Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science and Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Petr Smrček**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Comparison of scripting languages based on the native code interoperability**

Guidelines:

Get familiar with several scripting languages (at least Lua and Javascript). Study the API for interoperability of selected scripting environments with native code in C++. Implement simple C++ application capable of running code in selected scripting languages. Observe the efficiency of script execution in selected environments. Design measurements which minimize the impact of the efficiency of script execution on the efficiency of function calls from and to native code. Create several model applications with various levels of interoperability and complexity of the scripted code. Compare the selected scripting languages based on the appropriateness of usage in the model applications.

Bibliography/Sources:

IERUSALIMSCHY, R. Programming in lua. Lua. Org, 2013.
MAMONE, Mark. Practical Mono. Apress, 2005.
OUSTERHOUT, J. K. Scripting: Higher level programming for the 21st century. Computer. 1998, 31, 3, s. 23-30.
VARANESE, A. Game Scripting Mastery (Premier Press Game Development (Paperback)). Course
Technology Press, 2002.
GOODMAN, Danny. JavaScript bible. John Wiley & Sons, 2007.

Diploma Thesis Supervisor: Ing. Tomáš Barák

Valid until the end of the winter semester of academic year 2016/2017

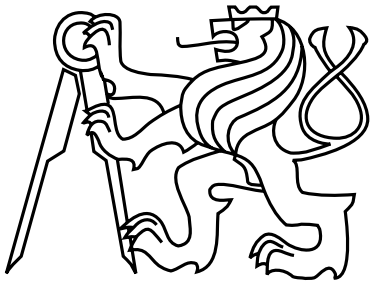
[redacted]
doc. Ing. Filip Železný, Ph.D.
Head of Department

L.S.

[redacted]
prof. Ing. Pavel Ripka, CSc.
Dean

Prague, October 8, 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Comparison of scripting languages based on the native code connectivity

Bc. Petr Smrček

Supervisor: Ing. Tomáš Barák

Study Programme: Open Informatics, Master

Field of Study: Software Engineering

January 10, 2016

Aknowledgements

I would like to thank my advisor Ing. Tomáš Barák for giving me the idea of this interesting work from which I learned a lot, and for helping me with suggestions and corrections. I would also like to thank my family and girlfriend for keeping me alive while I was mindlessly working on it.

Declaration

I declare that I elaborated this thesis on my own and that I mentioned all the information sources and literature that have been used in accordance with the Guideline for adhering to ethical principles in the course of elaborating an academic final thesis.

In Prague on January 10, 2016 .....

Abstract

This thesis studies capabilities of various scripting languages to communicate with native code. Languages Lua, JavaScript (implementation V8) and C# (implementation Mono) are studied and tested for performance of invoking script from C++ code and and calling the C++ code from script. This work also contains an implementation of a custom scripting language NativeScript based on LLVM and its comparison to the other languages. The results are discussed and explained using the language implementation details.

Abstrakt

Tato práce se zabývá schopnostmi různých skriptovacích jazyků komunikovat s nativním kódem. Jsou zkoumány jazyky Lua, JavaScript (implementace V8) a C# (implementace Mono) a je testován jejich výkon pro volání skriptu z C++ kódu a volání C++ kódu ze skriptu. Práce obsahuje také implementaci vlastního skriptovacího jazyka NativeScript postaveného na LLVM a jeho porovnání s ostatními jazyky. Výsledky jsou diskutovány a zdůvodněny na základě detailů implementací daných jazyků.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software development and programming languages are evolving as the time goes. As the computing power price decreases and the most expansive commodity becomes the work of the programmer, languages originally developed for scripting are getting more popular for general application development [78]. To the most important advantages of these languages belong simplicity, independence of platform and just in time compilation, which speed up the development greatly.

However, many of the applications today are still computational heavy while requiring to be responsive to the user. For this reason, performance plays an important role for them. Some of the examples are computer games or graphical editors. But even in such applications scripting can be still very useful. Computational heavy operations can be implemented natively to ensure the required speed, while the business logic, GUI or generally any code that tends to change often can be scripted. The code written in script than can be developed or modified without the need of recompilation of the whole application, sometimes even after the release by the users themselves. It is apparent that in such applications the execution tends to switch between the native code and the script quite often.

There are numerous scripting languages that offer C/C++ API to provide the required means of communications. There are multiple ways to implement such a language and each of them brings some advantages and disadvantages. The implementation details can affect not only the speed of its execution, compilation or data marshalling, but also the syntax of the language or the complexity of the code required to connect the specific C/C++ API with the script. For those reasons, the selection of correct scripting language for specific application can play an important role.

1.1 Aim of the project

The main purpose of this work is to examine several scripting languages and their binding capabilities to native code (C++). Under the term *scripting language* we understand a language (and its implementation), that is capable of interpretation or runtime compilation and communication with C/C++ code (or generally any system language). Such a language can be used to control other programs, to define high-level behovior, or to extend or customize existing programs.

In this work we mainly focus on the binding capabilities. There are already benchmarks comparing the language execution itself [25] [26], but we weren't able to find anything comparing the bindings for various scripting language types. We especially focus on the following criteria:

- Performance of invocation the script and passing parameters.
- Performance of invocation C/C++ callback from within the script and receiving parameters created by script.
- Effectiveness of optimizations for binding implementation and execution delegation.
- Ease of usage of the language and the binding API, including the availability of documentation.
- Requirements and difficulty of making the language operational, including size of library, legal issues and ease of compilation.

Rather than running generic tests for many languages, we select only few languages that significantly differ in their implementation approach. We study these languages in detail and identify advantages and disadvantages of their features. We also implement our own language and compare it to the existing solutions. The results of this project should be usable as a guide for selecting the best language type for given case. More, it should give an insight into the binding speeds and suggest which approaches and optimizations can increase performance for given language.

1.2 Contents

In chapter 2 we introduce the concept of scripting and the reasons for which scripting languages were created. We also present and discuss typical features of scripting languages and their implementations.

Chapter 3 presents 3rd-party languages (and their implementations) we selected for this work. We briefly discuss their history, main target and non-functional topics like licensing and platform availability. Most of the chapter is dedicated to the internal details of languages' implementations, that will be later used to explained the results in chapter 6. We also talk about their binding API and ease of usage.

Chapter 4 is dedicated to the topic of creating a custom scripting language. We discuss possible motivations to create a custom language instead of using existing solution. Then we present our own simple language NativeScript implemented using existing tools and frameworks like Flex, Bison and LLVM. We discuss its syntax and details of its implementation.

Chapter 5 contains description of our testing approach. We introduce our testing application, used environments, library settings and measurement tools. We also describe our testing scenarios and the language features, they are supposed to measure.

In chapter 6 we present the measured results. We use knowledge from chapters 3 and 4 to explain the results and compute approximations of real binding speeds without overheads. We also present recommendations for each language application.

Finally, in chapter 7 we summarize the results of this work and evaluate its success. We also discuss possible continuation of this work.

Chapter 2

Scripting Languages

Scripting programming languages are a specific type of languages. In this chapter we discuss their origin, purpose and typical features.

2.1 Evolution of programming languages

In order to better understand the concepts for scripting programming languages and scripting in general, we should briefly look at the reasons that made them birth. The programming languages in general were created to provide tools for defining what computer should do. As the time went, the programs got more complicated which caused new programming languages to emerge, to better address this issue. In the following part we will briefly describe the evolution of programming languages to show why the scripting programming languages were created and what is their purpose.

2.1.1 Beginnings

The first programming was nothing more than specifying the machine instructions manually. *Assembly programming* languages were used to do this. One statement in such a language represented a single machine instruction. The problem with this approach was that programmer had to do everything, including allocating the registers and handling the stack on procedure call. As the programs grew larger, the code became complicated and very difficult to maintain [55].

There was a need for some kind of abstraction and thus new higher-level programming languages such as Lisp or Fortran emerged by the late 1950s. By higher-level we mean that the ratio number of machine instructions per statement increased. Soon languages like C emerged. These languages enabled the programmer to use abstract constructs like variable assignments, function calls or cycles instead of manually handling the stack or the communication between memory and registers. The code was then translated into the machine code by compilers. We generally call this type of languages *system programming languages*. The performance of the system programming languages was lower than the performance of the assembly languages, but they enabled faster development of even bigger and complicated software and its reasonable maintenance [70].

2.1.2 Typing and objects

Typing in general is a way to specify a meaning of an information. By giving a type to some piece of data, we specify an information about its purpose and usage. For example by specifying some data to be a string of characters, we prevent it to be a part of arithmetic operations. In *statically typed languages* every piece of data has its type. Because the compiler has an information about how the data should be managed, it can discover some errors in usage during compilation. More, it can optimize the final machine code by generating specific instructions. In *dynamically typed languages* the usage of the data is not explicitly restricted, so the compiler has to generate additional instructions for checking the type. This reduces the performance of the final program, however it provides greater variability for the programmer [38].

As the software was becoming bigger and bigger, more abstraction was needed. The programming languages tended to become more programmer-friendly than machine-friendly. The new concept of *abstract data types* was introduced. Abstract data types in general represent concepts from the programmer point of view rather than just restricting usage of some bytes. An example of such a type can be a list. A programmer uses it to work with a set of data in a simple manner, although its computer representation might in fact be far more complex, for example the allocation of memory during additions to the list. *Object oriented languages* emerged around this principle, which enabled development of enormous software by big developer teams. It is not in capabilities of an individuals to comprehend such a big systems, so the abstract data types provided tools to build clearly defined interfaces and thus enabling big teams to efficiently cooperate. [44]

2.1.3 Scripting languages

The evolution as described so far enabled the development and maintenance of big and complicated programs. However, as the programs grew bigger, so did the amount of work the compiler had to do. Even small change in the code of a huge program can force a recompilation of a big part of it. This can take a significant amount of time, thus slowing the development greatly. Also, usually the syntax and principles of a system programming language require good programmers who had studied and worked with the language for some time.

These are some of the problems solved by scripting programming languages. The scripting programming languages tend to be simpler and easier to use. They are even higher-level than system programming languages, meaning a single statement can represent even hundreds or thousands of machine instructions. One of their main advantages is that they don't need to be compiled, at least not the same way the system programming languages are. Their source code is parsed and interpreted during runtime, which makes it in fact an additional program input. Scripting languages may thus seem to be at the top of the programming language evolution tree, since they provide even faster development and simpler syntax. However, those features come with a great performance cost and so these languages are often used only for some specific tasks. As the computers are getting more powerful, this performance impact is being compensated and today there are scripting languages used even for development of whole applications [78].

We can see that one of the the main trends in evolution of the programming languages is going closer to the user - the programmer. However the code has to be always executed by the machine in the end and making the job easier for the programmer makes the job more difficult for the machine. The computing power of the hardware is still growing, however there are applications that need as much speed as possible, so even assembly programming languages are still used today.

2.2 Scripting principles

There is not a strict definition of what a scripting programming language exactly is. For the purpose of this work we will use this term in connection with languages that are compiled or interpreted during runtime. We should also note here that this feature depends more on the specific implementation of the language rather that on the language itself. Compilers can be written even for languages originally created for scripting and on the other hand, any language can be interpreted. However, the syntax of the scripting language is usually more convenient for scripting and vice versa, so different cases are not as common. For this reason and reader's convenience, in the rest of the work we will be talking about scripting languages, even though we really mean also their implementations, that use interpretation. In this section we will describe the common applications of scripts and the basic principles upon which they work.

2.2.1 Main purpose

There is quite a lot of scripting programming languages today, used in various areas. E.g. on Unix platforms, shell scripts or the terminal input in general can be referred as scripting language code. Shell interprets the commands from the script or user input and executes respective actions. Another application of scripting is web, most often using JavaScript. The web page downloaded from the web server can contain JavaScript code, which is then interpreted by the web browser. It can be used to e.g. interactively modify the GUI created from the HTML, to start another HTTP communication or even to give commands to the GPU [72]. Apart from that, scripting is also used to enable users to modify or add some features of already released application. Such application can offer its users to e.g. modify the GUI, automate some of the often used commands [61] or do computations (e.g. spreadsheet macros).

All those examples above have something in common. The scripting language is not used to develop the whole application. It merely assumes there is already a set of components implemented in another language (Unix programs like ls, DOM in web browser, predefined functions is spreadsheet application). The script doesn't implement complex algorithms or data structures here. It simply gives commands and glues those components together. It connects smaller pieces to build higher-level logic. This approach is very powerful in the way that most of the machine execution happens efficiently within those components, while the script developer can build custom and often complex logic using simple language (e.g. connecting shell commands using pipes). All without the need of compilation delay and platform issues.

2.2.2 JIT and efficiency

The high performance cost of the scripting languages comes mainly from the fact that they are interpreted. The compiler of a system programming language usually translates the character + in the source code directly into machine instruction *ADD*. The interpreter however has to parse the input, check the correctness of the syntax and than call its own *add(x,y)* function, that contains the machine instruction *ADD*. We can see that many additional instructions have to be executed to achieve the same result. The above approach is however quite naive and today interpreters are much more sophisticated.

The obvious optimization is to parse and process the input once and save it within some data structure for later use. This way functions or other constructs can be predefined and used later without the need to parse the code again. This however only prevents repeated parsing of the input and there is no improvement in the execution itself. The code itself can be small and the execution quite long by using recursion or loops.

The common approach of increasing the performance of script execution is to actually compile it. The compilation happens during runtime and this principle is generally called *just-in-time compilation*, shortly JIT [34]. There are two main approaches:

- **Compilation to machine code**. For the best performance the script can be compiled into the target machine code directly. This compilation can take significant amount of time and is dependent on the target platform, so the script doesn't have to work everywhere.

- **Compilation to bytecode**. Bytecode is an intermediate code representation [43]. It is usually still platform independent, but no longer human-readable, because it is represented by numeric codes, constants and references. However it enables much faster interpretation than direct interpretation of source code. The program that executes the bytecode instructions is usually referred as a *virtual machine*, since it simulates real machine executing real instructions.

Some programming languages use both these principles. E.g. Java compiles its source into bytecode, stored in its *.class* files. This bytecode remains platform independent and can be distributed. When the program is ran, the platform specific interpreter called *Java virtual machine* converts bytecode into the native code and executes it [80].

It might seem that by compiling the scripting languages we returned back to the waiting for the time-consuming compilation and gained nothing in the end. That is not entirely correct. The JIT compilation happens during runtime and thus it allows features like *adaptive optimization*. The complexity and level of optimization can be chosen with respect to the requirements on delay of the start of the script execution. The code can be e.g. interpreted the first time it is ran to be executed quickly and then compiled to more efficient form, as it is used more often. Also when the scripting language is used to invoke the underlying native components rather that to implement the whole application, usually only a small portion of the code has to be recompiled on change and the rest can remain cached.

2.2.3 Ease of usage

System programming languages or generally lower-level programming languages let the programmer work with the basic constructs like bit operations or memory allocation. Due to this fact, the programmers have to understand the theory behind in order to write a working and quality code. They have to be educated in the topic so only specialists can do this work efficiently. Also considerable amount of the effort is spent on handling these constructs instead of on the implementation of the actual logic itself. The code created by this approach is usually very fast and optimized, however its development can be very costly.

On the other hand, scripting programming languages tend to be simple and platform independent. They solve many of the low-level problems automatically at the cost of performance. They also tend to be dynamically typed so the programmer doesn't have to care about types and usually neither declarations. Their basic components are chosen for power and ease of use rather than an efficient mapping onto the underlying hardware. Instead of indexed arrays, hash tables can be used for their higher generality and even numbers can be sometimes represented by strings to unify the handling of variables [69].

Because of these features, more of the total development effort can be spent on the desired logic. The programmers don't have to understand the mapping of the code onto hardware that much which enables even less seasoned programmers or often even other members of the team to create script code. Another huge benefit is the lack of compilation time which enables the developers to try out and test their code immediately. This way any errors can be fixed and tested immediately instead of waiting for another compilation. The development of the scripts thus tend to be cheaper and faster and they can be even used as tools for the other members of the team, like UX designers or animators, to incorporate their work into the final product directly.

The seasoned programmers can focus on the low-level algorithms and data structures to create fast and optimized components, which can be used by the other members of the team trough scripting. This approach balances the development cost (and time) and the performance.

2.3 Properties of scripting languages

Earlier we discussed the main principles of scripting programming languages. Here we will peek into other, more detailed properties which scripting programming language can have and we will discuss their advantages and disadvantages.

2.3.1 Memory safety

Memory related errors can cause the program to crash, expose a security leak or behave unpredictably. The lower-level languages tend to allow the programmer to manipulate the memory freely, e.g. access memory directly using an arbitrary address. This feature allows the result code to be very fast, since it lacks the extra operations that check the correctness of the memory operations. However, this feature allows memory errors and corruption to occur, which often causes unrecoverable crash of the program.

The higher-level languages, especially scripting languages, usually ensure the memory safety by explicitly checking the correctness of memory operations. This introduces additional performance overhead, however any errors are caught and can be handled. Even the crash of the virtual machine can be often handled from the native code, into which it is embedded. Thus even errors that slip trough the testing phase of development don't have to be always fatal. Because the errors are caught, more information about what caused the crash can be available. Developers can fix the code faster, further speeding up the development.

Since one of the main purposes of the scripting programming languages is to abstract the low-level operations from the programmer, they almost always ensure the memory safety. Sometimes their syntax may even hide the fact, that memory operations happen in the background. Even scripting languages that compile into the native code can ensure memory safety by generating guaranteed safe code, if the syntax of the language allows it.

2.3.2 Garbage collection

In lower-level programming languages the programmer usually has to explicitly mark that some memory won't be needed anymore by the program and the operating system can reclaim it. Not releasing the no longer needed memory is known as a *memory leak* which can eventually cause the program to crash, because all its available memory is exhausted. The release of the memory can become impossible when the last reference to it is lost.

Higher-level programming language implementations usually provide a feature called *garbage collector*. It is a part of the virtual machine that automatically frees the memory, that can be no longer used by the program. There are many approaches for garbage collection [63]:

- **Reference counting**. The GC uses pointers to allocated memory to count its usages. When there is no reference to some part of memory left, the GC frees this memory. The advantage is that the memory is freed as soon as possible and incrementally during the whole execution. However, reference counts have to be updated quite often and additional effort has to be put into breaking cycles (two pointers each referencing to the memory location of the other one).
- **Mark and sweep**. The GC happens in two phases. First, all memory accessible by the program is marked by iterating in tree-like fashion from all root variables. Second, the memory is scanned for unmarked objects and they are deallocated. The advantage is that no additional performance cost due to reference manipulation within the code is introduced, but the drawback is that the garbage collection happens all at once, so the execution has to be paused and the application can become unresponsive from time to time.
- **Copying**. Instead of deallocating individual pieces of memory that are no longer reachable, GC copies the still used pieces of memory into one continuous block and updates all the pointers. The rest of the memory can be used in its entirety, making the further allocation very fast. Because the pointers have to be updated, this GC is suitable only for languages, where pointers can be identified with certainty. It also requires at least twice the amount of memory, which is split and each time GC occurs, all pieces are copied to the other half.

- **Generational**. Because most memory allocations live only for a short time and the parts that actually survive the GC cycle are likely to stay in the memory for a while, it might be unnecessary to check the whole memory during each GC cycle. Generational GC assigns all new allocations into generation 0 and over time they are being promoted into higher generations (1, then 2 and so on). Lower generations are collected more often. This heuristic takes advantage of the observations of general behavior of memory and it tends to shorten the average GC time. However, the no longer needed long-live memory may remain allocated for longer time, since it is not checked that often.

- **Concurrent**. The GC happens concurrently with the program execution in a separate thread (or multiple threads). This approach mainly takes advantage of the multiprocessor machines. The application execution usually still has to be paused from time to time, but only for the minimal necessary time and the rest of the GC can be executed while the program is running. However, all the memory access of the running execution threads has to be often done in a specific way, which can impact the performance.

Even with the garbage collector, a memory leak may occur. If the memory is no longer used, but it is still reachable from the currently used variables, the GC cannot claim it. Thus the programmer still has to take care of getting rid of unnecessary references, especially if the execution is going to take some time and the memory might start to fill up.

While embedding the scripting language into the native application, some scripting data may be referenced from the C/C++ code only and the native API thus should provide means to cooperate with the GC. This can be achieved by using special pointers to this data, keeping the data alive only within the stack context of the C/C++ code where they are used, or providing explicit constructs for manual deallocation.

The garbage collection is generally slightly slower than freeing the objects explicitly. However the actual performance depends on many details and often there can be almost no additional cost [54].

2.3.3 Threads

Most of the modern hardware supports multiple threads of execution. Since there are usually multiple cores or even processors available, the thread execution can be truly concurrent, which can be convenient when the script needs to e.g. run some longer algorithm without disrupting its other responsibilities. It is simple to run a virtual machine per thread to achieve multiple concurrent script executions. However, if the concurrent script executions should cooperate, e.g. exchange or share data, the situation might be more complicated.

There might be virtual threads provided by the virtual machine available. However, they may not always run really concurrently. The overall difficulty of running a virtual machine is increasing with threads, since features like GC have to deal with multiple simultaneous allocating attempts or locks on various structures of the virtual machine itself, since system threads can interrupt it as well. More, threading is a feature provided by operating system and thus there is has to be specifically provided for various operating systems, reducing portability.

2.3.4 Native interface

The script has to be capable of communicating with the native optimized code. When the execution of the script is initiated, it might require arguments - and vice versa, the script might need to invoke some component with some arguments, or just simply return result value. For these reasons, scripting programming languages are usually designed to be easily embeddable and make the communication easy from the both sides. Within the script code, declarations without definitions may be used or sometimes even values and functions not declared can be used. On the native side, the implementation of the scripting language usually offers an API for defining function callbacks, passing or retrieving values and invoking script code.

The usual requirement of a scripting language is to be multiplatform. This might have impact on the representation of the data within the script and on the ways the data are exchanged between the script and the native application. There are multiple approaches of the representation of the primitive types (e.g. numbers):

- **Native values**. The scripting language implementation is compiled together with the native application for given platform and thus the native types can be used directly for the representation of the values within the script. This approach makes the data exchange faster, since the value can be simply copied. However, there is a small impact on the platform independence of the scripting language, since e.g. the maximal value of a number may not be clearly defined.
- **Internal platform independent values**. The scripting language implementation uses its own internal representation. An example of this is a representation of every value as a string. The advantage is a total platform independence, but the cost is necessity of marshalling the values passed trough the boundary between the script and the native code, which can cause significant performance loss.

Chapter 3

Languages Tested

We tested several different scripting languages in this work. They were selected in order to examine and compare different scripting approaches. All of the tested languages and interpreters are multiplatform (operational on at least the major platforms), free to use, embeddable from C++ and widely used. In this chapter we introduce all of the 3rd-party languages and interpreters tested, discuss their features and notable implementation details and evaluate their documentation and ease of installation and usage. The details discussed in this section might also be of interest to anyone who is trying to choose a scripting language and requires some specific features.

This section doesn't discuss the custom language and engine NativeScript, that was created during this work and compared to the existing solutions. It is discussed in chapter 4

3.1 Lua

Since Lua is generally less known compared to the other selected languages, we will discuss its inner syntax and constructs it offers. We will also examine its binding principles and some notable details of its inner implementation. There are many libraries extending the small and basic Lua core, so we will introduce some of the them.

3.1.1 Introduction

Lua [7] is a lightweight and easy to use scripting language. Its official interpreter we use in this work is written in clean ANSI C and compilable with C++, thus being embeddable on many systems and into many kinds of programs. It has a very small codebase (625 KB as of the version 5.3.0), so it can be compiled quickly, out-of-the-box and even on small devices without a lot of memory. Lua provides only a small number of primitives and constructs by default so there are e.g. no objects or inheritance. However it is highly extensible, so all those features can be easily implemented within the language itself.

Lua was created in 1993 at the Pontifical Catholic University of Rio de Janeiro in Brazil [59]. It is originally an academic language, however since its creation it highly evolved and became widely used in various fields of programming, especially game development [1].

Despite having such a small codebase, Lua became a mature language and today it contains modern features like incremental garbage collector or full lexical scoping.

Lua in this test represents lightweight and easy to use scripting language. There are many similar languages, e.g. AngelScript, GameMonkey, Io, Pawn, Squirrel or Scheme. We chose Lua for this work because of its widespread usage. The comparison with some other similar languages has been already made [81].

3.1.2 The language

Since its creation, the purpose of Lua was to be very lightweight and small, while being widely portable. Because of that, there are only 8 data types. Lua is dynamically typed, thus variables are typeless. The type is saved together with the value, so a single variable can contain different types during its lifespan. The notable features of the types are:

- **boolean** Simple true or false. However, any type can be used in condition and only false or nil evaluate to false.
- **nil** Has a single value *nil*. Its purpose is to be different from any other value.
- **number** All Lua numbers are double-precision floating-point numbers. It can be however changed to different type during compilation.
- **string** Strings in Lua are immutable and can be also used to hold any binary data.
- **function** Functions in Lua are first-class values. It means that all functions are anonymous and are given names by being assigned into variables as values. The functions have also full lexical scoping (also called static scoping [77]).
- **user data** Serves as a storage for arbitrary C data. It is mainly used trough the C API.
- **thread** Represents a Lua coroutine - a separate line of execution with its own stack etc. Coroutines however run sequentially rather concurrently than concurrently and cannot be preempted (by other coroutines).
- **table** Lua table is an associative array with additional extra features making it an universal building block for mostly anything. It can be indexed by any value except nil and has dynamic size. Like functions, tables are anonymous and can be assigned to any variable. Any table can be additionally assigned with *metatable*, that can change its default behavior, e.g. field access.

Lua has a lot of convenient features implemented into its syntax. E.g. the expression `myTable.tableField` is equivalent to `myTable["tableField"]`, which makes the tables available for being used also as packages or objects. Another example is a function definition: `function inc(x) return x+1 end` is equivalent to `inc = function(x) return x+1 end` and there are many other similar examples.

3.1.3 Binding to C/C++

The border between Lua values and C/C++ values is represented by a simple stack. Anytime a programmer wants to send a value into the Lua environment, she pushes it into the stack and invokes some API function, which specifies how the value will be used. E.g. the invocation of a Lua function call pops from the stack the value containing the function and every argument of that function. The stack is accessed by the programmer only from the native side, from the Lua side values are pushed and popped automatically.

By using this approach the problem of sending a value from a static type system into dynamic system is solved elegantly. Value of any type can be pushed to the stack with appropriate function. While popping a value, the developer should know its type and use appropriate function as well. If she doesn't know, the API provides functions like *lua_istable(...)*.

There is also no problem for the GC to know what values are still used on the native side, because simply none are. The values can be only read into C variables or created by pushing the stack, so the actual Lua representation of the values can be garbage collected freely.

3.1.4 Internal implementation

Lua compiles its source into bytecode instructions (called *opcodes*) [67], that are then interpreted by its VM. The bytecode is binary and highly optimized to make the VM run fast. It can be ran from memory or saved into a file and ran later, which can be used for caching the compilation.

Since version 5.0, Lua's virtual machine is register-based, which means that the instructions contain direct addresses of the operands. Upon entering a function, Lua allocates all local values into the registers. As a result, local variables access is very fast, since it avoids the *push* and *pop* operations to get local values from the stack. The drawback of this approach is that the instructions must hold the addresses of the operands, thus taking more space (4 bytes in Lua), which can grow the bytecode size and the time it takes to decode it [58].

The values in Lua are represented as *tagged-unions*, which are value-type pairs, where the type is represented by a simple integer and the value is represented by union. The values of types of dynamic size like strings or tables are implemented as pointers within the union. Numbers and booleans are stored unboxed directly in the union.

By default Lua's internal representation for number is C type *double*, which doesn't have guaranteed its limits by C standards. However, if the target platform implements the floating-point standard IEEE 754 [83], double has to represent at least its 32-bit format. In reality, on most platforms double is represented by the 64-bit format. If Lua is built for platform that doesn't guarantee these formats, the type can be still changed. So even though the number type in Lua is theoretically not entirely multiplatform, it requires the attention only in special cases and there are tools to solve the issue.

The strings are *internalized*, which means that a single copy of each string is stored within a hash table, which allows fast comparison and indexing using the hash table keys. If the string is very long, not all bytes are used to compute the hash, since such a operation would be very costly for long strings [58].

Although tables can be indexed by any value, if the indexes are integers, Lua saves the values in actual array. Single Lua table can have some values stored in an array and some in a hash table simultaneously. This can reduce the memory usage and increase the field access without the need for the programmer to specify the optimal data structure [46].

Lua uses classic mark-and-sweep garbage collector. From version 5.1 it is incremental instead of atomic, which means that only a part of the memory is collected during one cycle, resulting in shorter pauses.

3.1.5 Extension libraries

Because Lua's target is to be small, its core doesn't have the features that wouldn't be used by the majority of the applications. There are many libraries that extend Lua's functionality or improve some of its features for the cost of size or making the solution platform specific. Some of the libraries might also provide bindings to other libraries so they can be used directly from Lua. We mention only few of them that we consider interesting for this work, although there are many more [8].

- **LuaJIT** LuaJIT [71] is a competent implementation of the Lua interpreter. It is generally way faster than the official Lua implementation, but it is less portable and takes more memory and doesn't support all of the latest features of the official implementation. It is written in assembler and uses advanced techniques like tracing-compiler or SSA-based optimizations.

- **Threading support** Lua itself doesn't support concurrent execution of multiple threads. Its couroutines only create an illusion of this, while executing one after another and not being preemptive. In order to achieve actual concurrent execution, intercommunication and more thread-like behavior in general for multiple Lua scripts, extension library has to be used. There are many libraries with different approaches [9].

- **LuaBind** If there are many C/C++ functions that need to be exposed to Lua or many Lua functions that need to be called from C/C++, the binding code can grow to notable size. Especially if the interface changes often, it can be tedious to create and maintain the bindings. LuaBind [75] is a library, that automatically generates the binding code for C++ callback and provides an interface to simplify calls to Lua functions as well. The drawback is that LuaBind works only for C++ and might not be capable of solving more complicated cases. There are more alternatives, namely e.g. LuaPlus [62] or toLua [39].

3.1.6 Documentation and installation

Almost everything about Lua is simple. Its library can be built from source code (625 KB as of the version 5.3.0) without any dependencies using standard compilers. Linking the static library increases the program size by 390 KB. It is distributed under MIT license [10] which requires only a mention that Lua is included in the software.

There is a book about Lua (written by its developers), which describes the syntax and principles of the binding API and Lua itself [57]. The official site [7] contains these information online as well. There are several articles describing design decisions of Lua, history of its development and some of its inner functions [67] [59] [58] [46].

3.2 Javascript

We will introduced the history of JavaScript and the engine V8 we have selected for this work. We will talk about some of the advanced features, that enable V8 engine to be fast and examine the state of its documentation.

3.2.1 Introduction

JavaScript is a scripting language mainly focused on web. It was created as a part of Netscape 2 and then spread into other web browsers. However, the languages differed slightly in each browser - they had even different names. JavaScript is today standardized under the name of ECMAScript and its latest specification is ECMA v6 [76]. The similarity of names with Java was purely a marketing ploy [47].

There are multiple JavaScript engines, basically for each major browser there is one (V8 for Google Chrome and Opera, Chakra for Internet Explorer, SpiderMonkey for Mozilla Firefox and Nitro for Safari), but there are many more [79]. We have selected V8 from Google since it is open source and comes with good embedding API. SpiderMonkey appeared as a good candidate as well, since its speed is comparable and they both have similar features. Our decision was thus based on the embedding API, which looks more interesting for V8 for comparisons with the other APIs we test. In this work JavaScript with V8 represents a scripting language, which is highly optimized using complex engine to achieve high performance.

V8 is written in C++ and implements ECMAScript as specified in ECMA-262, 5th edition. It compiles JavaScript directly into native code and most of the today's major processor architectures are covered officially or by extensions [52] [30] [31]. It was created by Google as a JavaScript engine for their browser Google Chrome and was released together with its first version in 2008. Because of its high performance and the availability due to being open source, it was used in other projects as well - probably most notably in Node.js [78]. It is quite sizable and complex piece of software (source code alone has 28MB as of the latest version 4.9.234), however it is very fast due to the use of advanced techniques.

3.2.2 V8 hidden classes

The objects in JavaScript are created by functions. If the function is called using the keyword *new*, it is used as a constructor. The objects have no type, they consist of the properties assigned to them during runtime. Because the objects don't have static types, the interpreter cannot use many optimizations, that statically typed languages offer.

V8 uses a somewhat hybrid approach, where it allows the objects to be dynamic as required by the language specification, however during runtime it creates constructs that describe the current object type and contain optimizations. These constructs are called *hidden*

classes. The adjective hidden is used because the programmer working with JavaScript code doesn't know about them, they are used only internally [51].

The justification for the usage of hidden classes comes from the observation, that while most of the objects in JavaScript are created dynamically, after creation they actually have the very same structure. By creating an optimized code for this structure, a big part of the execution can be optimized while keeping the amount of generated code small (not having separate optimizations for each object instance).

The assignment of the hidden classes happens during object creation and modification. New hidden class is created for each new property added to the instance, thus object can jump trough many classes during its creation. However, an old hidden class remembers the step that resulted into the creation of new hidden class. If another object having this old class does the same step, the hidden class that is remembered to be the result of this step is reused instead of creating identical new one. Because of this approach, all objects created the same way end up having the very same hidden class.

3.2.3 Inline caching

Inline caching [56] is the way of generating optimized code for operations that repeat frequently. It takes advantage of the information provided by the hidden classes. Inline cache is a small native code stub, that contains the optimization. It is created during runtime, based on the information gathered during previous execution of the code.

When a property is used for the first time, V8 does standard dynamic lookup of the value, since there is no information about its location known beforehand. However, during this lookup it creates and compiles the inline cache for this access, which will be used next time. This optimized code basically checks whether the hidden class of the object is the same class this code was generated for and if it is, the optimized code is executed. If there is a mismatch, it suggests the fact that in this situation there might be multiple objects and the code is deoptimized to do the dynamic lookup again. However, the already generated stubs can be still used [40].

The inline cache is flushed during garbage collection. This way the engine gets rid of the code stubs that are probably no longer going to be used and gives the chance to create new, optimized stub again, even if this part of the code was deoptimized previously. The fact that GC could flush the cache suggests that some part of the execution finished and the new part might take advantage of the optimized code again.

Because a lot of JavaScript code might be executed only once, e.g. initialization code, the generated optimized code might not be used at all. Because of that, the code might be actually generated not the first time it is run, but only after few usages.

3.2.4 Data representation

All JavaScript objects are aligned to 4 bytes within V8. This means, that the last 2 bits of their 32-bit pointers are always going to be 0. V8 uses these bits to optimize numbers. If the last bit of the object pointer is 0 (its value is even), the previous 31 bits represent a signed integer directly, instead of a pointer address. The integer value can be acquired by simple bit shift. If the number value won't fit the 31 bits or is a floating point, it will be boxed

inside an object as a double-precision floating point compliant with IEEE 754 [83]. V8 can ensure this number format, since it directly creates the native code.

Each object consist of a pointer to its hidden class and pointers to properties (string indexed values) and elements (integer indexed values in incremental order). If the code is optimized by the inline caching, the pointers to values can lead to arrays and the access requires only an offset, otherwise they lead to dictionary [41]. There are more optimizations going on, e.g. when the array contains only double values, they are stored unboxed. If the values are mixed, double values have to be boxed.

V8 uses C types to represent its types, however it makes sure the representation is always the same on all platforms. It also contains few of its own type implementations, e.g. the small 31-bit signed integers, for optimization purposes.

3.2.5 Compilers

V8 compiles JavaScript directly into machine code, there is no intermediate bytecode representation. It actually uses 2 different compilers:

The first one, *full-codegen compiler*, processes the abstract syntax tree and generates native code using the inline caches, as was shown above. All local values are stored on stack or heap and appropriate instructions are necessary to retrieve them or store them. The compiler runs only when the function is actually used for the first time, to avoid long starting delay caused by unnecessary compilations.

After that, profiling starts to examine which parts of the code are *hot* - used often. It also gathers information about what data types are passed trough certain parts of code. Once V8 has gathered enough information, it can attempt to improve the hot code by launching its optimization compiler, called *Crankshaft* [42]. It generates high-level intermediate representation (*HIR*) first, which is in SSA form. Several optimizations like constant folding or method inlining are performed here, with the help of the type information. This form is then compiled into low-level intermediate representation (*LIR*), which resembles machine code but is still mostly platform-independent. Other optimizations are performed here and the result is prepared for allocating the local variables to registers. More about the HIR and LIR forms in general can be found here: [32].

In order to decrease startup delay, V8 also contains a snapshot feature. It can basically cache the entire heap state to drive so it can be loaded and used immediately after next engine start.

3.2.6 Garbage collection

Because a small integer values differ from the pointers in the last bit, GC cannot accidentally think these are pointers as well, as some GCs like Boehm-Demers-Weiser do. Thus it always knows which data represent the pointers and which not.

V8 implements a generational GC with 2 generations. Objects are always allocated into *new-space*, which is quite small (around 1 - 8 MB). When this space fills up, the GC starts and quickly removes the dead objects from the new-space. Objects that survive 2 GC cycles are promoted to the *old-space*. GC for the old-space is run once it reaches certain size and usually takes considerable more time than the GC for new-space.

Pointers that lead from old-space to new-space are maintained in a *store-buffer* to keep the new-space GC fast. The code that handles this is called *write barrier* is it has to be executed anytime a pointer is written [73].

3.2.7 API

The embedding API is based on C++ objects and their types. It provides handle types for storing references to JavaScript objects, which also can control their scope of existence. All values passed from C++ to JavaScript have to be wrapped in those handlers. Once the handler leaves its scope, its value can be subject to GC, unless the handler is explicitly made persistent. Templates for functions and objects can be created with custom function and access callbacks to C++. Property accesses on objects can also have their own callbacks.

V8 enables several independent JavaScript environments using a concept of contexts and isolates. Contexts represent independent JavaScript environments, however for running multiple simultaneous threads, separate isolates have to be used. Isolates can be also locked by a specific thread, so other threads cannot use it during that time.

3.2.8 Documentation and installation

Documentation for the basic embedding of V8 is available [29], however it is not always up to date, as during our research some of the code samples didn't match the actual API. The API itself changes often as well, so we recommend to always use single specific version. Another drawback is that the documentation doesn't contain everything. It is good for understanding the principles and writing small code samples, but it was insufficient even for writing the benchmarks in this work. The missing information is usually searchable in the API reference [28] or on user forums, however this state is far from ideal.

The details of V8 internals are officially discussed in videos by Google [53], in their documentation page [51] or blog posts [50], however, there are practically no articles about V8, just few that discuss similar features but in different languages [32] [56].

The building of V8 engine requires few additional steps, especially if the system doesn't already contain prerequisites. V8 is build with help of repository helper *depot_tools*, python and meta build system *GYP*, which have to be installed first. On Windows, Visual Studio 2013 or cygwin are required to build (those are the supported build options). The situation on Linux is easier, since there is no need for cygwin and the build is done using make. The files fetched by depot_tools have together around 700 MB, linking the library increases the program by 5 MB and compiled as dynamic library it takes about 4.5 to 8 MB, depending on the features selected (e.g. internationalization support can be dropped). V8 is available under the BSD license [2], which requires again basically only a mention V8 is in the program.

3.3 C#

We will discuss the possibilities of using C# as a scripting language trough Mono. We will talk about its JIT feature, garbage collector and the features enabling fast communication between C/C++ and C#.

3.3.1 Introduction

C# is an object-oriented programming language, based on C++ and Java. It was developed by Microsoft as part of the .NET initiative and later standardized by ECMA [33] and ISO [60]. It was originally developed for Microsoft's virtual machine called *Common Language Runtime* (CLR), which is used to execute all programs written for .NET. CLR is a part of an implementation of Microsoft's *Common Language Interface* (CLI), which describes the environment that allow high-level languages to be used independent of platform.

C# is a language not mainly targeted on scripting. We have selected it for this work as a representation of a language intended mainly for system programming, but leveraged for scripting. C# is however already used for scripting with notable popularity [27]. Since C# is targeted more on system programming than scripting, it has some lower-level features like static typing or bitwise operators. However, it has also a garbage collector and is memory safe. Because of that, it is lacking some of the ease of usage, but still is more user-friendly that a purely system programming language like C++. The performance is also higher than for a purely scripting language.

We have selected Mono as the environment for testing C# for scripting, especially because of its multiplatform support and C++ embedding API. Mono is an open-source multiplatform implementation of CLI based on the ECMA standard. It consists of compiler, the runtime and various libraries. It was founded and sponsored by company Ximian, later it came under Novell and today it's lead by Xamarin [11]. Mono is generic byte-code level virtual machine capable of compiling and running multiple languages, not just C#. Each supported language is compiled into Mono byte code and then ran by the generic VM.

Mono is quite big. Static linking requires a commercial license and the binary installation takes 450MB on hard drive. However, it is possible to pick only small portion of the installation based on required features. For example for the purposes of this work, only 8.5MB would be sufficient.

3.3.2 JIT

C# is compiled into a low-level but still human-readable language called *common intermediate language* (CIL), which is packed into assemblies. Mono and .NET both produce the very same multiplatform assemblies, however Mono doesn't support all the .NET features yet, so some .NET code might not work under Mono. C# is a statically typed language and thus provides a lot of information that can be used for optimization.

Mono uses a JIT compiler, that similarly like V8 generates machine code. There is an option to use LLVM instead, however we will use LLVM in chapter 4, so we chose to use the default JIT compiler. LLVM produces faster code, however it takes more time to compile and not all features required by Mono are supported (in that case it falls back to the default JIT) [17]. If using LLVM, JIT IR (intermediate representation) is generated from the assemblies, transformed into SSA form and then transformed to LLVM IR.

The JIT first transforms the CIL into its *linear intermediate representation* [16]. Then optimizations take place and appropriate local and global values are allocated to registers. Finally, machine code is produced.

Mono can precompile the code, instead of the lazy compilation that occurs only when the code is run for the first time. This feature is generally called *ahead of time compilation* (AoT). In Mono, this process creates *position independent code* (PIC) which is a bit slower compared to JIT code, but AoT can use more optimizations, since the compilation time can be longer. The drawback is that the result code is no longer portable [12].

Mono JIT is using a concept called *trampolines*. They are small pieces of native code, that can be used as stubs. E.g. they can be placed as a target of call instructions instead of the functions, that should be called, so when the call happens the trampoline executes instead. It contains code, that starts JIT, compiles target function and connects the call to the newly generating function, cutting the trampoline off. Because of this feature, functions are lazily compiled which means no time is lost on compilation code that may never be executed. There are multiple trampoline kinds for various purposes [24].

3.3.3 Garbage collection

Mono originally used *Conservative Boehm Garbage Collector* [36]. Such garbage collector doesn't modify heap in any way (like e.g. reference counting), which means it has to explicitly search for pointers. It might not clean all the unreachable objects, but it is unlikely that it would result in memory leak growing over time. This GC is still available in Mono, however recommended is their more advanced *Precise SGen Garbage Collector* [21], which will be discussed in this section.

SGen is generational and it allocates objects into several sections:

- **Nursery** All new small objects are allocated here. Nursery is relatively small, by default is has 4 MB and never grows. Whenever there is no space for new objects to allocate, quick GC cycle goes trough the nursery, frees objects that are no longer reachable and copies the survivors to the old generation. There is one exception - pinned objects. An object can be pinned explicitly by programmer using C# directive *fixed*, or if it is referenced in communication with the native code and thus its location cannot be changed. Those objects stay in the nursery and can be moved only when they are no longer pinned. In order to prevent synchronization locks during simultaneous allocation by multiple threads, each thread owns a small piece of the nursery (called *thread local allocation buffer*) exclusively.

- **Old generation** Objects that are expected to live for longer time (since they were not collected as garbage from nursery) are stored here. Major GC cycle that frees memory from this part happens less often than small and fast nursery cycles. Similarly to V8, Mono implements write-barriers to handle pointers from old generation object to nursery.

- **Large objects** Objects larger than some threshold are expansive to move, so they are allocated into their own OS pages. When they are no longer referenced, they are deleted by releasing their pages back to the OS. This happens in the major GC cycle where old generation is examined.

When GC occurs, all threads have to be stopped. However, once the mark part of GC is done, threads can be restarted. Sweep part treats data that are no longer reachable and thus it can happen while the rest of the program is still running.

3.3.4 Native code connection

CLI standard defines ways to communicate with the C/C++ code. In the standard, classic C/C++ code is generally called *unmanaged code*, while C# code is called *managed code*. The invocation of managed code from unmanaged code depends on the C# engine, since it has to be started. The CLI standard method of invoking unmanaged code from managed code is called *Platform Invoke* (P/Invoke). Since C# is statically typed object-oriented language, all external methods or structures used as parameters have to be declared within the C# code as well as within the C/C++ code.

P/Invoke uses *dlsyn* function on Unix or *GetProcessAddress* function on Windows to get the address of the target method. The target method has to be C ABI-compatible, which can be achieved in C++ by wrapping it into *extern "C" { ... }* block to avoid C++ method names mangling. Additionally, the calling convention has to be set according to the platform (e.g. *Stdcall* fro Windows or *Cdecl* for Unix). In C# the function has to be declared with *DllImport(...)* attribute specifying the external library that contains the target function (or the fact that the function is within the same executable). Then this function can be normally called from C#. The drawback here is that the data passed have to be copied or even marshaled, if the formats aren't compatible [19].

Mono offers another way to call underlying C/C++ code that doesn't follow CLI standard and was meant to provide fast interoperability with its own libraries. Programmers can however normally use this feature, which is called *Internal calls*. Internal calls work directly with Mono inner data representation and Mono API is required to manipulate them. E.g. you can get a pointer to the actual char data of Mono string trough the embedding API. By accessing the data directly the overhead of communication between C/C++ and C# is significantly lowered [82]. However, the C# code becomes dependent on Mono, since instead of DllImport attribute, Mono specific one is used. Since we are using Mono for all platforms, we chose to use this faster approach.

C# method can be invoked using Mono API, however, this approach is slower compared to using *unmanaged to managed thunks*. Using thunk will create a custom trampoline from unmanaged code to managed code for the particular method signature that is being invoked. When this code is called, the thunk makes sure that the target function is compiled and than replaces itself with direct call to the native code generated from C#. No parameter validation occurs (except for mandatory MonoException pointer), it must be ensured by programmer. Because of that, once JIT is compiled, the transition is very fast. However on parameter mismatch, the program might run into unexpected issues.

3.3.5 Threads

C# has a direct syntax support for threads and Mono uses system threads to implement this feature. Because threads are OS dependent feature, Mono offers special interface called *io-layer*, that has to be specifically implemented for target OS. This layer provides all OS

specific features including network sockets, etc. Mono runtime has locks on several levels (global, domain, JIT code, etc.) working in hierarchy [23]. For interprocess communication, shared file mapping across multiple Mono processes is used.

3.3.6 Documentation and installation

Dynamic compilation of C# sources into assemblies is not trivial, since Mono's C++ API doesn't support it. The most straightforward solution is to run mcs compiler (part of Mono) using system call from C++. More elegant and portable solution is to write your own compiler in C#, which requires only several lines of code, and distribute it as an assembly with your program. The compiler can be then run from C++ as any other script. We have however encountered undocumented errors using this approach, and since we mainly focused on the performance of the already compiled script, for the purposes of this work we used the mcs system call which worked well enough. Compared to the other languages, we observed that the compilation of C# assemblies takes considerable time (even seconds for our simple scenarios).

The state of the documentation [15] is similar as for V8. There are official basics and the API [13] available online but they are insufficient and sometimes inaccurate. The details of Mono implementation are mainly discussed in their website [14] and there is also a book available [45].

Mono doesn't officially support 64-bit version for Windows [20] and static linking requires commercial license [18], so we haven't been able to find a different workaround, than using 32-bit version dynamically. It can be installed (takes up to 450MB on hard drive) or compiled from source. However, it is possible to pick only small portion of the installation based on required features. For the purposes of this benchmark, only 8.5 MB were required, which could be most probably further reduced [22].

Chapter 4

Custom Language

We design our own simple scripting language called NativeScript in this chapter. We implement an engine to run it using two different approaches (AST interpretation and JIT using LLVM), including API for communication with C++. We also discuss the motivation to create a new scripting language and the details of NativeScript implementation and libraries used. NativeScript isn't as rich as the other languages discussed in chapter 3, although it is Turing complete [74]. It is a highly specialized language for certain tasks, specifically double operations and communication with native code. Its performance is compared to the other languages in chapter 6 together with the benefits and caveats of the idea of implementing a custom scripting language.

4.1 Motivation

Developers can sometimes decide to create their own scripting language/engine. The motivation behind it can be various:

- **Specific features** The software might require some specific features and languages that provide them cannot be used from different reasons.
- **Customization** The software is expected to grow or change greatly and the selected scripting language might need to be altered accordingly.
- **Performance** The scripting languages is required for only very specific tasks. Because of that, great part of the existing solutions would be unused and even slowing the critical features down (e.g. dynamic types where only single type is required).
- **Fun** For some people, developing their own language might be an exciting experience and they are willing to invest the extra effort necessary to create it.

The important fact about developing a custom language is that the task is not simple. As you can see from the history of the scripting languages discussed in chapter 3, it took a considerable amount of effort and time to get the languages and engines into the current state. It is possible to create a custom programming language with a compiler or interpreter

in few days, however the result is very likely to be poor. The design of the syntax should enable fast and convenient development and maintenance of the code and the implementation should provide required performance.

Most likely there is already a suitable scripting language that has all the required features, since somebody most likely already faced the same issues before. Thus before making the decision to create a new language, verification of the reasons should be made. E.g. if the problem is performance, the software or language should be evaluated or profiled first and the newly designed language should be proven to really make the difference. Also, the developers (especially the new ones) will have to learn the new language and there might be other difficulties or risks, e.g. security.

If it is really necessary to implement a custom scripting language, the developers should posses at least some theoretical knowledge of computational theory and terms like *Turing completeness* or *halting problem* should be known to them. Luckily, there are libraries and frameworks that can provide great help when composing a new language and those should be evaluated (we use some in this chapter).

4.2 LLVM

The main framework we use is the LLVM [65]. LLVM stands for *Low Level Virtual Machine* and it is basically a compiler framework that can produce an optimized machine code based on a high-level information provided. LLVM contains a subproject called *Clang*, which is a C/C++/Objective-C compiler based on LLVM, that is comparable (if not better) with compilers like *gcc* and is widely used by Apple [68].

LLVM works with code in a form called *intermediate representation* (IR). The main advantage is that this representation is language and platform independent. It is slightly richer than assembly languages, so it keeps things like type information and the control flow and data flow representations. However, it doesn't contain high-level constructs like classes, inheritance or exceptions. These properties make it universal for description of practically any language, while still being able to keep enough information for optimizations. However, some of the languages, especially those that contain higher-level constructs like e.g. Java probably won't be as efficient with LLVM, since it won't be able to optimize them as well as their specialized compilers.

LLVM IR is in *static single assignment form* (SSA form) which allows only a single assignment to each variable. This form is suitable for many optimizations which can be performed faster. For imperative languages that require mutable variables, LLVM suggest an approach of using stack allocated memory, which doesn't have to be in SSA form. LLVM can automatically optimize this representation into registers, which makes the generation of IR for imperative languages more convenient [5].

Once the code is in IR, optimization passes can be run. LLVM implements many of them and provides a pluggable interface for implementation of custom pass. Because of this, specific optimizations for given language can be selected to customize the trade-off between code generation speed and code execution speed.

The optimized IR can be executed using an *execution engine*. There are interpreter and JIT implementations available in LLVM, although there is also a pluggable interface

for custom engine implementation. We are using LLVM JIT to produce machine code in NativeScript. Machine code can be produced into memory and dynamically linked. The C++ API of LLVM enables the pointers to the generated machine code to be retrieved, and also to pass pointers to custom C++ functions, which will be called from within the machine code. Because direct addresses are passed, the transitions between C++ code and script are very fast.

There is a lot of documentation on the official web page of LLVM [6] describing the principles and usage of LLVM. There are also guides for writing pluggable modules (e.g. custom optimizing pass).

There are already scripting languages written using llvm [49] [35]. We wanted to implement our own more for the purpose of showing the process rather than to use the actual product.

4.3 Features included

Because the design and implementation of a full scripting language is a time consuming process, NativeScript contains only a small amount of features. We implement just enough to make the language Turing complete and to be able compare its binding speeds to the other languages from chapter 3.

NativeScript's main building block is a function. Every source file is simply a list of functions, which can be either defined by specifying a body, or just declared to be bound to C++ callback later. Those functions can be then invoked from C++. Every function returns a single value and accepts 0 to n arguments. Functions are distinguished purely by their name, there is no polymorphism.

There is only a single type available, representing a 64-bit floating point number. Because of that, types aren't involved in the syntax at all, since all variables are automatically doubles. This simplifies the language greatly, while not having any performance cost for dynamic types. Types are known by the compiler automatically, so it can generate specific instructions directly. However, the list of possible features of NativeScript is significantly reduced by this. NativeScript in this work represents a custom language tailored for a specific task (double operations), so this is in fact an optimization.

Variables can be used for assignments and as a part of expression. Every variable is automatically declared first time it is assigned, so there is no syntax for declarations. Functions parameters are the only input variables usable within the function - there are no global variables. NativeScript has *static scoping* [77] in its simplest form, since the functions cannot be nested.

NativeScript supports loops and conditions, which makes it (together with the variables) Turing complete [37]. It also supports binary operators $+$, $-$, $*$, $<$ and $==$. Value 0 is considered to represent *false*, any other value represents *true*.

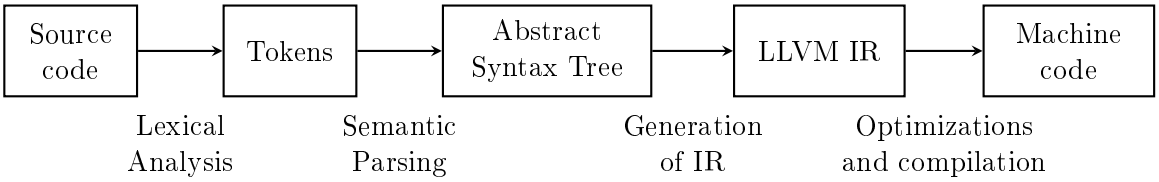
The library provides C++ interface capable of parsing source code from file or string. The result script then can be either interpreted by iterating its AST or compiled into native code using LLVM and executed. For both versions callbacks can be specified and any script function can be invoked. NativeScript is memory and type safe and the only runtime error

possible is caused by overflowing the double type or stack during deep recursion. The errors during parsing and compilation are printed to *stderr*.

4.4 Implementation

In this section we closely look at the implementation details of NativeScript. It consists of a lexer and a parser which produce an AST, that can be directly interpreted. The compilation process generates LLVM IR, runs optimizations and generates machine code. The compilation steps and subproducts are depicted in figure 4.1.

Figure 4.1: Compilation process in NativeScript.



4.4.1 Lexer and Parser

Lexer is a tool that reads a source code and translates it into tokens, which represent atomic building blocks. Parser then takes those tokens and generates an AST based on the syntax rules of the language. We use lexer and parser generators Flex and Bison [66] for those tasks.

Flex is a library capable of generating a C source code of lexer defined by a configuration file. The configuration file contains rules which consist of a regular expression and an action executed on match. The action usually consist of returning the appropriate token, however, arbitrary C code can be specified. Arbitrary C code can be also put to the initial and the final part of the generated file. This feature is used mainly for including necessary files, declarations or custom function definitions. Apart from parsing all basic tokens, notable features of NativeScript's flex configuration file are line counting for better error feedback and definition of functions for switching between file and string parsing.

Bison works on the same principle as Flex. Its configuration file is however slightly more complex. It is meant to contain a specification of a *context-free grammar* [48]. The rules of such a grammar consist of terminal and non-terminal symbols. At the beginning of the configuration file, union type is defined, to cover all the possible types that can represent symbols (both terminal and non-terminal). Then terminal symbols are defined together with their types. The default type doesn't have to be specified nor here, neither within the union - it is an enumeration describing tokens without any metadata. Then, non-terminal symbols are defined. Finally the rules of the grammar are defined using previously defined symbols. For each possible output of a rule, arbitrary C code can be executed. This is usually used to create the AST nodes and connect them together. The nodes be used to directly represent the non-terminal symbols and thus accessible as metadata during the execution of the rules.

Our grammar has a list of functions as its root. Each function consists of a declaration and optionally a body. The body can contain arbitrary expressions. For simplicity, every

statement in NativeScript is an expression, so e.g. condition is in fact a ternary operator and the loop always returns 0. The value of the final expression of the body (annotated by keyword *return*) is used as a return value of the encapsulation function. The precedence of the binary operators is defined directly in the configuration file. This description of the grammar is simplified, since we didn't want to sink into tedious details.

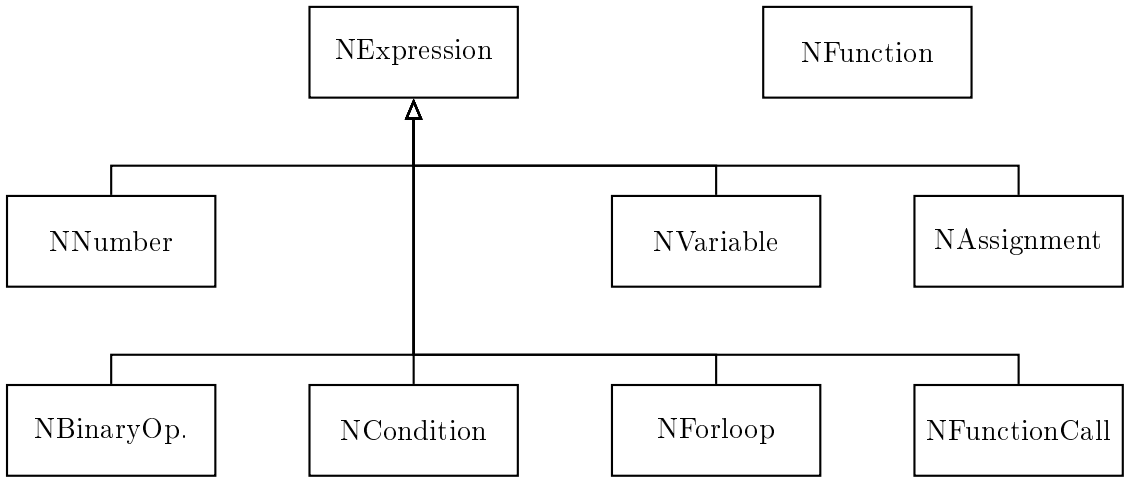
4.4.2 Abstract syntax tree

Abstract syntax tree represents the structure of the program within the memory. Its nodes represent operations and commands, whose operands are represented as nodes as well. Using this approach the tree can be iterated from some node to visit all parts of code that participate on this node's execution.

We use C++ classes and inheritance, specifically the composite pattern, to represent the nodes of our AST. The two root types are *Expression* and *Function* (representing declaration, not call). All other types are direct extensions of Expression, as can be seen on figure 4.2. The only logic implemented within the nodes is for its creation within constructor. Thus, the nodes just represent the structure and its data. For iterating the AST (to generate IR or interpret the code) visitor pattern is used. This slightly lowers the performance, however it makes the code maintainable, reusable and easily extendable.

The AST is produced when the source code is parsed. It is stored within the *ns::Script* object of the C++ interface of NativeScript. It can be further used to create an interpreter or a compiled code.

Figure 4.2: Node hierarchy of NativeScript's AST.



4.4.3 Intermediate Representation

To generate the machine code using LLVM, IR must be generated first. In LLVM, IR is generated into top-level containers called modules. A module holds all the memory of its IR and can be compiled separately. In NativeScript, for each parsed string or file, new module is created to contain the IR. Then, function pass manager is created and several optimization

passes are assigned to it. Finally, for each function in AST root, IR is generated by using visitor pattern and function pass manager is executed on the result. All of this happens in our *ns::Executor* class.

To help with IR emitting, LLVM provides convenient class *llvm::IRBuilder* which remembers the current position within IR and inserts instructions based on methods called. So e.g. instruction for floating point number addition is created by simply calling *builder.CreateFAdd* and passing LLVM representations of operands as arguments.

IR instructions are always emitted to blocks. Blocks are simple containers that hold instructions in serial manner. They can be assigned to functions, or e.g. used as targets of conditional branching. We also use them in loop representation.

The basic class representing an SSA register is *llvm::Value*, however since our language is imperative, we allocate stack memory and then let LLVM optimization pass *mem2reg* to convert it into registers. We keep all variable representations for the current scope in a map, so they are accessible during the visitor traversals. LLVM can optimize stack allocations to registers only under certain conditions - the only limitation relevant for NativeScript is that all allocations must happen at the beginning of a function (its initial block), before it can e.g. branch because of condition. To achieve that, we simply move all declarations at the very beginning of the function and don't allow any variable shading.

Similarly like local variables, we hold all declared functions in a map so they can be used as targets of call instructions. We don't need forward declarations in source code, since we first declare all functions and then loop over them again to generate their bodies. If the declaration is external, the function doesn't have the body and has to be bound to callback using C++ interface, before the machine code is generated.

4.4.4 Compilation and execution

Once the IR is generated and optimized, external function declarations have to be bound to callbacks. This is done trough *llvm::ExecutionEngine* class. The ExecutionEngine is created trough builder class by passing it a module. Details about target architecture can be specified, otherwise they will be set based on the module passed.

We use MCJIT as the implementation [4] of ExecutionEngine in NativeScript. It is capable of emitting machine code into memory and dynamically linking it to the current executable. It is more a dynamic machine code generator than an actual JIT. It compiles whole modules, instead of individual functions as they are called. The modules can be recompiled or linked together with other modules [64], however, for the purpose of this work, single module compilation is enough.

For MCJIT code generation on Windows, we had to change the format of the generated objects to ELF, since the dynamic linker for the default COFF format (32-bit) was not implemented in LLVM version 3.7.0.

Before the code can be generated, all declarations must be resolved. This can be done by passing C++ function pointer to ExecutionEngine, together with the LLVM object, that represents function (*llvm::Function*). Once all callbacks have been assigned, the ExecutionEngine can be finalized, which causes it to generate actual machine code. After that, pointers to addresses of the generated functions can be retrieved and used to invoke those functions.

4.4.5 Interpretation

The interpretation of NativeScript is simpler, but also slower. The interpreter executes NativeScript by evaluating the AST tree. No finalization is necessary and callbacks for script external declarations can be passed anytime. The interpreter traverses the AST tree using a visitor pattern, which causes it to use an extra function call for each node, however enables the AST tree to be universal and reusable.

When a root function is called, the interpreter stores its parameters in a map representing local variables and then evaluates its body expressions. Each expression evaluates its subexpressions and the result of the last expression in a body (after keyword return) is returned.

The external callbacks are stored in a map indexed by their names and the interpreter looks them up when an external function is called. Callback functions must have specific type. They have 2 parameters - argument count and array of doubles representing the arguments. This universal declaration can be dynamically called while representing any NativeScript declaration.

The main advantage of the interpretation is that it doesn't cause any compilation delay and has no other dependencies, thus can be run on any machine that can compile its C++ code. LLVM today contains many target platforms for generation of machine code and since its spread and frequent usage it is unlikely that some major platforms won't be supported in the future. However, the dependency is there.

4.5 API and guide

In this section we describe NativeScript's syntax and its C++ interface together with explanations what is happening when certain API functions are invoked.

4.5.1 NativeScript syntax

Source code of NativeScript is basically just a list of functions. There are 2 function types:

- **Function with definition** This is the function executed by NativeScript. It consists of its name, parameter names and body. Body consists of expressions separated by a semicolon and the last expression within a body has to start with the keyword *return*. A simple example can look like this: `add(x,y) {return x+y;}`.
- **External declaration** These constructs are used to represent an external function, that is implemented in C++ and can be called from within NativeScript. External declarations don't have a body and begin with a keyword *external*. A simple example can look like this: `external add(x,y)`.

No forward declarations are necessary. The function body can contain any number of expressions. Variables don't need to be declared, it happens automatically the first time they are assigned. Variable and function names can be any alphanumeric strings containing

underscores, however, the first symbol cannot be a digit. The reserved keywords are *if*, *else*, *external*, *for* and *return*. NativeScript is case sensitive.

There are math operators +, − and ∗ and boolean operators < and ==. All values are floating point numbers so false is represented as 0 and true as any other value. Multiplication has precedence, however brackets (and) can be used. Assignment is done using operator =, e.g.: `varToBeAssigned = valueToAssign;`. Number literals can be written using numbers and optional point symbol. All numbers have to begin with a digit (leading point symbol is not allowed).

Condition is implemented as a ternary operator using brackets and keywords *if* and *else*, e.g.: `if (a < b) a else b;`. The for loop is also an expression returning always value 0. It has the following syntax: `for (loopVar = initValue, loopCondition, additionToLoopVar) expressionToExecute;`. The value of expression *additionToLoopVar* is added to *loopVar* at the end of each loop execution. Blocks of code aren't implemented, so there can be only a single expression in the loop or the condition. However, this expression can be a function call and functions can contain multiple expressions. Function calls are represented simply a name and a list of arguments, e.g.: `add(1,2);`.

The syntax is very simple and there are probably several features that could be added to make it more convenient - like more operators, blocks of code and some kind of arrays. However, for the purpose of this work, it is sufficient.

4.5.2 C++ API

The C++ API consists of 3 classes:

- **ns::Script** This class represents a parsed source file. It can be created by static calls *ns::Script::parseString* or *ns::Script::parseFile*. Those calls return a pointer to the newly created *ns::Script* object. This object already contains an AST produced by flex and bison.

- **ns::InterpretableScript** This class represents a script, that can be run using interpreter. It can be created by calling *getInterpreter* function on the *ns::Script* object. Callbacks for external declarations within NativeScript can be specified by calling function *bindExternal*. Type *ns::ExternalFunction* is a required function type for all callbacks. NativeScript function can be executed by calling function *runFunction*.

- **ns::CompiledScript** This class represents a script, that can be ran using LLVM. It can be created by calling *compile* function on the *ns::Script* object. During this call, an IR and ExecutionEngine are created, however no code is compiled yet. This object has 2 states - initial and finalized. When created, it in its initial state. In this state, function *bindExternal* can be called to bind C++ callbacks to external declarations within NativeScript. By calling function *getFunction* for the first time, this object goes into finalized state. This transition means generation of the actual machine code. After that, the *bindExternal* function cannot be called anymore and if there were any unresolved external declarations, the machine code generation will fail. The function *getFunction* returns a function pointer that can be cast to match the declaration within

NativeScript and called directly to invoke the function. There is also debug method *dumpIR*, which prints generated IR to standard output.

All those 3 objects are independent on each other. That means that once e.g. *CompileScript* object is created, its original *Script* object can be destroyed or used to construct other classes. In order to avoid copying of the AST, each function in the root collection is held using *std::shared_ptr*. Once created, AST nodes are never altered, so the function nodes of AST (and their child nodes) can be shared.

For maximal conectivity, no standard library constructs are passed trough the interface. This is especially convenient for cases where the NativeScript is compiled with different standard library implementation than the application, that links it. Also, NativeScript handles all of its memory itself. That is the reason why the object constructors and destructors are private. To destroy any of these objects, simply call its method *free*;

Chapter 5

Performance Testing

To test the interoperability of selected scripting languages, we developed a testing application in C++. In this chapter we discuss the testing application details, the environment and tools used for running the tests and the various scenarios and features tested. The results of the tests are presented and discussed in chapter 6.

5.1 Testing application

Because we aim to test the interoperability of scripting languages with the native code, we developed an application in C++, which contains the bindings to the selected scripting languages. The specific test of given language binding can be executed by running the application with specific arguments. The application contains unified measurement and scenario logic in order to ensure the testing environment to be identical for all languages.

User guide for the testing application is in appendix C.3.

5.1.1 Architecture

The application is designed so that the additional languages or scenarios can be easily added. The common logic of scenarios, argument processing and measurement, which is independent of the scripting language, is implemented in separate files and API is provided. The initialization and convenient methods for a specific scripting language are implemented separately as well. Each test case then specifies its name which is used as an application argument to run it. This approach greatly reduces the code duplication and ensures the scenarios are ran the same way for all the scripting languages. More, the implementation of specific test cases then mostly uses those API calls which causes them to be simpler and more readable. We are aware that this approach introduces the overhead of native function calls, however this mostly exists outside the measurements. The overhead which impacts the measurements is expected to be minimal, since all scenarios share it and the function call count is minimal and clearly apparent from the code.

In order to avoid hidden overhead, no inheritance is used and global variables are used instead. However, the whole application is divided into namespaces by scripting language and scenario in order to avoid intervention of the global variables. Global variables are used

only when necessary, e.g. when the script calls native function and some context is required in its body. We carefully identify the overhead during profiling in chapter 6.

5.1.2 Measurement tools

Measurements for CPU time and wall time are both implemented. By default, wall time is enabled, but this can be changed in file *settings.h* using preprocessor macros and recompiling the application. We chose to use preprocessor macros here instead of application arguments to simplify the usage of the testing application. We prioritize the wall measurement because it is more precise on Windows. The difference of both measurement results is in our case less than 1% (mean of all differences was 0.8%, standard deviation 0.75%), thus we consider the difference irrelevant.

In all test cases, only operations which are necessary for every script invocation are measured into the result time. We exclude the script initialization and compilation, but we include the initialization of the script call arguments or the processing of the return value. We measure the time before and after the measured section and compute the difference. There is some known overhead of our code, e.g. counting the results to verify the correctness of the result and prevent optimizations that could occur because the results wouldn't be used. We profiled the application and discovered the size of this overhead. It is discussed in section chapter 6.

For measuring the CPU time on Windows we use *GetProcessTimes* function, since we weren't aware of a better option. It measures time accurately on all circumstances, even e.g. core switch. The drawback is that the results are based on sampling and this information is updated only 64 times per second. Thus the precision is not very high. However, the tests are designed to take several seconds on average and never less than $200ms$, so this lack of precision shouldn't bias the results greatly. However, because of this, we prefer the wall measurement.

For measuring wall time on Windows we use *QueryPerformanceCounter*. On Windows 7, the measurement has very fine precision (in our case the precision was under $1\mu s$, can be discovered by calling *QueryPerformanceFrequency* function) and is synchronized across cores as well. We use it as our primary measurement tool.

On Linux we use function *clock_gettime* with clocks *CLOCK_PROCESS_CPUTIME_ID* for CPU measurement and *CLOCK_MONOTONIC* for wall time measurement. The CPU time in this case might be biased if the process is transferred to another CPU. It depends on the specific implementation of the clock. The default measurement type is wall time, so we recommend using the CPU timer on Linux with caution, although the inaccuracy shouldn't be significantly big.

5.1.3 Library configurations

For all scenarios we use the same versions and configuration of all libraries:

- **V8 4.4.9.1** It is currently the last version that doesn't cause segmentation fault on Linux for our compilation settings. To lower the size of the library, we use it without the internationalization support, since it is not needed for the tests. We link V8 as

a shared library, to avoid mismatch in settings in the rest of the code (e.g. standard library type). For all script runs, we use a single shared context. Since the scripts don't use any global data, context sharing doesn't cause any trouble - it is a suitable optimization. However, for general usage separate contexts might be needed, which will most probably cause additional allocations for each script run.

- **Lua 5.3.0** The newest version available at the time we were construction tests. We link it as a static library, since it is very small and is not made by default for linking as shared library (additional effort would be needed).
- **Mono 4.2.1.102** The latest stable version. Because 64bit version is not supported for Windows, we were forced to make the whole testing application 32bit. Licensing allows to use Mono freely only as a shared library, so we simply use the downloaded binaries. We ran all tests for both Mono's garbage collectors (Boehm and SGen) and observed no significant difference (under 2% in all cases).
- **LLVM 3.7.0** It was the latest version during the creation of this work. We believe we have discovered a bug in version 3.7.0 that caused the library to crash on Windows. We made it operational by adding one line into one include file. Details are in appendix D.
- **SDL 2.0.3** SDL is not very important for the testing itself, it is used just for the visualization of point simulation scenario. We linked it as a shared library.

5.2 Environment

Each test was ran in the same environment as a separate process, one after another. The environment was 64-bit Microsoft Windows 7 Home Premium Service Pack 1, processor Intel Core i5 M450 2.4 GHz with 8 GB RAM 1067 MHz. The benchmark was compiled using Microsoft Visual Studio Community 2013 as 32-bit application in Release configuration. Power settings were set to no processor limitations and only few other critical processes were running during the tests.

The whole testing application is compilable and runnable on Linux. However we were able to run the test only on virtualized Ubuntu 14.04 and the results differed significantly from the Windows results. Since some operations on this virtualized OS were generally unnaturally slow and some ran at normal speed, we couldn't consider these results as reliable. Thus we show and discuss only results generated on Windows.

5.3 Profiling

To discover which parts on engines cause performance costs under certain circumstances, we profiled some of the tests. We used AMD CodeXL version 1.9.10103 for time sampling on Windows. We profiled the expression and callback scenarios for the minimal and maximal parameter counts. The accuracy of results is only moderate, since the method itself is relatively inaccurate and the internals of the engines can be quite complicated. E.g. Lua has its virtual machine implemented with heavy usage of macros, which is quite difficult to profile.

The execution of languages compiled to native code couldn't be profiled in detail, since the code was dynamically created during runtime and no symbol information was available. However, AMD CodeXL was at least able to identify this code as unknown module and count samples for it as a whole. Thus we were able to see which portion of execution happened within this module. Based on the implementation details of languages discussed in chapter 3, we expect this code to be mainly the actual script execution. We were also able to track the other support features like garbage collection or call and parameters checks for some cases.

The profiling results help to explain the results of the tests and should be taken as suggestions and highly probable reasons behind the results. However, the testing results should be taken as the primary source of information, since unlike the profiling they can be measured precisely.

5.4 Scenarios

There are 3 main testing scenarios, each targeted to test different part of the interoperability. In this section we describe each of them including the features they are supposed to measure.

5.4.1 Expression evaluation

This test measures mainly the function call from C++ to scripting environment with varying number of parameters. The script executes a simple evaluation of few variables using math operations and returns the result. Each script is called 2 000 000 times (each time using different values) and total execution time is measured.

The expression pattern used is always the same, to eliminate the bias caused by mathematical operations: *(v*v)+(v+v)*(v-v)*, where *v* represents the variables. For each language, 2 to 6 parameters are passed and assigned to the expression so that optimization by rearranging the expression doesn't happen. Using this approach, we are able to measure the impact of sending different number of parameters. However, parameters have to be passed in order to prevent the script to return a constant value, which could be also optimized. The cost of a call without parameters is only extrapolated and assumed from profiling.

The expected overhead caused by test implementation consist of the loop for repeated script execution, calculation of the input parameters and adding the result to a single result variable. In some cases the generated parameters have to be put to an array to be able to be passed to the script. This is not counted as overhead, since it is a necessary part of invoking the C/C++ interface. All these overheads showed to be very small and actually significantly altering the results only for script executions with speed close to the speed of C/C++. More importantly, for all languages these overheads were the same.

To show how significant impact even simple optimization can have, we show the results with and without the basic performance recommendations. For JavaScript (V8) and Lua we wrap the expression within a function and call the function instead. The local operations in these languages are significantly faster than global ones [3] [46]. The function wrap for both Lua and JavaScript can be done automatically right before compilation so there is no API difference for the rest of the code. For C# we use generated direct C++ function (called *thunk*) instead of normal method invocation. Thunks require to just define a C++ function

interface and use it to cast the function, which might be actually more convenient than doing generic calls every time. For NativeScript we use the interpreted version as the naive case and the LLVM compiled version as the optimized case.

An interesting special case can happen when V8 receives integer values as parameters, even if they are passed trough the floating point API as type double. V8 is capable of recognizing such values and runs significantly faster. Other languages showed no significant difference in performance results, thus we show only comparison for V8.

For the optimized case of Mono and NativeScript the execution was so fast, that the measurements became inaccurate. For this reason, we executed 100 times more expressions and the measured times were divided by 100. This created slightly different testing environment for those cases, because e.g. the values of the variables passed to the expressions were different, since they all had to be distinct. However we believe that these differences are negligible and unimportant for the final results.

5.4.2 Callback function

By callback function we mean a call from scripting environment to C++. To measure a performance of such a call, we run a script, which has a loop with a simple callback. Callback can have from 0 to 4 arguments which are all simply the current value of the loop iteration variable. On the C++ side, all parameters are retrieved and collected to verify all the callbacks we really executed and all parameters were sent with correct values. We ran each callback 50 000 000 times.

The expected overhead of this scenario is the loop execution within the script, the collection of the parameters on the C++ side and the initial script call. The initial script call is negligible compared to the high number of callback calls. The collection of parameters on the C++ side was examined by profiling so this part of the overhead can be estimated. The overhead caused by the execution of the loop creates the biggest bias for this scenario, since for some engines it can be hard to be estimated by profiling. However we believe that this measurement is still more accurate than measuring a single callback separately.

5.4.3 Point simulation

This is a model scenario for more realistic scripting application. Its purpose is to show performance of the languages for combined binding usage and various optimization attempts. The test represents a simulation of movements of 400 points in a 2-dimensional space. The behavior of the points is scripted and thus in each of the 1000 frames there is a script ran for each point. The script inputs are a current point and a lead point structures (wrapping positions and velocities) which contain getters and setters implemented as callbacks to C++. The script computes the distance between the current point and the lead point and updates current point's velocity and position. The lead point moves in a circle in the center of the space to ensure the other points keep moving. The measurement starts after initialization, right before the first frame execution, and ends after the last frame execution. The scenario can be visualized and the lead point position can follow the mouse to see the scripted behavior. However while measuring, the rendering and all unnecessary code remains inactive to achieve accurate results.

Since the point parameters are passed to the script as structures with methods, this scenario is not implemented in NativeScript. Implementation of structures with members would imply adding types to NativeScript and many other features, which would make it way more complicated and it would become a project out of bounds of this work.

The scenario was run in 3 different configurations for each scripting language and once in C++ for comparison. The configurations show the performance of different approaches and optimization attempts:

- **naive** All of the point behavior is scripted and getters and setters are used for all value accesses. This is the most straightforward implementation of the behavior.
- **half delegated** The computation of the point distance and updating the velocities is implemented within the script the same way as in the naive approach. The update of point positions is however implemented in C++ and invoked from the script using callback (with point as an argument). Thus approximately half of the script execution happens within the script and the other half within C++. The results of this scenario should reveal the impacts of an optimization attempt to delegate common and rarely changing parts of logic to C++ for performance improvement.
- **optimized** All of the behavior is scripted, however the callback count is minimized. Each getter is called once at the beginning of the script and its result is stored to local variable, that is used for the computations instead. Similarly, at the end of the script, each setter is called once to pass the final results. This scenario should imply when and how much it pays off to limit the usage of getters and introduce additional temporary variables.

Chapter 6

Results and Discussion

In this chapter we present the results generated by our testing application for each scenario and its configurations. We discuss the reasons for these results based on the languages study from chapter 3. We also compute the approximate cost of binding call invocation and parameter passing using the results and profiling of the engine, trying to exclude as much overhead and bias as possible. In the end, we discuss advantages and disadvantages of specific languages and for which applications they are suitable.

In this section we present the results for each scenario, discuss it and use it to compute the approximation of binding costs.

6.1 Expression

This scenario measures the performance of invoking script execution from C/C++. Figure 6.1 contains results produced by our testing application for the expression scenario. Figure 6.2 contains the same results, however the chart is zoomed in so the differences between very fast executions were apparent.

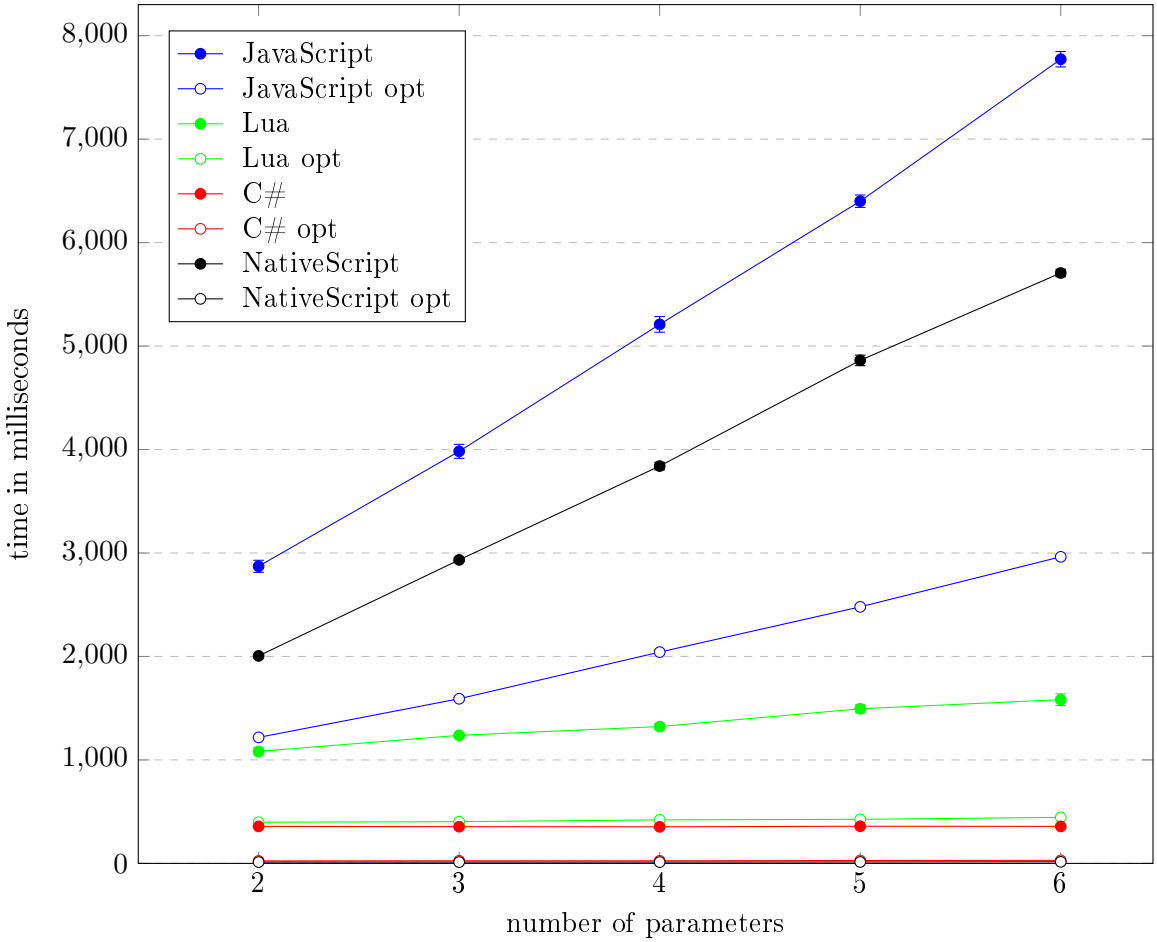
6.1.1 Result explanation

The dependency of time execution on the number of parameters seems to be linear for all cases. The passing of parameters to C# and the compiled version of NativeScript seems to have almost no impact on the execution speed, since they are simply passed by value without any marshalling or checking.

Lua transforms the parameters into its union structure, however from the results we can see this operation is quite fast. In the unoptimized version, Lua assigns the values into the global table, which takes considerably more time than simply setting the local registers in the optimized version. The execution of Lua script is also faster when using local registers, than accessing the global table for each variable.

The unoptimized version of C# did considerable amount of checks for each call, thus being slower than the optimized version, which simply called its native compiled code by address directly from the C/C++ side. Profiling showed, than the time of the execution itself for both C# versions was similar.

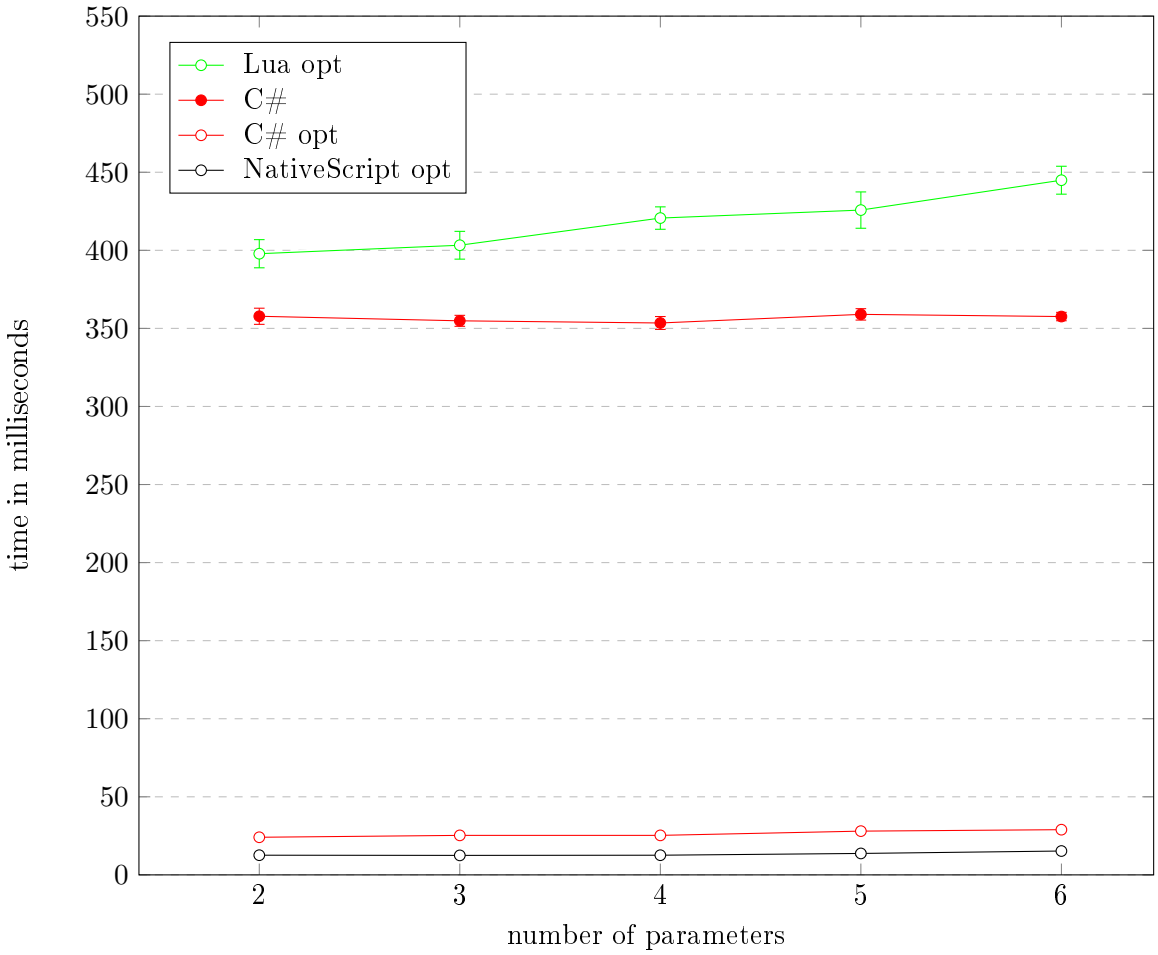
Figure 6.1: Evaluation of 2 million expressions with varying number of parameters.



The interpreted version on NativeScript had a big performance cost for parameter manipulation, since it used a map indexed by string (parameter name) for value lookups. Profiling showed that 85-90% of the execution was spent on the map lookups and saves. The performance cost of using the visitor pattern was around 3%.

JavaScript showed to be the slowest language for the transition from C/C++ to script execution. Its naive implementation uses property set on the global object, which is very slow. We assume this has a connection with V8's hidden classes system and inline caches, since these constructs are used for properties and need to be set. Significant execution time was also spent on allocation and garbage collection. The optimized version needed around 100 times less work to be done for each parameter, although allocation and GC cost was almost the same as for the unoptimized version. However when the values passed were integers, the allocation and GC took approximately 10 times less time. It appears to be caused by the V8's optimization of storage of small integers, that are saved directly in the memory that is normally used for pointer to object, that boxes the number. The comparison of JavaScript results for floating-point values and integers is in figure 6.3. Profiling showed that the overhead of the call itself required also a significant amount of checks and JavaScript

Figure 6.2: Evaluation of 2 million expressions with varying number of parameters (zoomed).



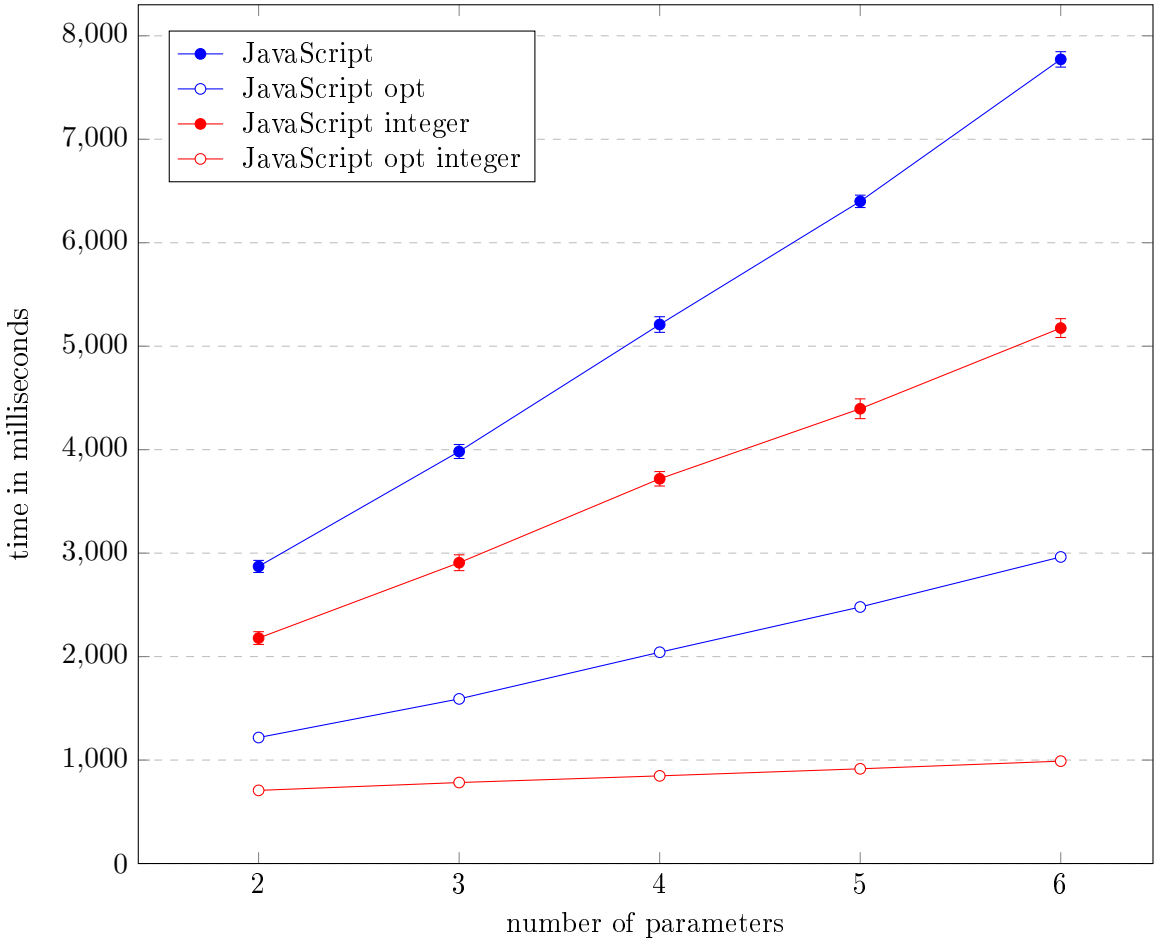
thus ended up having the slowest call invocation across all the languages tested (even for the optimized version).

6.1.2 Cost computations method

From the measured data, we computed the cost of the pure call from C/C++ to scripting environment and the cost of each added parameter. The results are in tables 6.1 and 6.2. We used two independent methods to compute these numbers:

- **Profiling** We profiled the tests for minimal and maximal number of parameters and found out what portion of the execution time was spent in certain areas. Then, we multiplied these fractions with the respective execution times we collected from the tests to get time information and averaged the values for each language configuration. This method had the advantage that we could subtract known overheads, however the precision was based on our assumptions of what parts of the library code were responsible for specific features.

Figure 6.3: Evaluation of 2 million expressions with varying number of parameters for V8 using floating-point values (default) or integers.



- **Test results** We computed the values from the measured data itself. For the call cost we simply linearly extrapolated the chart lines to get values for 0 parameters. We computed the extrapolation for all pairs $(2, 3..6)$ of parameter counts and averaged the results. This method is based on real and exact results, however it assumes, that the dependency on parameter counts is linear even for 0 and 1 parameter.

For almost all cases, the results of both methods were very similar. The presented numbers represent a single call.

6.1.3 Computation of call invocation cost

In table 6.1 we add a column with profiling data without overhead caused by our measurement and script execution itself. The script execution overhead was generally quite high, since expression evaluation took place.

We present the actual call cost of optimized configurations for NativeScript and C# to be 0, since from the facts mentioned in chapters 3 and 4 it appeared it consisted only

Table 6.1: Speed approximation of the call invocation from C/C++ to script (nanoseconds).

	extrap 0 par	profil 0 par	profil no over
NativeScript opt	6	7	0
C# opt	11	13	0
NativeScript	72	138	37
Lua opt	190	190	59
Lua	406	458	70
C#	180	178	134
JavaScript opt	198	215	176
JavaScript	265	257	210

of a single native function call. During profiling, we weren't able to identify any additional operations and when we forcibly sent invalid data types, the programs crashed. From this we assumed that there were no checks, only a single native function call. We weren't able to profile the dynamically created native code that executed the scripts, but it seemed that no or only few additional checks happened there, since the execution was also very fast.

From the table 6.1 we see that the overhead of execution for Lua was several times higher than the measured cost itself. This was expected, since Lua has generally the slowest execution from the measured languages, because it is not compiled to native code. The arguments are the same for the interpreted version of NativeScript. Additionally, in this case the extrapolated value and the value computed by profiling differed by a significant amount. This was most likely caused by the extrapolation inaccuracy, since the measured execution times were quite high (further from 0) compared to the final extrapolated value.

6.1.4 Computation of parameter cost

Table 6.2: Approximation of speed penalty for additional parameter in the call from C/C++ to script (nanoseconds).

	result diff	profiling
C#	0	0
NativeScript opt	0.5	0
C# opt	0.5	0
Lua opt	6	5
JavaScript opt int	35	48
Lua	62	52
JavaScript opt	218	202
JavaScript int	375	387
NativeScript	463	443
JavaScript	612	591

The 0 values in table 6.2 measured by profiling are our assumptions that the arguments are in this case passed simply by value without any checks. The reasons are the same as for the 0 call overhead mentioned earlier. C# optimization to use thunks instead of method

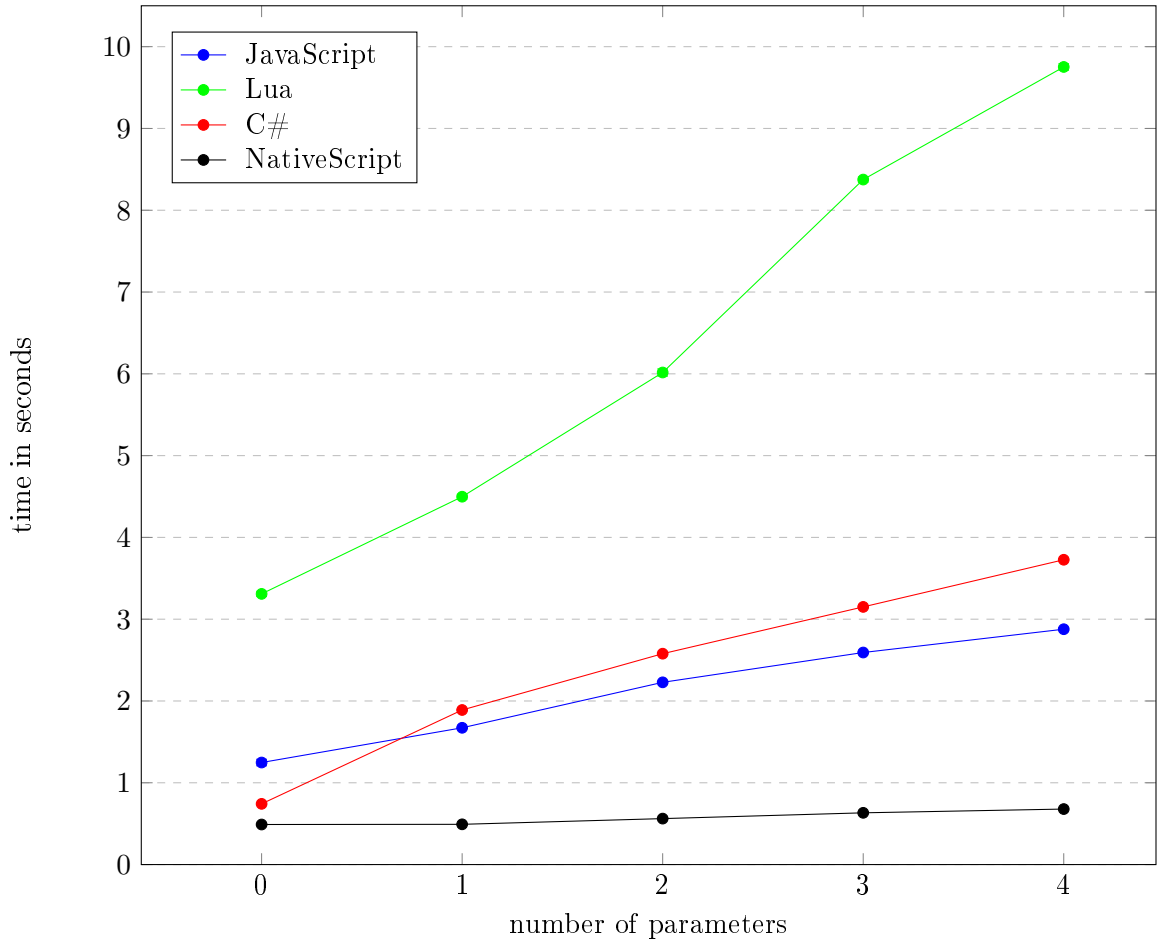
invocation seems to be affecting only the call overhead, since the cost of passing parameters is negligible even for the unoptimized version.

We are not aware of a reason for the difference between profiled and computed values for the optimized integer version of JavaScript. We assume that this difference is caused by measurement errors or misinterpretation of information during profiling. Other values differ only by a small amount.

6.2 Callback

Callback scenario measures the cost of call from scripting environment to C/C++ code and the cost of passing a parameter. Figure 6.4 shows results produced by our testing application for the callback scenario. The result times appear to be mostly linear with the number of callback parameters with few exceptions.

Figure 6.4: 50 million calls from script to native code with varying number of arguments.



In table 6.3 there are times per callback for each language. They were taken directly from the results, since we ran a test case for passing 0 parameters. However, the measurements include overheads, which we attempted to separate by profiling.

During profiling for this scenario we were trying to discover what part of the test result was due to overhead and what part was the actual callback or parameter passing. However, especially for the languages compiled to native code, we weren't able to examine the code that invoked our callback function and thus profiling really only separated the overhead of our result gathering from the rest of the execution time. For this reason, we added script execution time into table 6.3 to serve as an upper bound for the possible callback cost (callback cost for Lua is already excluded from this number). Since there was almost no code to execute within the script and it all happened natively, we assumed that most of the execution was actually the callback invocation.

Lua was the slowest language for both callback invocation and parameter passing speed. During profiling we were able to separate the call instruction and count its function subtree (excluding our result counting) as call overhead. The curve in the figure 6.4 gets steeper for 3 parameters. We examined the generated Lua bytecode and discovered no differences from other cases. The profiling shoved increased execution time of manipulating variables and loading values and it appears that it was caused by garbage collection, which can be invoked during execution of those instructions. However we cannot reliably confirm this hypothesis, since that whole part was implemented using preprocessor macros.

Table 6.3: Speed approximation of the call invocation from script to C/C++ (nanoseconds).

	test result	profile call	profile exec
NativeScript	10	-	2
C#	14.8	-	6.4
JavaScript	23.6	-	10.5
Lua	66.4	18	36.5

We computed the cost of adding a parameter to callback as an average of differences between the individual results. However, this number is biased due to the overhead of saving the result. Similarly to the callback invocation, we were unable to reliably profile the engines to get exact cost for each parameter. However, we identified some parts of the execution that handle parameters and we divided the rest of the execution by parameter count and added into table 6.4 as the upper bound.

Table 6.4: Approximation of speed penalty for additional parameter in the call script to C/C++ (nanoseconds).

	result diff	profile call	profile exec
NativeScript	1.2	-	0.5
JavaScript	10.2	3.2	2.2
C#	16.8	-	9.5
Lua	38.5	9	19

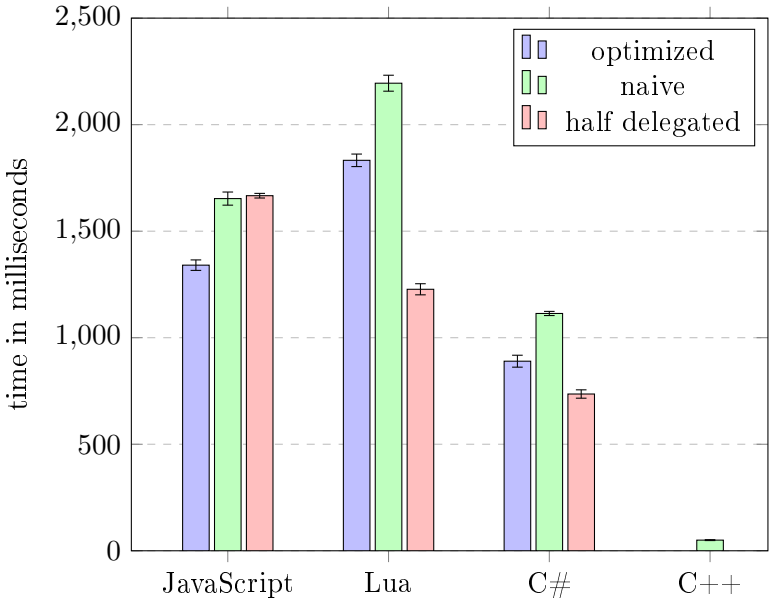
In JavaScript we were able to compute the cost of extracting the parameter value within our callback function. It is very likely that this was the most of the cost for adding the parameter, since the rest of the JavaScript execution is already very fast. In Lua we were also able to compute this value. Most of the rest of the Lua's execution was looking up the callback function in its global table.

There appears to be a big difference between 0 and 1 parameter for C#. We observed significantly lower execution time for 0 parameters, however, the whole execution happened in dynamically compiled code, so we were unable to discover more details.

6.3 Point simulation

This scenario represents more realistic script usage and tries to give answers for some optimization questions. We weren't profiling in this case, since the information about binding costs for different languages was already acquired by previous scenarios. In point simulation we simply comment the results using this information and summarize the strong and weak spots for each language. The results for all configurations and the speed of purely native execution are in figure 6.5.

Figure 6.5: Results of various configurations of point simulation scenario.



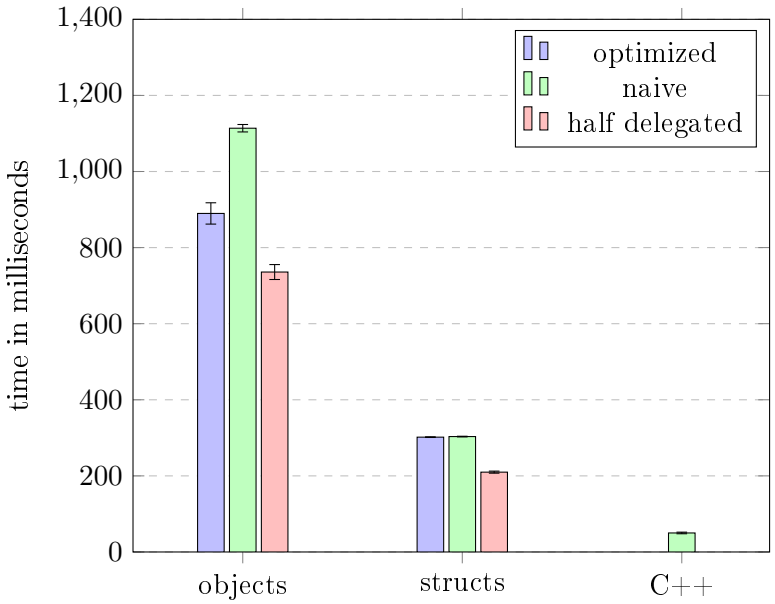
JavaScript has slow binding speed especially when the script is called from C/C++. However once started, the execution is very fast. This is especially valuable, since unlike C#, JavaScript has dynamic types and generally higher-level syntax. The callbacks to C/C++ seems to be also slow, which can be observed from the half delegated case, which has the same speed as naive approach. JavaScript does a lot of garbage collection, as we observed in the expression scenario. That is probably the reason, why the optimized case is only slightly faster than the naive case - it requires allocation of extra variables.

Lua appears to have moderately fast invocation of script code, however its execution is slow compared to the other languages. Even though the callbacks tend to be slower, for half delegated case its execution speed almost halves because of that fact. The optimized case doesn't really speed up the execution, it only avoids several callbacks and thus speeds up the scenario only slightly.

C# is clearly winning for all configurations. We already know that there is almost no cost for calling the script from C/C++. However in point simulation scenario we had to create objects representing the current point each time. The execution itself is known to be very fast for C#, also thanks to the fact that it is a statically typed language with a lot of meta information for the compiler. The callbacks to C/C++ already have some cost, which can be seen from the faster optimized configuration. The binding cost is low enough for delegation of part of the execution into the C/C++ to pay off, as can be seen from the half delegated test case.

In figure 6.6 we can see that most of the C# execution was spent on manipulation of the point objects. When using the structs, the execution is more than 3 times faster and only 6 times slower than the C++ implementation, even though script is invoked. The usage of struct fields instead of getters and setters is practically free, since a shared struct data are used for C++ and C# and the values can be read and written directly. Because of that, the optimized scenario has the same speed as the naive approach. The delegation of half of the code has similar effect as for objects. Thus we recommend the usage of structs if possible.

Figure 6.6: Results of various configurations of point simulation scenario for C#, using different constructs to pass data.



6.4 Recommendations

In this section we give recommendations for the best scripting language for various cases based on the results.

Although from the performance results C# might appear as the best choice for all cases, Mono cannot be linked statically for free and it doesn't officially support 64-bit Windows. In our opinion (among the tested interpreters) Mono is also the hardest to make operational

and the embedding API is the least documented. There are issues with compilation, which is not built into the embedding API, although the workarounds are available. C# is also a strongly typed language, which can sometimes be beneficial, sometimes not.

Lua is very easy to use and it is still reasonably fast. It is also the most portable solution, since it doesn't require special code for each target platform (the other languages have to generate native code). Compared to C# which is statically typed and compiled to native code, it was only 2 times slower in the point simulation scenario. If no algorithms or long executions are to be done within the script, Lua can be a suitable solution even for performance critical applications, since the binding operations are quite fast.

If more execution speed is required and the language should be still dynamically typed, JavaScript is a good option. It requires moderate time for setup and learning. However, for applications that use the binding very often, it is not the best solution. Its primary target is web and fast execution of bigger amounts of code including algorithms, that cannot be implemented in system programming language simply because they are dynamically downloaded from web server.

The decision to implement a custom language should be always carefully considered. Language NativeScript implemented as a part of this work was fastest in all tests. However, the reason for this is that is was implemented *specially* to be fast in those tests. For its simplicity and single variable type it was not sufficient for the point scenario implementation. Features that would allow NativeScript to implement it would require e.g. to add a type system which would on the other hand add overhead to the currently implemented scenarios. A language for a specific purpose can be implemented and outperform 3rd-party solutions, however its further development and widening the area of its usage can quickly throw away all its original benefits.

From the testing of the NativeScript's interpreter we discovered, that 85-90% of its execution was spent on map lookup of local variables just because they were indexed by strings. Simple optimization of using rather constants and array could greatly improve its performance, even though this optimization would be de-facto already a precompilation. Note that profiling was necessary to discover this information. Thus when implementing a language for specific purpose, make sure to profile it to discover potential issues and improvement opportunities.

Chapter 7

Conclusion

In this chapter we summarize and evaluate results of this work. We also suggest further improvements.

7.1 Evaluation of this work

In this work we analyzed the concept of scripting, discussed its origins, purpose and usual features. We selected 3 scripting languages and their implementations that each has different syntax and different implementation approaches. We studied these languages in detail to discover their potential for different scenarios of native interoperability. We also designed and implemented highly specialized custom language NativeScript using LLVM. We discussed its features, performance and caveats and thus showed an example and results of a decision to create a custom language.

We created a testing application that embedded selected scripting language implementations and NativeScript. We created several scenarios, each designed to test a different part of the interoperability. We executed the tests and profiled the scenarios to reveal overheads and performance costs of different operations for each language. We approximated the cost of interoperability operations alone using test results and profiling results. In cases the cost couldn't be accurately determined, we at least set its bounds. We also measured effectiveness of several optimizations that can be attempted during script implementation.

The overall results successfully compared selected languages in the tested areas. Based on the results we also summarized recommendations for each language (and generally scripting approach) usage.

7.2 Future work

This work could be easily extended by adding additional scripting languages. The test application was designed to simplify the task of another language addition.

Also more scenarios could be added. We only briefly mentioned passing more complex parameters like objects or structures. The application also contains few experimental scenarios that weren't discussed in this work. Those can be executed for further details or taken as an inspiration for custom scenario implementations.

We mainly measured execution time, however, there are more criteria that could be relevant. The language implementations could differ in memory consumption, length of pauses due to garbage collection, extensibility or capability of error recovery.

The tests could be also executed on other platforms to see it there are any differences. We unfortunately weren't capable of generating reliable results for Linux, however the whole application is fully compilable and operational on Ubuntu 14.04. We believe that port to Mac and other platforms should be already easy.

Bibliography

[1] *Lua uses* [online]. 2015. [cit. 6. 12. 2015]. Available from: <http://lua-users.org/wiki/LuaUses>.

[2] The BSD 3-Clause License. Available from: <http://opensource.org/licenses/BSD-3-Clause>.

[3] *Variable access performance* [online]. 2012. [cit. 8. 10. 2012]. Browser test, run using Google Chrome to test V8. Available from: <http://jsperf.com/variable-access-performance>.

[4] *MCJIT Design and Implementation* [online]. 2015. [cit. 28. 11. 2015]. Available from: <http://llvm.org/docs/MCJITDesignAndImplementation.html>.

[5] *Kaleidoscope: Extending the Language: Mutable Variables* [online]. 2015. [cit. 25. 11. 2015]. Available from: <http://llvm.org/docs/tutorial/LangImpl7.html>.

[6] *The LLVM Compiler Infrastructure* [online]. 2015. [cit. 26. 11. 2015]. Available from: <http://llvm.org>.

[7] *Lua* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://www.lua.org/>.

[8] *Libraries And Bindings* [online]. 2015. [cit. 14. 5. 2015]. Available from: <http://lua-users.org/wiki/LibrariesAndBindings>.

[9] *Multi Tasking* [online]. 2015. [cit. 28. 4. 2015]. Available from: <http://lua-users.org/wiki/MultiTasking>.

[10] The MIT License. Available from: <http://opensource.org/licenses/mit-license.html>.

[11] *About Mono* [online]. 2015. [cit. 28. 10. 2015]. Available from: <http://www.mono-project.com/docs/about-mono/>.

[12] *Ahead of Time Compilation (AOT)* [online]. 2015. [cit. 2. 11. 2015]. Available from: <http://www.mono-project.com/docs/advanced/runtime/docs/aot/>.

[13] *Mono API Reference* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://docs.go-mono.com/>.

[14] *Documentation* [online]. 2015. [cit. 28. 10. 2015]. Available from: <http://www.mono-project.com/docs>.

[15] *Embedding Mono* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://www.mono-project.com/docs/advanced/embedding/>.

[16] *Linear IR* [online]. 2015. [cit. 30. 10. 2015]. Available from: <http://www.mono-project.com/docs/advanced/runtime/docs/linear-ir/>.

[17] *Mono LLVM* [online]. 2015. [cit. 30. 10. 2015]. Available from: <http://www.mono-project.com/docs/advanced/mono-llvm/>.

[18] *Mono Licensing* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://www.mono-project.com/docs/faq/licensing/>.

[19] *Interop with Native Libraries* [online]. 2015. [cit. 3. 11. 2015]. Available from: <http://www.mono-project.com/docs/advanced/pinvoke/>.

[20] *Mono Supported Platforms* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://www.mono-project.com/docs/about-mono/supported-platforms/>.

[21] *Generational GC* [online]. 2015. [cit. 3. 11. 2015]. Available from: <http://www.mono-project.com/docs/advanced/garbage-collector/sgen/>.

[22] *Small footprint* [online]. 2015. [cit. 7. 11. 2015]. Available from: <http://www.mono-project.com/docs/compiling-mono/small-footprint/>.

[23] *Thread Safety/Synchronization* [online]. 2015. [cit. 5. 11. 2015]. Available from: <http://www.mono-project.com/docs/advanced/runtime/docs/thread-safety/>.

[24] *Trampolines* [online]. 2015. [cit. 2. 11. 2015]. Available from: <http://www.mono-project.com/docs/advanced/runtime/docs/trampolines/>.

[25] The Computer Language Benchmarks Game. <http://benchmarksgame.alioth.debian.org/>, . Accessed: 2015-11-30.

[26] Benchmark language implementations. <http://attractivechaos.github.io/plb/>, . Accessed: 2015-11-29.

[27] *DOCUMENTATION, UNITY SCRIPTING LANGUAGES AND YOU* [online]. 2014. [cit. 3. 9. 2014]. Available from: <http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/>.

[28] *V8 API Reference Guide* [online]. 2015. [cit. 31. 7. 2015]. Available from: <http://v8.paulfryzel.com/docs/master>.

[29] *V8 Embedder's Guide* [online]. 2015. [cit. 15. 5. 2015]. Available from: <https://developers.google.com/v8/embed>.

[30] *Port of Google V8 javascript engine to PowerPC* [online]. Available from: <https://github.com/andrewlow/v8ppc>.

[31] *Port of Google V8 javascript engine to z Systems* [online]. Available from: <https://github.com/andrewlow/v8z>.

[32] ALPERN, B. et al. The Jalapeno virtual machine. *IBM Systems Journal*. 2000, 39, 1, s. 211–238.

[33] ASSOCIATION, E. C. M. et al. Standard ECMA-334: C# Language Specification, 2005.

[34] AYCOCK, J. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*. 2003, 35, 2, s. 97–113.

[35] BEZANSON, J. et al. *Julia* [online]. 2015. Available from: <http://julialang.org/>.

[36] BOEHM, H.-J. Space efficient conservative garbage collection. In *ACM SIGPLAN Notices*, 28, s. 197–206. ACM, 1993.

[37] BÖHM, C. – JACOPINI, G. Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*. 1966, 9, 5, s. 366–371.

[38] CARDELLI, L. Type systems. *ACM Computing Surveys*. 1996, 28, 1, s. 263–264.

[39] CELES, W. *toLua - accessing C/C++ code from Lua* [online]. 2012. Available from: <Seehttp://webserver2.tecgraf.puc-rio.br/~celes/tolua/>.

[40] CONROD, J. *A tour of V8: full compiler* [online]. 2015. [cit. 28. 11. 2015]. Available from: <http://jayconrod.com/posts/51/a-tour-of-v8-full-compiler>.

[41] CONROD, J. *A tour of V8: object representation* [online]. 2013. [cit. 13. 12. 2013]. Available from: <http://jayconrod.com/posts/52/a-tour-of-v8-object-representation>.

[42] CONROD, J. *A tour of V8: Crankshaft, the optimizing compiler* [online]. 2013. [cit. 13. 12. 2013]. Available from: <http://jayconrod.com/posts/54/a-tour-of-v8-crankshaft-the-optimizing-compiler>.

[43] DAHM, M. Byte code engineering. In *JIT'99*. Springer, 1999. s. 267–277.

[44] DALE, N. – WALKER, H. M. *Abstract data types: specifications, implementations, and applications*. Jones & Bartlett Learning, 1996.

[45] DUMBILL, E. – BORNSTEIN, N. M. *MONO: A developer's notebook*. " O'Reilly Media, Inc.", 2004.

[46] FIGUEIREDO, L. H. d. – CELES, W. – IERUSALIMSCHY, R. *Lua Programming Gems*. Lua. org, 2008.

[47] FLANAGAN, D. *JavaScript: the definitive guide*. " O'Reilly Media, Inc.", 2006.

[48] GINSBURG, S. *The Mathematical Theory of Context Free Languages.[Mit Fig.]*. McGraw-Hill Book Company, 1966.

[49] GOOGLE. *The Crack Programming Language* [online]. 2012. Available from: <https://github.com/crack-lang/crack>.

[50] GOOGLE. *Chrome V8 Blog Posts* [online]. 2015. [cit. 11. 8. 2015]. Available from: <https://developers.google.com/v8/blog-posts>.

[51] GOOGLE. *Design Elements* [online]. 2012. [cit. 17. 9. 2012]. Available from: <https://developers.google.com/v8/design>.

[52] GOOGLE. *Introduction* [online]. 2012. [cit. 17. 9. 2012]. Available from: <https://developers.google.com/v8/intro>.

[53] GOOGLE. *Chrome Developer Tools: Videos* [online]. 2012. [cit. 6. 7. 2012]. Available from: <https://developers.google.com/v8/videos>.

[54] HERTZ, M. – BERGER, E. D. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices*, 40, s. 313–326. ACM, 2005.

[55] HEY, T. – PÁPAY, G. *The computing universe: a journey through a revolution*. Cambridge University Press, 2014.

[56] HÖLZLE, U. – CHAMBERS, C. – UNGAR, D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, s. 21–38. Springer, 1991.

[57] IERUSALIMSCHY, R. *Programming in lua*. Lua. Org, 2013.

[58] IERUSALIMSCHY, R. – DE FIGUEIREDO, L. H. – CELES FILHO, W. The Implementation of Lua 5.0. *J. UCS*. 2005, 11, 7, s. 1159–1176.

[59] IERUSALIMSCHY, R. – FIGUEIREDO, L. H. – CELES, W. The evolution of Lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, s. 2–1. ACM, 2007.

[60] ISO, I. IEC 23270: 2006—C# Programming Language, 2006.

[61] JAMES WHITEHEAD, I. – ROE, R. *World of Warcraft programming: A guide and reference for creating WoW addons*. John Wiley & Sons, 2011.

[62] JENSEN, J. C. *LuaPlus* [online]. 2010. Available from: <https://github.com/jjensen/luaplus51-all>.

[63] JONES, R. – LINS, R. D. Garbage collection: algorithms for automatic dynamic memory management. 1996.

[64] KAYLOR, A. *Using MCJIT with the Kaleidoscope Tutorial* [online]. 2013. [cit. 28. 11. 2015]. Available from: <http://blog.llvm.org/2013/07/using-mcjit-with-kaleidoscope-tutorial.html>.

[65] LATTNER, C. – ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, s. 75–86. IEEE, 2004.

[66] LEVINE, J. *Flex & Bison: Text Processing Tools*. " O'Reilly Media, Inc.", 2009.

[67] MAN, K.-H. A no-frills introduction to Lua 5 VM instructions.

[68] NAROFF, S. Clang intro. *Apple:[sn]*. 2009, s. 41.

[69] OUSTERHOUT, J. K. *Tcl: An embeddable command language*. Citeseer, 1989.

[70] OUSTERHOUT, J. K. Scripting: Higher level programming for the 21st century. *Computer*. 1998, 31, 3, s. 23–30.

[71] PALL, M. *The luajit project* [online]. 2008. Available from: <http://luajit.org>.

[72] PARISI, T. *WebGL: up and running*. O'Reilly Media, Inc., 2012.

[73] PAYER, H. – MCILROY, R. *Getting Garbage Collection for Free* [online]. 2015. [cit. 7. 8. 2015]. Available from: <http://v8project.blogspot.cz/2015/08/getting-garbage-collection-for-free.html>.

[74] SIPSER, M. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[75] SOFTWARE, R. *Luabind* [online]. 2005. Available from: <http://www.rasterbar.com/products/luabind.html>.

[76] STANDARD, E. 262: ECMAScript Language Specification, June 2015. 2015. Available from: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

[77] TANTER, É. Beyond static and dynamic scope. In *ACM Sigplan Notices*, 44, s. 3–14. ACM, 2009.

[78] TILKOV, S. – VINOSKI, S. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*. 2010, , 6, s. 80–83.

[79] WIKIPEDIA. List of ECMAScript engines, 2015. Available from: <https://en.wikipedia.org/wiki/List_of_ECMAScript_engines>. [Online; accessed 22-December-2015].

[80] WILLIS, I. Method of compiling bytecode to native code, May 22 2007. US Patent 7,222,336.

[81] WINKLE, L. V. *Game Scripting Languages* [online]. 2009. [cit. 3. 9. 2014]. Available from: <http://blogs.unity3d.com/2014/09/03/documentation-unity-scripting-languages-and-you/>.

[82] ZHUKOV, S. *Mono unmanaged calls performance* [online]. 2014. [cit. 29. 4. 2014]. Available from: <http://forcedtoadmin.blogspot.cz/2014/04/mono-unmanaged-calls-performance.html>.

[83] ZURAS, D. et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*. 2008, s. 1–70.

Appendix A

Abbreviation list

GUI Graphical user interface

API Application Programming Interface

LLVM Low Level Virtual Machine

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

GPU Graphics processing unit

CPU Central processing unit

DOM Document Object Model

JIT Just-in-time

UX User experience

GC Garbage collection

VM Virtual machine

SSA Static single assignment

HIR High-level intermediate representation

LIR Low-level intermediate representation

CLR Common Language Runtime

CLI Common Language Interface

CIL Common Intermediate Language

PIC Position independent code

AoT Ahead-of-time compilation

OS Operating system

ABI Application binary interface

AST Abstract Syntax tree

IR Intermediate representation

MSVC Microsoft Visual C++

Appendix B

Contents of attached CD

- /textSources - Latex source files of the thesis
- smrcepet_2016master.pdf - PDF version of the thesis
- /codeSources - all program source files
  - /nativeScript - source files of our custom language NativeScript
  - /scriptSpeed - source files of our testing application
  - README.txt - installation and user manual
  - runTests.sh - script for execution of all tests (Linux)
  - runTests.bat - script for execution of all tests (Windows)
  - OutputProcessor.class - utility for generating statistics from test results
- README.txt - description of contents of the CD

Appendix C

Installation and user guide

The installation mainly consist of getting all the dependencies. Once the dependencies are installed, cmake can be used to create makefile (Linux) or MSVC project (Windows).

C.1 Dependencies

We present guide to getting the dependency binaries for Ubuntu 14.04 and Windows 7. If the binaries aren't available, we present compilation guide to create them manually.

When building for Windows, download 32bit version of the libraries, since Mono isn't easily available on Windows in 64bit version.

Versions of the libraries used in this work are specified in chapter 5. You can try to use newer versions, but we cannot guarantee compatibility. This guide describes getting the same versions we used.

We would like to note, that the dependency installation guide might not work if e.g. some of the binaries or underlying guides change. It should be taken only as an convenient help in getting the dependencies, since we cannot ensure its correctness.

C.1.1 Flex and Bison

On Ubuntu (and most other Linux versions), these libraries are available trough the package system:

```
apt-get install flex bison
```

On Windows, binaries are available on sourcefoge: <http://sourceforge.net/projects/winflexbison/>. Add the path to the binaries to system variable PATH.

C.1.2 LLVM

Binaries for Windows and some other platforms are available on the llvm download site: <http://llvm.org/releases/download.html>. Download binaries for clang version 3.7.0 and install them. LLVM libraries are included.

On Ubuntu LLVM libraries are available trough packages, although it might be necessary to set LLVM repository first:

```
wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key|sudo apt-key add -
```

```
apt-get install libllvm3.7 llvm-3.7-dev
```

C.1.3 V8

V8 libraries are available trough Ubuntu package system:

```
apt-get install libv8 libv8-dev
```

On Windows, it must be built from source. The guide is available here <https://github.com/v8/v8/wiki/Using%20Git>. We present the necessary steps here:

- First get *depot_tools* using this guide: <http://dev.chromium.org/developers/how-tos/install-depot-tools>

  Cygwin isn't mandatory, but can be used if it is convenient for you.

- Now start *cmd.exe* and run depot_tools to make sure it's up to date:

  ```
  gclient
  ```

- Make sure you are in a directory you selected for building V8. Get V8 and go into its directory:

  ```
  fetch v8
  ```

  ```
  cd v8
  ```

- Generate MSVC project using GYP. Official guide is here: <https://github.com/v8/v8/wiki/Building%20with%20Gyp#visual-studio>. You need python for this step. Just run:

  ```
  python build\gyp_v8
  ```

- To build just V8 libraries (nothing more is required for this work), go to folder *tools/gyp* and open *v8.sln*.

- Select *Release* configuration and build the solution.

C.1.4 Mono

On Linux, add package repository following the official guide: <http://www.mono-project.com/docs/getting-started/install/linux/>. Install the complete version using:

```
apt-get install mono-complete
```

On Windows, download and install the 32bit version: <http://www.mono-project.com/download/#download-win>.

C.1.5 Lua

On Linux, use package system:

```
apt-get install lua5.3
```

On Windows, libraries are available for download here: <http://sourceforge.net/projects/luabinaries/files/5.3/Windows%20Libraries/>. Select static and download version *lua-5.3_ Win32_ vc10_ lib.zip*.

C.1.6 SDL

On Linux, SDL is available trough packaging system:

```
apt-get install libsdl2-dev
```

On Windows, binaries can be downloaded from SDL web: <https://www.libsdl.org/download-2.0.php>. Get 32bit version.

C.2 Compilation

After all dependencies have been acquired, NativeScript and the testing application can be built using CMake. CMake can be obtained from its website <https://cmake.org/download/> or using package system:

```
apt-get install cmake
```

First create build directory outside of the sources. In-source builds are prohibited. Then the steps differ based on the platform:

C.2.1 Linux

- Go into the build directory.
- Run command

  ```
  cmake <path_to_scriptSpeed> -DV8_DIR=<path_to_v8>
  ```

  CMake will try to find the other dependencies, however, some paths might need to be specified. Run the command to see which variables need to be set for your case.
- After successfully generating makefile, build everything using command

  ```
  make
  ```
- Copy directory *scripts* from the source directory to the directory with the executables you just built. It contains script source codes that are tested.

C.2.2 Windows

- Run CMake gui executable.
- Set paths to project source directory and your build directory (has to be different). Set generator to Visual Studio 12 2013 (you can try different version, however we tested only this option). Again, paths to dependencies that weren't found have to be specified manually. Run configuration to see what paths are missing and how to specify them.
- *Configure* the build and then *Generate* the MSVC solution.
- Open the MSVC solution, select MinSizeRel configuration and build the solution.
- Copy directory *scripts* from the source directory to the directory with the executables you just built. It contains script source codes that are tested.
- If not on your PATH, copy dll files *v8.dll*, *mono-2.0.dll*, *SDL2.dll* to the directory with the executables you just built (its name should be *MinSizeRel*).

C.2.3 Running the tests

To run the testing application simply run *scrSpeed* executable with appropriate arguments. You can try e.g.:

```
./scrSpeed lua expression
```

Result should be a number showing the measured time.

To execute all test scenarios, copy script *runTests* from the source directory to the directory with scrSpeed executable and run it. It might take around half an hour to execute all the tests, it depends on your proccessing power. Results will be written into file *output.csv*.

To generate statistics from the results, you can use file *OutputProcessor.class*. You need Java runtime version 1.6 or later to run it. Assuming you are in the same folder as the file, run it using command

```
java OutputProcessor <results_file_name>
```

File *processed-<original-filename>* will be produced. First column represents mean, second standard deviation and third the test case name.

C.3 User guide

This section contains user guide to the testing application. We describe its possible run arguments and configurations.

C.3.1 Configurations

By default the application measures wall time and prints the result in a way appropriate for the runTests script. You can change these settings in file *settings.h* in directory *scriptSpeed*. Recompilation of the application is necessary for the changes to take effect.

C.3.2 Arguments

First argument of the testing application is always target language. Run the application without any arguments to see the options.

For each language, several scenarios are implemented. Run the application olny with the language argument to see which scenarios are available for given language. Scenario name is passed as the second argument.

Other arguments depend on the selected scenario:

- **expression** Third argument represents number of arguments passed in the script invocation. Possible values are from 2 to 6, default is 2. As a fourth argument string *opt* can be passed to specify optimized version of invocation should be used. Otherwise, naive approach is used.
- **callback** Third argument represents number of parameters passed with the callback. Default is 0. Fourth argument can be used to specify the number of callbacks invoked.
- **pointSimul** Third argument represents specific configuration of the point simulation scenario. To see available configurations for given language, run the application without specifying the third argument. Fourth argument can make the application to visualize the testing scenario rather than measure it. You can select values *show* to see the point movements that are tested, or *interactive* to make the leading point follow mouse. Fifth argument can specify custom frame count and sixth custom point count.

There are several additional experimental configurations implemented, however they were not included into this work. We selected scenarios and configurations that we believed to be the best (and sufficient) for gathering the results. So e.g. in Lua or JavaScript, point simulation scenario has configuration with program flow control within the script. It is called *loopControl*. However, since these configurations weren't added to this work, they weren't profiled and cleaned as much as the other scenarios. They may however serve as an inspiration or a basis for extending this work.

Appendix D

LLVM bug workaround

We believe we discovered a bug in LLVM. It happens during function call of *PHINode:Create* within which overriden operator *new* is called. Inside the new operator implementation, fields of the not yet created class are set, specifically value of field *HasHungOffUses* is set to true. However, during constructor call, this memory can be erased - C specification states the memory is undefined. On Windows using MSVC build, we encountered cases when during constructor call all fields have been set to 0, thus resetting the field value to false. We believe this implementation style to be a bad practise that doesn't guarantee corect behavior.

As a workaround we added one statement to include file *llvm/IR/Instructions.h* to line 2327:

```
this->HasHungOffUses = true;
```

This addition is the first line of method *allocHungoffUses(unsigned N)* so it basically makes sure that before any allocation the field is set correctly. This is by no means a robust bugfix. It is simply a fast solution that made this part of LLVM operational in our case. If you encounter issues during generation of IR, this solution might help you as well.