loops	3,434	1,046	1,057

Appendix E

Appendix E - Profiling data

Table E.1: Project Compiler - Profiler output - DeltaBlue (no opt)

Function	**Type**	**Time (self)**
GC_mark_from	GC	20,23 %
assign_property	Runtime support	12,38 %
get_own_property	Runtime support	8,37 %
GC_reclaim_clear	GC	6,92 %
GC_malloc	GC	6,49 %
_int_malloc	GC	4,71 %
GC_mark_and_push_stack	GC	4,02 %
__strelen_sse42	Native lib	3,44 %
memcpy	Native lib	2,65 %
vprintf	Native lib	2,45 %
GC_push_all_eager	GC	2,27 %
GC_header_cache_miss	GC	1,79 %
(unknown name)	Native lib	1,50 %
get_property	Runtime support	1,93 %
create_object_value	Runtime support	1,30 %
malloc	GC	1,24 %
GC_apply_to_all_blocks	GC	1,21 %
__memcpl_sse4_1	Native lib	0,92 %
GC_build_fl	GC	0,91 %

Table E.1: Project Compiler - Profiler output - DeltaBlue (no opt)

Function	Type	Time (self)
create_arguments_list	Runtime support	0,89 %
assign_variable	Runtime support	0,80 %
__GL_memset	Native lib	0,78 %
_IO_default_xsputn	Native lib	0,73 %
GC_allochblk_nth	GC	0,67 %
create_arguments	Runtime support	0,59 %
GC_find_header	GC	0,58 %
call_function	Runtime support	0,55 %
GC_add_to_black_list_stack	GC	0,55 %
copy_environment	Runtime support	0,51 %
strchrnul	Native lib	0,40 %
GC_next_used_block	GC	0,36 %
value_at	Runtime support	0,32 %
GC_generic_malloc_many	GC	0,32 %
vsprintf	Native lib	0,31 %
_IO_str_init_static_internal	Native lib	0,30 %
jsfunc_71	JS	0,29 %
GC_allochblk	GC	0,23 %
to_number	Runtime support	0,23 %
relational_comparison_op	Runtime support	0,21 %
GC_reclaim_block	GC	0,19 %
_itoa_word	Native lib	0,19 %
GC_set_fl_marks	GC	0,18 %
_IO_old_init	Native lib	0,18 %
__strtol_l_internal	Native lib	0,18 %
_IO_no_init	Native lib	0,17 %
to_primitive	Runtime support	0,17 %
addition_op	Runtime support	0,16 %
to_boolean	Runtime support	0,15 %
_IO_setb	Native lib	0,15 %
sprintf	Native lib	0,15 %
GC_clear_hdr_marks	GC	0,14 %
jsfunc_6	JS	0,13 %
jsfunc_7	JS	0,12 %
free	GC	0,12 %
GC_finish_collection	GC	0,11 %
jsfunc_69	JS	0,11 %
jsfunc_70	JS	0,11 %
jsfunc_72_chain_test	JS	0,10 %
GC_freehblk	GC	0,10 %
GC_reclaim_generic	GC	0,09 %

Table E.1: Project Compiler - Profiler output - DeltaBlue (no opt)

Function	**Type**	**Time (self)**
equality_comparison_op	Runtime support	0,09 %
GC_clear_stack	GC	0,08 %
get_argument	Runtime support	0,08 %
jsfunc_66	JS	0,08 %
get_index	GC	0,08 %
to_string	Runtime support	0,07 %
declare_variable	Runtime support	0,07 %
GC_clear_stack_inner	GC	0,07 %
jsfunc_42	JS	0,07 %
GC_number_stack_black_listed	GC	0,07 %
GC_start_world	GC	0,06 %
GC_mark_thread_local_free	GC	0,06 %
GC_push_next_marked_uncollectable	GC	0,06 %
GC_push_all_stacks	GC	0,06 %
GC_suspend_all	GC	0,06 %
setup_header	GC	0,06 %
GC_add_to_fl	GC	0,05 %
jsfunc_41	JS	0,05 %
GC_remove_from_fl	GC	0,05 %
GC_free_block_ending_at	GC	0,05 %
jsfunc_65	JS	0,05 %
clear_marks_for_block	GC	0,04 %
GC_install_counts	GC	0,04 %
jsfunc_54	JS	0,04 %
ceil	Native lib	0,04 %
GC_remove_protection	GC	0,04 %
GC_prev_block	GC	0,04 %
jsfunc_51	JS	0,04 %
GC_is_black_listed	GC	0,04 %
GC_approx_sp	GC	0,04 %
GC_mark_some	GC	0,03 %
GC_get_first_part	GC	0,03 %
jsfunc_74_change	JS	0,03 %
GC_register_dynlib_callback	GC	0,03 %
jsfunc_5	JS	0,03 %
Array_push	Runtime support	0,02 %
Unknown	Unknown	0,75 %

Table E.2: Project Compiler - Profiler output - DeltaBlue (opt)

Function	Type	Time (self)
GC_mark_from	GC	20,32 %
assign_property	Runtime support	12,58 %
get_own_property	Runtime support	8,51 %
GC_reclaim_clear	GC	6,91 %
GC_malloc	GC	6,60 %
_int_malloc	GC	4,78 %
GC_mark_and_push_stack	GC	4,24 %
__strelen_sse42	Native lib	3,53 %
memcpy	Native lib	2,69 %
vprintf	Native lib	2,49 %
GC_push_all_eager	GC	2,23 %
get_property	Runtime support	1,96 %
GC_header_cache_miss	GC	1,95 %
(unknown name)	Native lib	1,52 %
create_object_value	Runtime support	1,32 %
malloc	GC	1,26 %
GC_build_fl	GC	0,95 %
__memcpl_sse4_1	Native lib	0,94 %
create_arguments_object	Runtime support	0,90 %
assign_variable	Runtime support	0,81 %
__GL_memset	Native lib	0,78 %
_IO_default_xsputn	Native lib	0,74 %
GC_apply_to_all_blocks	GC	0,65 %
GC_find_header	GC	0,64 %
GC_add_to_black_list_stack	GC	0,64 %
create_arguments	Runtime support	0,60 %
call_function	Runtime support	0,56 %
GC_allochblk_nth	GC	0,55 %
copy_environment	Runtime support	0,52 %
strchrnul	Native lib	0,40 %
GC_generic_malloc_many	GC	0,32 %
vsprintf	Native lib	0,32 %
_IO_str_init_static_internal	Native lib	0,31 %
value_at	Runtime support	0,30 %
jsfunc_71	JS	0,25 %
GC_allochblk	GC	0,19 %
GC_reclaim_block	GC	0,19 %
_itoa_word	Native lib	0,19 %
GC_set_fl_marks	GC	0,19 %
_IO_old_init	Native lib	0,18 %
__strtol_l_internal	Native lib	0,18 %

Table E.2: Project Compiler - Profiler output - DeltaBlue (opt)

Function	Type	Time (self)
_IO_no_init	Native lib	0,18 %
_IO_setb	Native lib	0,18 %
GC_next_used_block	GC	0,17 %
relational_comparison_op	Runtime support	0,15 %
sprintf	Native lib	0,15 %
jsfunc_6	JS	0,13 %
jsfunc_7	JS	0,12 %
free	GC	0,12 %
GC_finish_collection	GC	0,12 %
to_number	Runtime support	0,11 %
jsfunc_69	JS	0,11 %
jsfunc_70	JS	0,11 %
GC_freehblk	GC	0,10 %
GC_reclaim_generic	GC	0,09 %
equality_comparison_op	Runtime support	0,09 %
jsfunc_72_chain_test	JS	0,08 %
GC_clear_stack	GC	0,08 %
get_argument	Runtime support	0,08 %
get_index	GC	0,08 %
to_primitive	Runtime support	0,07 %
jsfunc_66	JS	0,07 %
to_string	Runtime support	0,07 %
declare_variable	Runtime support	0,07 %
GC_clear_stack_inner	GC	0,07 %
GC_number_stack_black_listed	GC	0,07 %
GC_start_world	GC	0,07 %
GC_mark_thread_local_free	GC	0,07 %
jsfunc_42	JS	0,06 %
GC_push_next_marked_uncollectable	GC	0,06 %
GC_push_all_stacks	GC	0,06 %
GC_suspend_all	GC	0,06 %
setup_header	GC	0,06 %
GC_add_to_fl	GC	0,06 %
jsfunc_41	JS	0,05 %
GC_remove_from_fl	GC	0,05 %
GC_free_block_ending_at	GC	0,05 %
clear_marks_for_block	GC	0,05 %
GC_install_counts	GC	0,05 %
to_boolean	Runtime support	0,04 %
jsfunc_65	JS	0,04 %
jsfunc_54	JS	0,04 %

Table E.2: Project Compiler - Profiler output - DeltaBlue (opt)

Function	Type	Time (self)
ceil	Native lib	0,04 %
GC_remove_protection	GC	0,04 %
jsfunc_51	JS	0,04 %
GC_is_black_listed	GC	0,04 %
GC_approx_sp	GC	0,04 %
GC_mark_some	GC	0,03 %
GC_get_first_part	GC	0,03 %
jsfunc_74_change	JS	0,03 %
GC_register_dynlib_callback	GC	0,03 %
jsfunc_5	JS	0,03 %
GC_start_reclaim	GC	0,03 %
jsfunc_64	JS	0,03 %
Array_push	Runtime support	0,02 %
jsfunc_62	JS	0,02 %
GC_add_map_entry	GC	0,02 %
addition_op	Runtime support	0 %
Unknown	Unknown	0,80 %

Table E.3: Project Compiler - Profiler output - Dijkstra (no opt)

Function	Type	Time (self)
get_own_property	Runtime support	32,98 %
vfprintf	Native lib	11,15 %
jsfunc_5_Dijkstra	JS	6,59 %
get_property	Runtime support	5,57 %
__memcmp_sse4_1	Native lib	5,09 %
__strlen_sse42	Native lib	3,55 %
_IO_default_xsputn	Native lib	3,3 %
GC_mark_from	GC	2,83 %
value_at	Runtime support	2,8 %
to_number	Runtime support	2,62 %
relational_comparison_op	Runtime support	2,29 %
addition_op	Runtime support	2,15 %
to_primitive	Runtime support	2,05 %
strchrnul	Native lib	1,66 %
GC_malloc	GC	1,48 %
_itoa_word	Native lib	1,45 %
assign_variable	Runtime support	1,44 %
vsprintf	Native lib	1,3 %
_IO_str_init_static_internal	Native lib	1,26 %

Table E.3: Project Compiler - Profiler output - Dijkstra (no opt)

Function	Type	Time (self)
to_string	Runtime support	1,23 %
to_boolean	Runtime support	1,2 %
_IO_old_init	Native lib	0,76 %
_IO_no_init	Native lib	0,72 %
_IO_setb	Native lib	0,61 %
sprintf	Native lib	0,61 %
GC_header_cache_miss	GC	0,57 %
ceil	Native lib	0,51 %
free	GC	0,51 %
equality_comparison_op	Runtime support	0,34 %
GC_allochblk_nth	GC	0,28 %
GC_build_fl	GC	0,15 %
assign_property	Runtime support	0,14 %
GC_add_to_black_list_stack	GC	0,13 %
GC_reclaim_clear	GC	0,12 %
create_numeric_value	Runtime support	0,12 %
GC_find_header	GC	0,1 %
GC_allochblk	GC	0,09 %
__GI_memset	GC	0,04 %
GC_apply_to_all_blocks	GC	0,02 %
GC_generic_malloc_many	GC	0,02 %
_int_malloc	GC	0,02 %
GC_reclaim_block	GC	0,01 %
GC_clear_hdr_marks	GC	0,01 %
GC_freehblk	GC	0,01 %
get_index	GC	0,01 %
____strtol_l_internal	Native lib	0,01 %
0x000000000012a780	Native lib	0,01 %
GC_add_to_fl	GC	0,01 %
GC_remove_from_fl	GC	0,01 %
setup_header	GC	0,01 %
GC_number_stack_black_listed	GC	0,01 %

Table E.4: Project Compiler - Profiler output - Dijkstra (opt)

Function	Type	Time (self)
get_own_property	Runtime support	36,03 %
vfprintf	Native lib	12,19 %
jsfunc_5_Dijkstra	JS	6,88 %
get_property	Runtime support	6,09 %

Table E.4: Project Compiler - Profiler output - Dijkstra (opt)

Function	Type	Time (self)
__memcmp_sse4_1	Native lib	5,56 %
__strlen_sse42	Native lib	3,88 %
_IO_default_xsputn	Native lib	3,61 %
GC_mark_from	GC	3,09 %
value_at	Runtime support	2,75 %
strchrnul	Native lib	1,81 %
GC_malloc	GC	1,61 %
_itoa_word	Native lib	1,58 %
assign_variable	Runtime support	1,58 %
vsprintf	Native lib	1,42 %
_IO_str_init_static_internal	Native lib	1,38 %
to_string	Runtime support	1,34 %
to_number	Runtime support	1,32 %
_IO_old_init	Native lib	0,83 %
_IO_no_init	Native lib	0,79 %
_IO_setb	Native lib	0,67 %
sprintf	Native lib	0,67 %
GC_header_cache_miss	GC	0,62 %
ceil	Native lib	0,55 %
free	GC	0,55 %
to_boolean	Runtime support	0,44 %
equality_comparison_op	Runtime support	0,37 %
relational_comparison_op	Runtime support	0,35 %
GC_allochblk_nth	GC	0,34 %
create_boolean_value	Runtime support	0,24 %
create_numeric_value	Runtime support	0,19 %
to_primitive	Runtime support	0,17 %
GC_build_fl	GC	0,16 %
assign_property	Runtime support	0,16 %
GC_add_to_black_list_stack	GC	0,15 %
GC_reclaim_clear	GC	0,13 %
GC_allochblk	GC	0,11 %
GC_find_header	GC	0,1 %
__GI_memset	GC	0,04 %
GC_apply_to_all_blocks	GC	0,02 %
GC_generic_malloc_many	GC	0,02 %
_int_malloc	GC	0,02 %
GC_reclaim_block	GC	0,02 %
GC_clear_hdr_marks	GC	0,01 %
GC_freehblk	GC	0,01 %
get_index	GC	0,01 %

Table E.4: Project Compiler - Profiler output - Dijkstra (opt)

Function	**Type**	**Time (self)**
____strtol_l_internal	Native lib	0,01 %
0x000000000012a780	Native lib	0,01 %
GC_add_to_fl	GC	0,01 %
GC_remove_from_fl	GC	0,01 %
setup_header	GC	0,01 %
GC_number_stack_black_listed	GC	0,01 %
GC_push_next_marked_uncollectable	GC	0,01 %
GC_free_block_ending_at	GC	0,01 %

Table E.5: Project Compiler - Profiler output - Loops (no opt)

Function	**Type**	**Time (self)**
jsfunc_0_additionLoop <cycle 1>	JS	20,65 %
addition_op <cycle 1>	Runtime support	18,11 %
to_number <cycle 1>	Runtime support	15,22 %
to_primitive	Runtime support	11,96 %
assign_variable	Runtime support	9,06 %
value_at	Runtime support	9,06 %
relational_comparison_op <cycle 1>	Runtime support	8,51 %
to_boolean	Runtime support	5,07 %
create_numeric_value	Runtime support	1,45 %
create_boolean_value	Runtime support	0,91 %

Table E.6: Project Compiler - Profiler output - Loops (opt)

Function	**Type**	**Time (self)**
jsfunc_0_additionLoop	JS	40,45 %
assign_variable	Runtime support	28,89 %
value_at	Runtime support	23,11 %
create_numeric_value	Runtime support	4,62 %
create_boolean_value	Runtime support	2,89 %

Appendix F

Appendix F - NodeJs data

F.1 NodeJs optimization flag scores

The table contains the score for the Delta Blue benchmark suite for each of the optimization flags considered. Each flag is run both with no optimizations enabled and with the "inline cache" optimization enabled. The Impr. collumns show the relative improvement in score compared to the baseline score and the baseline score with "use_ic" respectively.

Table F.1: Relative effect of NodeJS optimization flags for DeltaBlue

	Score	**Impr.**	**Score with "use_ic"**	**Impr.**
Baseline (No opt)	3332		96188	
packed arrays	3299	-1%	94204	-2%
smi only arrays	3219	-3%	96452	0%
clever optimizations	3266	-2%	96651	0%
unbox double arrays	3312	-1%	94528	-2%
string slices	3299	-1%	95176	-1%
use range	3259	-2%	95097	-1%
eliminate dead phis	3269	-2%	94936	-1%
use gvn	3319	0%	97583	1%
use canonicalizing	3312	-1%	96400	0%
use inlining	3286	-1%	106615	11%
loop invariant code motion	3319	0%	95884	0%
deoptimize uncommon cases	3279	-2%	95950	0%
polymorphic inlining	3272	-2%	94628	-2%
use osr	3325	0%	94916	-1%
array bounds checks elimination	3272	-2%	93906	-2%
array index dehoisting	3325	0%	93490	-3%
dead code elimination	3312	-1%	96320	0%
optimize closures	3325	0%	95705	-1%
cache optimized code	3319	0%	96003	0%
inline construct	3272	-2%	95487	-1%
inline arguments	3292	-1%	95838	0%
inline accessors	3312	-1%	96757	1%
inline new	3332	0%	127997	33%
optimize for in	3233	-3%	94733	-2%
opt safe uint32 operations	3266	-2%	96466	0%
use ic	97014	2812%		

F.2 NodeJs Garbage Collection data for DeltaBlue

This is the output from NodeJs when running the DeltaBlue benchmark with no optimizations enabled and `--trace-gc` enabled. The total running time is the "real time" component from the "time" command.

```
total time spent: 2724ms

[31224]      737 ms: Scavenge 1.9 (19.1) -> 1.3 (20.1) MB, 33 ms
[31224]     1428 ms: Scavenge 2.1 (20.1) -> 1.3 (20.1) MB, 21 ms
[31224]     2218 ms: Scavenge 2.3 (20.1) -> 1.3 (20.1) MB, 0 ms
```

The percentage used in Garbage Collection is therefore
$\frac{33\text{ms}+21\text{ms}+0\text{ms}}{2724\text{ms}} = 0.019823789 = 1.9\%$

F.3 NodeJs profiling data for the DeltaBlue benchmark

The profiling data for NodeJs was collected with no optimizations eneabled. NodeJs was executed with the "–prof" output command and the results were parsed using the V8 "tick-processor" script.

```
Statistical profiling result (2365 ticks, 0 unaccounted, 0 excluded).

[Shared libraries]:
  ticks  total  nonlib   name
  1393   58.9%    0.0%  node-v0.10.10/out/Release/node
   649   27.4%    0.0%  b774d000-b774e000
   133    5.6%    0.0%  /lib/i386-linux-gnu/libc-2.15.so
    18    0.8%    0.0%  /lib/i386-linux-gnu/libpthread-2.15.so
     3    0.1%    0.0%  /lib/i386-linux-gnu/ld-2.15.so
     1    0.0%    0.0%  /lib/i386-linux-gnu/libm-2.15.so


[JavaScript]:
  ticks  total  nonlib   name
```

```
     55    2.3%   32.7%  Builtin: A builtin from the snapshot
     49    2.1%   29.2%  Builtin: A builtin from the snapshot {1}
     27    1.1%   16.1%  Stub: CEntryStub
      8    0.3%    4.8%  LazyCompile: Plan.execute
      4    0.2%    2.4%  LazyCompile: Plan.size
      4    0.2%    2.4%  LazyCompile: OrderedCollection.size
      3    0.1%    1.8%  Stub: StringAddStub
      3    0.1%    1.8%  LazyCompile: chainTest
      3    0.1%    1.8%  LazyCompile: BinaryConstraint.isSatisfied
      2    0.1%    1.2%  Stub: FastNewClosureStub
      1    0.0%    0.6%  Stub: CompareICStub
      1    0.0%    0.6%  RegExp: %[sdj%]
      1    0.0%    0.6%  LazyCompile: Strength.weakestOf
      1    0.0%    0.6%  LazyCompile: Planner.addPropagate
      1    0.0%    0.6%  LazyCompile: Plan.constraintAt
      1    0.0%    0.6%  LazyCompile: OrderedCollection.remove
      1    0.0%    0.6%  LazyCompile: OrderedCollection.at
      1    0.0%    0.6%  LazyCompile: EditConstraint.isInput
      1    0.0%    0.6%  LazyCompile: BinaryConstraint.input
      1    0.0%    0.6%  Builtin: A builtin from the snapshot {2}


 [C++]:
   ticks  total  nonlib   name

 [GC]:
   ticks  total  nonlib   name
     61    2.6%


[Bottom up (heavy) profile]:
  Note: percentage shows a share of a particular caller in the total
  amount of its parent calls.
  Callers occupying less than 2.0% are not shown.


     ticks parent  name
   1393   58.9%  node-v0.10.10/out/Release/node
    271   19.5%    LazyCompile: NativeModule.compile node.js:887
    271  100.0%      LazyCompile: NativeModule.require node.js:842
     58   21.4%        LazyCompile: tryFile module.js:138
     58  100.0%          LazyCompile: Module._findPath module.js:160
     58  100.0%            LazyCompile: Module._resolveFilename module.js:323
     50   18.5%        Function: <anonymous> stream.js:1
     50  100.0%          LazyCompile: NativeModule.compile node.js:887
  50  100.0%            LazyCompile: NativeModule.require node.js:842
  43   15.9%        LazyCompile: startup.globalVariables node.js:160
  43  100.0%          LazyCompile: startup node.js:30
  43  100.0%            Function: <anonymous> node.js:27
  22    8.1%        Function: <anonymous> tty.js:1
  22  100.0%          LazyCompile: NativeModule.compile node.js:887
  22  100.0%            LazyCompile: NativeModule.require node.js:842
  21    7.7%        LazyCompile: startup node.js:30
  21  100.0%          Function: <anonymous> node.js:27
  17    6.3%        Function: <anonymous> buffer.js:1
  17  100.0%          LazyCompile: NativeModule.compile node.js:887
  17  100.0%            LazyCompile: NativeModule.require node.js:842
  17    6.3%        Function: <anonymous> assert.js:1
  17  100.0%          LazyCompile: NativeModule.compile node.js:887
  17  100.0%            LazyCompile: NativeModule.require node.js:842
  16    5.9%        Function: <anonymous> net.js:1
  16  100.0%          LazyCompile: NativeModule.compile node.js:887
  16  100.0%            LazyCompile: NativeModule.require node.js:842
   8    3.0%        LazyCompile: startup.resolveArgv0 node.js:809
   8  100.0%          LazyCompile: startup node.js:30
   8  100.0%            Function: <anonymous> node.js:27
  60    4.3%    LazyCompile: Plan.size
  59   98.3%      LazyCompile: Plan.execute
  46   78.0%        LazyCompile: chainTest
  46  100.0%          LazyCompile: deltaBlue
  46  100.0%            LazyCompile: Measure
  13   22.0%        LazyCompile: change
  13  100.0%          LazyCompile: projectionTest
  13  100.0%            LazyCompile: deltaBlue
  59    4.2%    LazyCompile: Plan.execute
  33   55.9%      LazyCompile: chainTest
  33  100.0%        LazyCompile: deltaBlue
  33  100.0%          LazyCompile: Measure
  33  100.0%            LazyCompile: BenchmarkSuite.RunSingleBenchmark
  24   40.7%      LazyCompile: change
  24  100.0%        LazyCompile: projectionTest
  24  100.0%          LazyCompile: deltaBlue
  24  100.0%            LazyCompile: Measure
   2    3.4%      LazyCompile: Plan.execute
   2  100.0%        LazyCompile: chainTest
   2  100.0%          LazyCompile: deltaBlue
   2  100.0%            LazyCompile: Measure
  47    3.4%    LazyCompile: OrderedCollection.size
  28   59.6%      LazyCompile: Plan.size
```

```
     28  100.0%        LazyCompile: Plan.execute
     20   71.4%          LazyCompile: chainTest
     20  100.0%            LazyCompile: deltaBlue
      8   28.6%          LazyCompile: change
      8  100.0%            LazyCompile: projectionTest
      7   14.9%      LazyCompile: Planner.removePropagateFrom
      7  100.0%        LazyCompile: Planner.incrementalRemove
      7  100.0%          LazyCompile: Constraint.destroyConstraint
      7  100.0%            LazyCompile: change
      3    6.4%      LazyCompile: Planner.makePlan
      3  100.0%        LazyCompile: Planner.extractPlanFromConstraints
      3  100.0%          LazyCompile: change
      3  100.0%            LazyCompile: projectionTest
      3    6.4%      LazyCompile: Planner.addPropagate
      3  100.0%        LazyCompile: Constraint.satisfy
      3  100.0%          LazyCompile: Planner.incrementalAdd
      3  100.0%            LazyCompile: Constraint.addConstraint
      3    6.4%      LazyCompile: Planner.addConstraintsConsumingTo
      3  100.0%        LazyCompile: Planner.makePlan
      3  100.0%          LazyCompile: Planner.extractPlanFromConstraints
      3  100.0%            LazyCompile: change
      1    2.1%      LazyCompile: Planner.incrementalRemove
      1  100.0%        LazyCompile: Constraint.destroyConstraint
      1  100.0%          LazyCompile: change
      1  100.0%            LazyCompile: projectionTest
      1    2.1%      LazyCompile: Plan.execute
      1  100.0%        LazyCompile: change
      1  100.0%          LazyCompile: projectionTest
      1  100.0%            LazyCompile: deltaBlue
      1    2.1%      LazyCompile: OrderedCollection.size
      1  100.0%        LazyCompile: Plan.size
      1  100.0%          LazyCompile: Plan.execute
      1  100.0%            LazyCompile: chainTest
     45    3.2%    LazyCompile: Module._compile module.js:374
     45  100.0%      LazyCompile: Module._extensions..js module.js:472
     45  100.0%        LazyCompile: Module.load module.js:346
     45  100.0%          LazyCompile: Module._load module.js:275
     45  100.0%            LazyCompile: Module.runMain module.js:495
     44    3.2%    Function: <anonymous> buffer.js:1
     44  100.0%      LazyCompile: NativeModule.compile node.js:887
     44  100.0%        LazyCompile: NativeModule.require node.js:842
     44  100.0%          LazyCompile: startup.globalVariables node.js:160
     44  100.0%            LazyCompile: startup node.js:30
     39    2.8%    LazyCompile: ScaleConstraint.execute
```

```
  29   74.4%      LazyCompile: Plan.execute
  29  100.0%        LazyCompile: change
  29  100.0%          LazyCompile: projectionTest
  29  100.0%            LazyCompile: deltaBlue
   7   17.9%      LazyCompile: ScaleConstraint.recalculate
   6   85.7%        LazyCompile: Planner.addPropagate
   6  100.0%          LazyCompile: Constraint.satisfy
   6  100.0%            LazyCompile: Planner.incrementalAdd
   1   14.3%        LazyCompile: Planner.removePropagateFrom
   1  100.0%          LazyCompile: Planner.incrementalRemove
   1  100.0%            LazyCompile: Constraint.destroyConstraint
   3    7.7%      LazyCompile: ScaleConstraint.execute
   2   66.7%        LazyCompile: ScaleConstraint.recalculate
   2  100.0%          LazyCompile: Planner.removePropagateFrom
   2  100.0%            LazyCompile: Planner.incrementalRemove
   1   33.3%        LazyCompile: Plan.execute
   1  100.0%          LazyCompile: change
   1  100.0%            LazyCompile: projectionTest
  38    2.7%    LazyCompile: Plan.constraintAt
  38  100.0%      LazyCompile: Plan.execute
  25   65.8%        LazyCompile: chainTest
  25  100.0%          LazyCompile: deltaBlue
  25  100.0%            LazyCompile: Measure
  13   34.2%        LazyCompile: change
  13  100.0%          LazyCompile: projectionTest
  13  100.0%            LazyCompile: deltaBlue
  38    2.7%    LazyCompile: EqualityConstraint.execute
  38  100.0%      LazyCompile: Plan.execute
  38  100.0%        LazyCompile: chainTest
  38  100.0%          LazyCompile: deltaBlue
  38  100.0%            LazyCompile: Measure
  34    2.4%    LazyCompile: chainTest
  34  100.0%      LazyCompile: deltaBlue
  34  100.0%        LazyCompile: Measure
  34  100.0%          LazyCompile: BenchmarkSuite.RunSingleBenchmark
  34  100.0%            LazyCompile: RunNextBenchmark


 649   27.4%  b774d000-b774e000
  76   11.7%    LazyCompile: <anonymous> native date.js:145
  76  100.0%      LazyCompile: Measure
  76  100.0%        LazyCompile: BenchmarkSuite.RunSingleBenchmark
  76  100.0%          LazyCompile: RunNextBenchmark
  44   57.9%            LazyCompile: RunNextBenchmark
```

```
      32   42.1%           LazyCompile: RunStep
      24    3.7%    LazyCompile: startup node.js:30
      24  100.0%      Function: <anonymous> node.js:27
      20    3.1%    LazyCompile: Constraint.satisfy
      20  100.0%      LazyCompile: Planner.incrementalAdd
      20  100.0%        LazyCompile: Constraint.addConstraint
      12   60.0%          LazyCompile: UnaryConstraint
      10   83.3%            LazyCompile: StayConstraint
       2   16.7%            LazyCompile: EditConstraint
       8   40.0%          LazyCompile: BinaryConstraint
       8  100.0%            LazyCompile: EqualityConstraint
      15    2.3%    LazyCompile: chainTest
      15  100.0%      LazyCompile: deltaBlue
      15  100.0%        LazyCompile: Measure
      15  100.0%          LazyCompile: BenchmarkSuite.RunSingleBenchmark
      15  100.0%            LazyCompile: RunNextBenchmark
      13    2.0%    LazyCompile: Module._extensions..js module.js:472
      13  100.0%      LazyCompile: Module.load module.js:346
      13  100.0%        LazyCompile: Module._load module.js:275
      13  100.0%          LazyCompile: Module.runMain module.js:495
      13  100.0%            LazyCompile: startup node.js:30
      13    2.0%    LazyCompile: DefineObjectProperty native v8natives.js:695
      13  100.0%      LazyCompile: DefineOwnProperty native v8natives.js:924
      13  100.0%        LazyCompile: defineProperties native v8natives.js:1102
      13  100.0%          LazyCompile: create native v8natives.js:1038
       9   69.2%            LazyCompile: startup node.js:30
       4   30.8%            LazyCompile: exports.inherits util.js:553
```

F.4 NodeJs assembly output

The following assembly output was generated by running NodeJs with the `--print-code` and `--code-comments` flags for a simple for loop program. All optimizations were disabled and `--nolazy` was used to force V8 to output the instructions for the operations rather than stubs call to the JIT compiler.

The program source code:

```
for(var a = 0; a < 10; a++){
    console.log(a);
}
```

The output code, as produced by the V8 disassembler. The comments prefix with a single ";" are our comments - the ";;" comments are produced by the V8 system.

The code shows that the `a++` operation is implemented directly as an assembler operation, if it is represented as a small integer, and otherwise it is converted.

```
--- Code ---
kind = FUNCTION
Instructions (size = 232)
            ; Function header
0x2f315000     0  55             push ebp
0x2f315001     1  89e5           mov ebp,esp
0x2f315003     3  56             push esi
0x2f315004     4  57             push edi
0x2f315005     5  689180f02b     push 0x2bf08091
0x2f31500a    10  3b25f836f009   cmp esp,[0x9f036f8]
0x2f315010    16  7305           jnc 23  (0x2f315017)
            ;; debug: statement 10
            ;; code: STUB, StackCheckStub, minor: 0
0x2f315012    18  e8099e3f08     call 0x3770ee20

0x2f315017    23  33c0           xor eax,eax
0x2f315019    25  8945f4         mov [ebp+0xf4],eax
0x2f31501c    28  e965000000     jmp 134  (0x2f315086)

            ; Loop body - print "a" to console.log
0x2f315021    33  8b5613         mov edx,[esi+0x13]
            ;; object: 0x50315315 <String[7]: console>
0x2f315024    36  b915533150     mov ecx,0x50315315

            ;; debug: statement 96
            ;; code: contextual, LOAD_IC, UNINITIALIZED
0x2f315029    41  e8521b4008     call LoadIC_Initialize  (0x37716b80)

0x2f31502e    46  50             push eax
0x2f31502f    47  ff75f4         push [ebp+0xf4]
            ;; object: 0x50312369 <String[3]: log>
0x2f315032    50  b969233150     mov ecx,0x50312369

            ;; debug: statement 96
            ;; debug: position 104
            ;; code: CALL_IC, UNINITIALIZED, argc = 1 (id = 43)
```

```
0x2f315037    55  e884f63f08     call 0x377146c0


0x2f31503c    60  8b75fc         mov esi,[ebp+0xfc]
0x2f31503f    63  8b45f4         mov eax,[ebp+0xf4]

            ; test if the least significant bit is set
0x2f315042    66  a801           test al,0x1
            ; if it is not, "a1" is a number
            ; and a jump to the label "75" is performed
0x2f315044    68  7405           jz 75  (0x2f31504b)
            ; if it was set it is an object
            ; and "ToNumber" is called
            ;; debug: position 85
            ;; code: STUB, ToNumberStub, minor: 0
0x2f315046    70  e8155e3f08     call 0x3770ae60


            ; a++ - 0x2 is added because the least significant bit is a flag
0x2f31504b    75  83c002         add eax,0x2
0x2f31504e    78  7004           jo 84  (0x2f315054)
0x2f315050    80  a801           test al,0x1
0x2f315052    82  7211           jc 101  (0x2f315065)
0x2f315054    84  83e802         sub eax,0x2
0x2f315057    87  89c2           mov edx,eax
0x2f315059    89  b802000000     mov eax,0x2

            ;; debug: position 86
            ;; code: BINARY_OP_IC, UNINITIALIZED (id = 31)
0x2f31505e    94  e87d0a4108     call 0x37725ae0

0x2f315063    99  a811           test al,0x11
0x2f315065   101  8945f4         mov [ebp+0xf4],eax

            ;; object: 0x2500ba99 Cell for 5888
0x2f315068   104  bb99ba0025     mov ebx,0x2500ba99
0x2f31506d   109  836b0302       sub [ebx+0x3],0x2
0x2f315071   113  7913           jns 134  (0x2f315086)

            ;; debug: statement 62
            ;; code: STUB, InterruptStub, minor: 0
0x2f315073   115  e8c89b3f08     call 0x3770ec40

0x2f315078   120  a801           test al,0x1
```

```
            ;; object: 0x2500ba99 Cell for 5888
0x2f31507a   122  bb99ba0025     mov ebx,0x2500ba99
0x2f31507f   127  c74303002e0000 mov [ebx+0x3],0x2e00
0x2f315086   134  ff75f4         push [ebp+0xf4]
0x2f315089   137  b814000000     mov eax,0x14
0x2f31508e   142  5a             pop edx
0x2f31508f   143  89d1           mov ecx,edx
0x2f315091   145  0bc8           or ecx,eax
0x2f315093   147  f6c101         test_b cl,0x1
0x2f315096   150  7309           jnc 161  (0x2f3150a1)
0x2f315098   152  3bd0           cmp edx,eax
0x2f31509a   154  7c85           jl 33  (0x2f315021)
0x2f31509c   156  e90f000000     jmp 176  (0x2f3150b0)
            ;; debug: position 79
            ;; code: COMPARE_IC, UNINITIALIZED (id = 24)
0x2f3150a1   161  e87a923f08     call 0x3770e320


            ; test the loop condition and jump conditionally
            ; to the beginning of the loop (label 33)
0x2f3150a6   166  a810           test al,0x10
0x2f3150a8   168  85c0           test eax,eax
0x2f3150aa   170  0f8c71ffffff   jl 33  (0x2f315021)

            ;; object: 0x2bf08091 <undefined>
0x2f3150b0   176  b89180f02b     mov eax,0x2bf08091
            ;; object: 0x2500ba99 Cell for 5888
0x2f3150b5   181  bb99ba0025     mov ebx,0x2500ba99
0x2f3150ba   186  836b0302       sub [ebx+0x3],0x2
0x2f3150be   190  7913           jns 211  (0x2f3150d3)
0x2f3150c0   192  50             push eax
            ;; code: STUB, InterruptStub, minor: 0
0x2f3150c1   193  e87a9b3f08     call 0x3770ec40
0x2f3150c6   198  58             pop eax
            ;; object: 0x2500ba99 Cell for 5888
0x2f3150c7   199  bb99ba0025     mov ebx,0x2500ba99
0x2f3150cc   204  c74303002e0000 mov [ebx+0x3],0x2e00

            ;; debug: position 117
            ;; js return
0x2f3150d3   211  89ec           mov esp,ebp

0x2f3150d5   213  5d             pop ebp
```

```
0x2f3150d6   214  c21800         ret 0x18
0x2f3150d9   217  0f1f00         nop
```

Appendix G

Appendix G - RingoJs compiler output

To examine the object memory model of Rhino, the following JavaScript test program is compiled:

```
var o = {};
o.a = "foo";
o["a"] += "bar";

var a1 = "a";

var output = o[a1];
```

The output code is a class file. When de-compiled to java source, the main script function is the following:

```
private static Object _c_script_0(
    RingoObjectTest paramRingoObjectTest,
    Context paramContext,
    Scriptable paramScriptable1,
    Scriptable paramScriptable2,
    Object[] paramArrayOfObject)
{
    ScriptRuntime.initScript(paramRingoObjectTest,
    paramScriptable2, paramContext, paramScriptable1, false);

    Object localObject = Undefined.instance;
    ScriptRuntime.setName(
      ScriptRuntime.bind(paramContext, paramScriptable1, "o"),
      ScriptRuntime.newObjectLiteral(
        ScriptRuntime.emptyArgs,
        ScriptRuntime.emptyArgs,
        null,
        paramContext,
        paramScriptable1
      ),
      paramContext,
      paramScriptable1,
      "o"
    );

    localObject = ScriptRuntime.setObjectProp(
      ScriptRuntime.name(
        paramContext,
        paramScriptable1,
        "o"
      ),
      "a",
      "foo",
      paramContext
    );

    Object tmp64_61 =
      ScriptRuntime.name(paramContext, paramScriptable1, "o");
    String tmp67_65 = "a";

    localObject = ScriptRuntime.setObjectElem(
      tmp64_61,
      tmp67_65,
      ScriptRuntime.add(
        ScriptRuntime.getObjectElem(
          tmp64_61,
          tmp67_65,
          paramContext
        ),
        "bar"
      ),
      paramContext
    );

    ScriptRuntime.setName(
      ScriptRuntime.bind(
        paramContext,
        paramScriptable1,
        "a1"
      ),
      "a",
      paramContext,
      paramScriptable1,
      "a1"
    );

    ScriptRuntime.setName(
      ScriptRuntime.bind(paramContext, paramScriptable1, "output"),
      ScriptRuntime.getObjectElem(
        ScriptRuntime.name(paramContext, paramScriptable1, "o"),
        ScriptRuntime.name(paramContext, paramScriptable1, "a1"),
        paramContext,
        paramScriptable1
      ),
      paramContext,
      paramScriptable1,
      "output"
    );

    return localObject;
}
```

Loops test The inner loop of the "Loops" benchmark is:

```
var val = 0;
for(var x = 0; x <= 1000000; x++){
    val = val + x;
}
```

The compiled code for the script function for that loop is the following. The "_k0" variable is a constant defined in the class corresponding to the value 10000.

```
private static Object _c_script_0(
    RingoLoopTest paramRingoLoopTest,
    Context paramContext,
    Scriptable paramScriptable1,
    Scriptable paramScriptable2,
    Object[] paramArrayOfObject)
  {
    ScriptRuntime.initScript(paramRingoLoopTest, paramScriptable2,
      paramContext, paramScriptable1, false);
    Object localObject = Undefined.instance;
    ScriptRuntime.setName(
      ScriptRuntime.bind(paramContext, paramScriptable1, "val"),
      OptRuntime.zeroObj,
      paramContext,
      paramScriptable1,
      "val"
    );

    ScriptRuntime.setName(
      ScriptRuntime.bind(paramContext, paramScriptable1, "x"),
      OptRuntime.zeroObj,
      paramContext,
      paramScriptable1,
      "x"
    );

    while (true) {
      localObject = ScriptRuntime.setName(
        ScriptRuntime.bind(paramContext, paramScriptable1, "val"),
        ScriptRuntime.add(
          ScriptRuntime.name(paramContext, paramScriptable1, "val"),
          ScriptRuntime.name(paramContext, paramScriptable1, "x"),
          paramContext
        ),
        paramContext,
        paramScriptable1,
        "val"
      );
      ScriptRuntime.nameIncrDecr(paramScriptable1, "x", paramContext,
      2);

      if (!ScriptRuntime.cmp_LE(
        ScriptRuntime.name(paramContext, paramScriptable1, "x"), _k0))
        break;
    }

    return localObject;
 }
```

Bibliography

[abo13a] About perl. `http://www.perl.org/about.html`, February 2013. Accessed: 2013-03-01.

[abo13b] About ruby. `http://www.ruby-lang.org/en/about/`, February 2013. Accessed: 2013-03-01.

[abo13c] Ruby on rails. `http://rubyonrails.org/`, February 2013. Accessed: 2013-03-01.

[Aik13a] A. Aiken. Coursera course - compilers: Slides on constant propagatino. `https://class.coursera.org/compilers-003/lecture/index`, June 2013. Accessed: 2013-06-28.

[Aik13b] A. Aiken. Coursera course - compilers: Slides on orderings. `https://class.coursera.org/compilers-003/lecture/index`, June 2013. Accessed: 2013-06-28.

[ALSU86] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullmamn. *Compilers - Principles, Techniques, & Tools*. Addison-Wesley, second edition edition, 1986.

[Avr13] A. Avram. Latest dart vm beats jvm in deltablue benchmark. `http://www.infoq.com/news/2013/05/Dart-Java-DeltaBlue`, May 2013. Accessed: 2013-06-28.

[Bak12] L. Bak. Using map objects to access object properties in a dynamic object-oriented programming language, August 2012. Patent: US 8,244,755 B1.

[Boe13] H. Boehms. A garbage collector for c and c++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/, June 2013. Accessed: 2013-06-28.

[Con13] J. Conrod. A tour of v8: object representation. http://www.jayconrod.com/posts/52/a-tour-of-v8-object-representation, June 2013. Accessed: 2013-06-28.

[Cor13a] Microsoft Corporation. The c switch statement (c). http://msdn.microsoft.com/en-us/library/66k51h7a%28v=vs.80%29.aspx, June 2013. Accessed: 2013-06-28.

[Cor13b] Nullstone Corporation. Compiler optimizations. http://www.compileroptimizations.com/, June 2013. Accessed: 2013-06-28.

[Cox07] R. Cox. Regular expression matching can be simple and fast. http://swtch.com/~rsc/regexp/regexp1.html, January 2007. Accessed: 2013-06-28.

[cpl13] cplusplus.com. longjmp - c++ reference. http://www.cplusplus.com/reference/csetjmp/longjmp/, June 2013. Accessed: 2013-06-28.

[dev13a] Mozilla developer network - javascript. https://developer.mozilla.org/en-US/docs/JavaScript, February 2013. Accessed: 2013-06-20.

[Dev13b] RingoJs Developers. Ringojs. http://ringojs.org, June 2013. Accessed: 2013-06-28.

[Dev13c] Valgrind Developers. Valgrind home. http://valgrind.org/, June 2013. Accessed: 2013-06-28.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[ecm11] Ecmascript language specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm, June 2011. Accessed: 2013-06-28.

[eva13] Official perl documentation - eval. http://perldoc.perl.org/functions/eval.html, February 2013. Accessed: 2013-03-01.

[ext13] Extending and embedding the python interpreter. http://docs.python.org/2/extending/, February 2013. Accessed: 2013-03-01.

[Fou13] Python Software Foundation. Quotes about python. `http://www.python.org/about/quotes/`, February 2013. Accessed: 2013-03-01.

[GBJL00] Dick Grune, Henri Bal, Ceriel Jacobs, and Koen Langendoen. *Modern Compiler Design*. Wiley, 2000.

[GES$^+$09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[Git13] GitHub. Top languages - github. `https://github.com/languages/`, June 2013. Accessed: 2013-06-28.

[GMM] Gregor Gabrysiak, Stefan Marr, and Falko Menge. Meta programming and reflection in php.

[goo13] Google web toolkit. `https://developers.google.com/web-toolkit/overview`, February 2013. Accessed: 2013-03-01.

[Han10] Stefan Hanenberg. Doubts about the positive impact of static type systems on programming tasks in single developer projects - an empirical study. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 300–303. Springer Berlin Heidelberg, 2010.

[Han13] Troy D. Hanson. uthash: a hash table for c structures. `http://troydhanson.github.io/uthash/index.html`, June 2013. Accessed: 2013-06-28.

[Har08] Jon Harrop. *F# for Scientists*. Wiley, 2008.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

[Hem13] Z. Hemel. Dart2js outperforms hand-written javascript in deltablue benchmark. `http://www.infoq.com/news/2013/04/dart2js-outperforms-js`, April 2013. Accessed: 2013-06-28.

[IBM13] IBM. Ibm globalization - icu. `http://www-01.ibm.com/software/globalization/icu/`, June 2013. Accessed: 2013-06-28.

[Int11] International Organization for Standardization. *ISO/IEC 9899:201X*, April 2011. Accessed: 2013-03-01.

[IOC$^+$12] Kazuaki Ishizaki, Takeshi Ogasawara, José G. Castanos, Priya Nagpurkar, David Edelsohn, and Toshio Nakatani. Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. In Steven Hand and Dilma Da Silva, editors, *VEE*, pages 169–180. ACM, 2012.

[JGB12] G. Steele G. Bracha J. Gosling, B. Joy and A. Buckley. The java language specification. `http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`, July 2012. Accessed: 2013-03-01.

[Lip03] Eric Lippert. How do the script garbage collectors work? `http://blogs.msdn.com/b/ericlippert/archive/2003/09/17/53038.aspx`, September 2003. Accessed: 2013-06-20.

[Man10] D. Mandelin. Bug 542071 - analyze implementation of closures in v8. `https://bugzilla.mozilla.org/show_bug.cgi?id=542071`, January 2010. Accessed: 2013-06-28.

[Man13] David Mandelin. Bug 536277 - (jaeger) [meta] jaegermonkey: Baseline method jit. `https://bugzilla.mozilla.org/show_bug.cgi?id=536277`, May 2013. Accessed: 2013-06-28.

[MAso13] Antony Dovgal Nuno Lopes Hannes Magnusson Georg Richter Damien Seguy Jakub Vrana Mehdi Achour, Friedhelm Betz and several others. Php manual. `http://www.php.net/manual/en/index.php`, February 2013. Accessed: 2013-03-01.

[MW98] J. Maloney and M. Wolczko. Uw deltablue constraint solver. `http://www.cs.washington.edu/research/constraints/deltablue/`, June 1998. Accessed: 2013-06-28.

[NBD$^+$05] Oscar Nierstrasz, Alexandre Bergel, Marcus Denker, Stéphane Ducasse, Markus Gälli, and Roel Wuyts. On the revival of dynamic languages. In Thomas Gschwind, Uwe Aßmann, and Oscar Nierstrasz, editors, *Software Composition*, volume 3628 of *Lecture Notes in Computer Science*, pages 1–13. Springer Berlin Heidelberg, 2005.

[nod13] node.js home page. `http://nodejs.org/`, February 2013. Accessed: 2013-06-20.

[Ous98] J.K. Ousterhout. Scripting: higher level programming for the 21st century. *Computer*, 31(3):23 –30, mar 1998.

[OZPSG10] F. Ortin, D. Zapico, J. B G Perez-Schofield, and M. Garcia. Including both static and dynamic typing in the same programming language. *Software, IET*, 4(4):268–282, 2010.

[Par12] Terence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2012.

[Pau07] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12 –15, feb. 2007.

[per13a] Official perl documentation - perldata. `http://perldoc.perl.org/perldata.pdf`, February 2013. Accessed: 2013-03-01.

[per13b] Official perl documentation - perlobj. `http://perldoc.perl.org/perlobj.pdf`, February 2013. Accessed: 2013-03-01.

[per13c] Official perl documentation - perlop. `http://perldoc.perl.org/perlop.pdf`, February 2013. Accessed: 2013-06-28.

[per13d] Official perl documentation - sub. `http://perldoc.perl.org/functions/sub.html`, February 2013. Accessed: 2013-03-01.

[Per13e] N. Pershin. Inheritance and the prototype chain - javascript | mdn. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain`, May 2013. Accessed: 2013-06-28.

[Pre00] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23 –29, oct 2000.

[Pro13] Mono Project. Mono:runtime - mono. `http://www.mono-project.com/Mono:Runtime#Mono.27s_use_of_Boehm_GC`, June 2013. Accessed: 2013-06-28.

[pyt13a] The python language reference. `http://docs.python.org/2/reference/`, February 2013. Accessed: 2013-03-01.

[pyt13b] The python standard library. `http://docs.python.org/2/library/index.html`, February 2013. Accessed: 2013-03-01.

[RHBV11] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that man do. `http://www.cs.purdue.edu/homes/gkrichar/papers/eval-ecoop-2011.pdf`, 2011. Accessed: 2013-06-28.

[rub10] Programming languages | ruby. http://www.ipa.go.jp/osc/english/ruby/Ruby_final_draft_enu_20100825.pdf, August 2010. Accessed: 2013-03-01.

[Rus04] M. V. Rushton. Static and dynamic type systems. http://thesis.haverford.edu/dspace/bitstream/handle/10066/624/2004RushtonM.pdf?sequence=2, April 2004. Master thesis, Accessed: 2013-06-28.

[Ser13a] Amazon Web Services. Amazon ec2 faqs. http://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it, June 2013. Accessed: 2013-06-28.

[Ser13b] Amazon Web Services. Amazon ec2 instances. http://aws.amazon.com/ec2/instance-types/, June 2013. Accessed: 2013-06-28.

[SH11] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. *SIGPLAN Not.*, 47(2):97–106, October 2011.

[Shi13] J. Shin. memusg - measure memory usage of processes. https://gist.github.com/netj/526585, June 2013. Accessed: 2013-06-28.

[Sof13] TIOBE Software. Tiobe software: Tiobe index:. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, June 2013. Accessed: 2013-03-01.

[Tea12] Google Chromium Team. The benchmark - octane – google developers. https://developers.google.com/octane/benchmark, August 2012. Accessed: 2013-06-28.

[tea13a] Google's V8 team. assembler.h. https://code.google.com/p/v8/source/browse/trunk/src/assembler.h, June 2013. Accessed: 2013-06-28.

[tea13b] Google's V8 team. v8 - v8 javascript engine - google project hosting. http://code.google.com/p/v8/, June 2013. Accessed: 2013-06-28.

[Tea13c] Mozilla Javascript Team. Memory management - javascript | mdn:. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management, June 2013. Accessed: 2013-06-28.

[Tea13d] Mozilla Rhino Team. Mozilla rhino | github. https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/ScriptRuntime.java?source=cc, June 2013. Accessed: 2013-07-24.

[Tea13e] Mozilla Rhino Team. Nativeobject.java. https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/NativeObject.java, June 2013. Accessed: 2013-06-28.

[Tea13f] Mozilla Rhino Team. Rhino | mdn. https://developer.mozilla.org/en/docs/Rhino, June 2013. Accessed: 2013-06-28.

[Tea13g] Mozilla Rhino Team. Scriptableobject.java. https://github.com/mozilla/rhino/blob/master/src/org/mozilla/javascript/ScriptableObject.java, June 2013. Accessed: 2013-06-28.

[tea13h] NodeJs team. Executing javascript node.js v0.10.12 manual & documentation. http://nodejs.org/api/vm.html, June 2013. Accessed: 2013-06-28.

[Tra09] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, July 2009.

[Wel13] David Welton. Programming language popularity. http://www.langpop.com/, February 2013. Accessed: 2013-03-01.

[Win11] A. Wingo. a closer look at crankshaft, v8's optimizing compiler. http://wingolog.org/archives/2011/08/02/a-closer-look-at-crankshaft-v8s-optimizing-compiler, August 2011. Accessed: 2013-06-28.

[Win12a] A. Wingo. inside full-codegen, v8's baseline compiler. http://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler, July 2012. Accessed: 2013-06-28.

[Win12b] A. Wingo. v8: a tale of two compilers. http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers, July 2012. Accessed: 2013-06-28.