"Incremental compiler development"

Mouton, Youri

ABSTRACT

Popular programming languages are ever growing projects, evolving as new features are implemented and issues are fixed. Some projects even rewrite their compiler when needed, for example the PHP project when they released the seventh version. We try to explore whether and how programming language interpreter development can be achieved in an incremental way to avoid rewrites, critical to making compiler development better. We first analyse and motivate the choice of Truffle and GraalVM as tools for compiler development. We then find a case study that allows us to showcase the incremental nature of our work. We then dive deeper into the details of the implementation of the desired features we discussed in the case study. We illustrate how we tried to write an interpreter for a modern dynamic object-oriented language with metaprogramming features in an incremental fashion. Finally we reflect on the development process and whether the objective of this dissertation has been achieved, including a discussion of the hard challenges and the lessons learned from the point of view of the proposed methodology.

CITE THIS VERSION

UCLouvain

École polytechnique de Louvain

Incremental Compiler Development

Author: **Youri Mouton**
Supervisors: **Kim Mens, Nicolas Laurent**
Readers: **Xavier Gillard, Guillaume Maudoux**
Academic year 2018–2019
Master [120] in Computer Science

I would like to thank Kim Mens and Nicolas Laurent for their availability during the year. They guided me throughout the writing of this dissertation.

I would like to thank Guillaume Maudoux and Xavier Gillard for agreeing to be readers.

Finally I would like to thank my family and friends for supporting me throughout the writing process.

Abstract

Popular programming languages are ever growing projects, evolving as new features are implemented and issues are fixed. Some projects even rewrite their compiler when needed, for example the PHP project when they released the seventh version.

We try to explore whether and how programming language interpreter development can be achieved in an incremental way to avoid rewrites, critical to making compiler development better. We first analyse and motivate the choice of Truffle and GraalVM as tools for compiler development. We then find a case study that allows us to showcase the incremental nature of our work.

We then dive deeper into the details of the implementation of the desired features we discussed in the case study. We illustrate how we tried to write an interpreter for a modern dynamic object-oriented language with metaprogramming features in an incremental fashion.

Finally we reflect on the development process and whether the objective of this dissertation has been achieved, including a discussion of the hard challenges and the lessons learned from the point of view of the proposed methodology.

Contents

List of Figures

List of Listings

Acronyms

ANTLR ANother Tool for Language Recognition. 2, 3, 5, 6, 29, 31

AST Abstract Syntax Tree. 2, 5, 7, 8, 9, 10, 31, 37

CI Continuous Integration. 48

DSL Domain-Specific Language. 8, 9

EA Escape Analysis. 11, 12

EBNF Extended Backus-Naur Form. 18

GC Garbage Collector. 13

GCC GNU C Compiler. 1

IDE Integrated Developement Environment. 51

ISA Instruction Set Architecture. 2

JIT Just-in-time. 2, 11

JVM Java Virtual Machine. 2, 10, 11, 13

OOP Object-oriented Programming. 17, 22, 24, 28, 32, 34, 35, 36, 46, 50

OS Operating System. 2

OSM Object Storage Model. 10, 32, 35, 36

PEA Partial Escape Analysis. 11, 12

TDD Test-Driven Development. 2, 3, 49

VM Virtual Machine. 11

Chapter 1

Introduction

It is no secret that compilers are often huge and complicated projects that require skills that seem out of reach for the average computer scientist. They are often written in C/C++, assembly and the code is scattered amongst many files. For example, the popular GNU C Compiler (GCC) compiler consisted of almost 15 million lines of code in 2015 [1]! This makes diving in compiler development seem like an impossible task to many programmers.

Although difficult, the exercise of writing an interpreter or a compiler is very important if we want to truly understand how programming languages work. Every computer scientist should have implemented at least an interpreter [2]. This doesn't necessarily have to be hard!

This thesis focuses on showing that writing compilers for all sorts of languages can be done with relative ease without compromising performance and make it grow in a maintainable manner, thus saving enormous amount of time that can be wasted when rewriting a compiler or having to implement common tasks such as object inheritance from scratch.

1.1 Incremental Compiler Development

To analyse whether the task we defined above is possible, we would like to build a compiler for a language that will grow in size over time as we learn about grammars, parsing and compiler creation. For that we need a set of tools that will allow us to avoid having to recreate a whole new project from scratch whenever we want to evolve the language thus wasting precious time. The tool infrastructure should make it easy to make quick additions to the language grammar and add new nodes to finally adapt the existing code to take these nodes into account. With this in mind we should in effect be able to extend our language with new ideas and maybe even

major paradigm changes as we please.

1.2 Technologies

The objective of this effort is to find and evaluate and analyse the right set of tools that will allow us to write a fast compiler quickly. To accomplish this in a timely manner we want to avoid the hassle of low-level work as much as possible. Learning the most popular Instruction Set Architectures (ISAs) is very time consuming. For example $x86\_64$ or amd64 have 2,034 different instructions available [3] and the machine code is cryptic if not unreadable. We can avoid this by using for example the Java Virtual Machine (JVM) as target since then all the machine dependent work is done for us. In other words, if we use the JVM as our target machine, we can make our compiler produce programs that can run on many different architectures and Operating Systems (OSs) that the JVM supports. Furthermore, JVM assembly code is much easier to read as it is stack based. Besides the aforementionned reasons, a reference book we used throughout the research for this thesis teaches compiler building using the JVM [4].

A programming language can be represented by an Abstract Syntax Tree (AST), a tree representation of the abstract syntactic structure of source code. To implement a programming language, one needs a parser to read source code and generate the nodes of the AST and a compiler to make the node meaningful in order to produce results.

The particular technologies chosen for this research are ANother Tool for Language Recognition (ANTLR), a tool to generate parsers, Truffle, to make the AST produced by ANTLR meaningful, and Graal for Just-in-time (JIT) compilation. These tools and the reason for their usage will be analysed and motivated in the next chapter.

1.3 Validation

Testing a programming language is usually done by executing programs and checking for their input. We wish to make tests more interesting by testing not only programs but syntax itself and various stages of the source code interpretation by the compiler.

The development methodology we choose to implement our work is Test-Driven Development (TDD), which means that we write tests before we start implementing a feature we want. By writing a generous amount of tests and using continuous

integration through the combination of Github[1] and Travis-CI[2], we can ensure that the adoption of a new feature isn't inadvertently breaking another.

All of the code of this project is Open-Source and available on Github. Results of the continuous integration tests for each commit can be found under the commits tab on Github.

The tests will mostly be snippets of code illustrating a feature. We define pass tests as tests that must work in order for the test to pass and fail tests as tests that need to fail in order for the test to pass. Fail tests are useful to make sure our grammar or compiler isn't broken and that we are not accidentally accepting wrong programs in our language. We will use JUnit tests to apply the TDD methodology.

As the language we're implementing evolves there will be breaking changes along the way. For this we will tag versions and make new branches when we implement a whole new language construct. The reader will be able to follow the language's evolution by reading commits and by switching git branches. This will also make it easy to diagnose problems since we can just go back a few commits.

Finally, we use Coveralls[3] to make sure we have extensive code coverage on the whole project.

1.3.1 Contributions

We hope to contribute to the field of compiler research in the following ways:

- Showing that compilers can be developed in a modern way using the right set of tools;
- Showing that compilers can be tested using TDD, further making compiler development a manageable task.

Structure

The remainder of this document is structured as follows.

We will first introduce the chosen tools Truffle, Graal and ANTLR in **Chapter 2**.

The language we plan to build incrementally will be described in **Chapter 3** where we try to come up with an interesting scenario to illustrate the incremental property of the development phase of our compiler project.

[1] **https://github.com/nbyouri/simplelanguage**
[2] **https://travis-ci.org/nbyouri/simplelanguage**
[3] **https://coveralls.io/github/nbyouri/simplelanguage**

Work on the language and its evolution, the challenges, the problems encountered and the lessons learned along the way will be described in **Chapter 4** and **Chapter 5**.

Chapter 6 describes some of the difficulties we encountered and how we implemented validation strategies for our language.

Chapter 7 hints at some interesting possible future developments and gives a conclusion to this thesis.

We wish you a pleasant reading.

Chapter 2

Background

In this chapter we motivate the choice of tools we deem fit for the task we've set to accomplish: ANTLR, Truffle and Graal.

2.1 Parser

When writing a compiler, it is easiest to generate an AST interpreter, since parsers for context free grammars process source code in a tree-like fashion. Parse trees are a natural way to express a program; we just need to add an execute method to each node of the tree that will call its childrens' execute method. That's theoretically all we need to get the language up and running.

The task of writing the parser can be automated by using a parser generator as doing the task of writing code to deal with nodes by hand is tedious and can lead to mistakes. We use ANTLR for this project because it is featureful, and is capable of generating performant top-down parsers for LL(k) grammars. More importantly, it is simple: the lexer, parser, AST and documentation can all be generated from a single source file. Finally, it is very popular thus examples are widely available. Examples of ANTLR grammar definitions to create compilers for languages like C, Java, Cobol, JSON, PHP or Oberon are available online.[1]

2.1.1 ANTLR IntelliJ plugin

When changing the ANTLR grammar file, we can quickly test it by first running *Generate ANTLR Recognizer* on our file via the context menu and then typing some code in the ANTLR Preview window in IntelliJ.

[1] **https://github.com/antlr/grammars-v4**

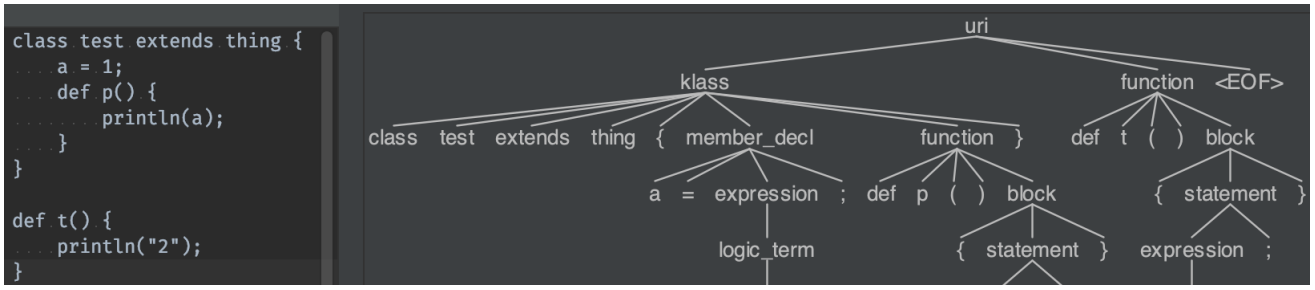

Figure 2.1: Uri oop grammar test

We can see that ANTLR accepts the input code we tried to implement. Wrong input like trying to define an interface, which we did not implement in the grammar will yield the following error:

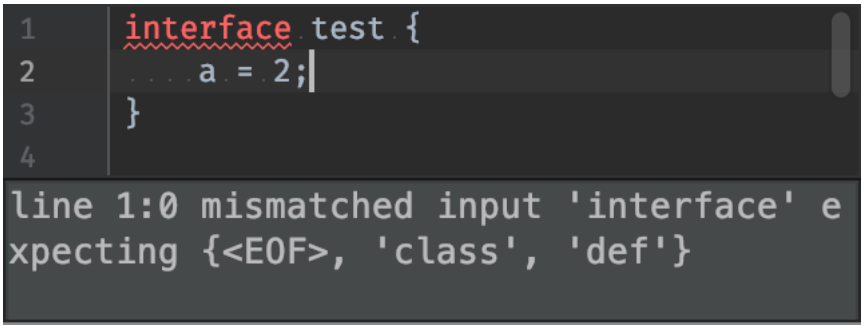

Figure 2.2: Uri interface grammar test

Looking at Figure 2.2, we can see that our grammar expects either *class* or *def* to define a class or a function respectively.

2.1.2 Alternatives

There exist other parser generators such as Yacc, Flex and JavaCC among the popular ones but none that supported Java offered the amount of features ANTLR has such as the IntelliJ plugin. The drawbacks of using ANTLR is that it requires a runtime program and it is more heavyweight than its competition. For our usage, we feel that ANTLR is the best fit as we primarily focus on ease of use and its popularity means it is widely documented.

2.2 Truffle

Truffle[2] is a framework to create fast AST interpreters that will be used along with Graal to create a compiler having performance characteristics comparable to bytecode interpreters. It will be fast and easy to hack into as it is written in Java. Graal is described further in this document. Truffle is actively being developped by a team of about 40 programmers at Oracle.

The alternative to using Truffle is the developer has to write code to handle the nodes and make them meaningful, and more importantly make the interpretation of source code efficient. Typical languages use a large amount of in-house optimizations that could simply be shared by a common framework, which is what Truffle is intended to be. This means that a lot of development time is spared from the developer.

A common pitfall of AST interpreters is that they're slow and for this reason implementers will usually speed up interpretation with bytecodes. However such optimizations are complicated and their binary representation makes them hard to work with while ASTs are much easier to manipulate. We will show that by incorporating profiling information, and dynamically rewriting the AST, we can achieve good performance.

2.2.1 AST representation

AST nodes are represented as shown in the listing below. It is the abstract class Node that all other nodes will implement. It contains a method for executing the node. The execution frame is given as parameter to the execute method, the frame contains itself an Object[] array with the values of the local variables. All values are represented as Java type Object. Every node contains a pointer to its parent node in the AST, which allows for fast replacement of a node with another node.

[2] **https://github.com/oracle/graal/tree/master/truffle**

```
abstract class Node {
    // Executes the operation encoded by this
    // node and returns the result.
    public abstract Object execute(Frame f);
    // Link to the parent node and utility to
    // replace a node in the AST.
    private Node parent;
    protected void replace(Node newNode);
}
```

Listing 1: AST node class.

2.2.2 Tree rewriting

Replacing a node by another one in the AST represents a way to improve the executed AST at runtime by replacing a node with a more specialized node. Profiling might hint at a node that will perform better at some operation after analysis of operands. Specialized nodes make assumptions about the operands that will hold later in the execution. If the assumption does not hold, the node might be replaced with a node that handles a more generic case. Let's take for example the add function shown below.

```
function add(a, b) {
    return a + b;
}
```

Listing 2: Javascript add function.

This function needs to handle the case where Ints, Doubles and Strings are used as arguments. But instead of using a single node that can handle all three cases, we could use several nodes that each do different checks. Truffle will use an other appropriate node as input requires if the current node doesn't handle the predicted type. Figure 2.1 shows the different transition states as described in the Self-Optimizing AST Interpreters paper [5]. The top node *Uninitialized* represents the state after parsing where there isn't any type information available. Then there is a transition towards the *Integer* node since Integer addition is very fast. Upon detecting floating point numbers, we transition towards the *Double* node. The *String* node does the String concatenation. The number of add nodes depends on the language, one might not support string concatenation or one might support many more types of addition.

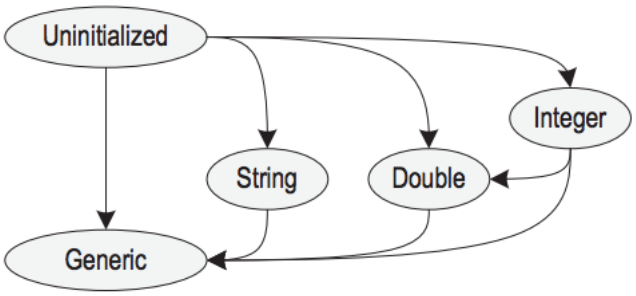

Figure 2.3: Different add nodes transitions

Truffle provides the machinery for implementing such self-optimizing AST interpreters for new languages as discussed above. It is a good reason to use Truffle as having optimized execution will lead to faster execution ties. The Domain-Specific Language (DSL) Truffle provides allows the implementer to spend less time on boilerplate code required by a compiler for such tasks as defining nodes or implementing objects.

2.2.3 Truffle specialization example

The Truffle code for the add operation as described above is shown below, where we find the use of the Truffle DSL to set the node name in the AST node with annotation @NodeInfo. The @Specialization annotation denotes an operation for a type, here the add node works on longs, BigIntegers and Strings. The *rewriteOn* keyword which is part of the @Specialization annotation allows to transition to the node below, in order of definition, when the execution throws an ArithmeticException. The String add specialization uses the *isString* guard to ensure that at least one of the arguments is a string. A method annotated with @Fallback must be last and is treated as a @Specialization that implicitly links all the guards of all other declared @Specialization annotated methods of the operation in a negated form. The @TruffleBoundary annotation allows Truffle to determine whether the method throws a ControlFlowException which is used internally for performance optimizations, they are used for control flow nodes such as while, break, continue, if, ...

```
@NodeInfo(shortName = "+")
public abstract class URIAddNode extends URIBinaryNode {
    @Specialization(rewriteOn = ArithmeticException.class)
    protected long add(long left, long right) {
        return Math.addExact(left, right);
    }
```

```
    @Specialization
    @TruffleBoundary
    protected BigInteger add(BigInteger left, BigInteger right) {
        return left.add(right);
    }

    @Specialization(guards = "isString(left, right)")
    @TruffleBoundary
    protected String add(Object left, Object right) {
        return left.toString() + right.toString();
    }

    protected boolean isString(Object a, Object b) {
        return a instanceof String || b instanceof String;
    }

    @Fallback
    protected Object typeError(Object left, Object right) {
        throw URIException.typeError(this, left, right);
    }
}
```

Listing 3: Truffle add node definition.

2.2.4 Object Storage Model

The Object Storage Model (OSM) standardises and optimises the machinery used to handle dynamic objects. It allows the developer to skip the design of all runtime mechanisms for managing dynamically typed objects, reducing code duplication and potential errors. The OSM is generic, language agnostic and portable, it can be used to implement lots of different types of dynamic languages. For example, the Truffle-based language runtimes of Ruby and Javascript share the same type specialization using the OSM [6].

2.2.5 Polyglot

Truffle provides language interoperability through the Polyglot framework allowing Truffle languages to call each other. They can also call other JVM languages like Java, Kotlin and Scala. The way this works is by joining the ASTs of the two JVM languages together, they are then compiled and optimised as a single tree by Graal. Truffle provides a test[3] that we use in our project to make sure our language

[3] https://github.com/oracle/graal/blob/master/truffle/docs/TCK.md

is compatible with Polyglot, i.e interoperable with other JVM languages. This is desirable if we want add new constructs to existing languages or if we want to mix two languages we've written. Polyglot is described further in the "High-Performance Cross-Language Interoperability in a Multi-language Runtime" paper [7].

2.2.6 Debugging

It is desirable to have good debugging facilities when developing a new language. Many new languages don't have good debugging and profiling tools from the start, seriously hurting the language's usability. Generally we want the running code to stay close to the source, meaning optimizations would be disabled thus making debugging slow.

Truffle allows to generate code that stays close to the source when debugging without slowing down execution. Compiler optimizations can be enabled, this is possible because Truffle and Graal generate metadata when compiling the source code that can be used to deoptimize running code back to it's original state. A NetBeans GUI plugin is available to debug Truffle languages. [8]

2.2.7 Conclusion

To conclude we fee like Truffle is the perfect fit as framework to interpret source code as it provides a large amount of APIs to help with common tasks such as types, control flow nodes, caches, ... It also performs a lot of optimization for free, which is critical if we want a fast compiler.

2.3 Graal

Graal is a state-of-the-art open-source compiler developed at Oracle[4]. It is what will allow us to generate a fast compiler combined with Truffle. It is built on top of the Java HotSpot Virtual Machine (VM), adding optional JIT compilation. It features very advanced optimizations such as Partial Escape Analysis (PEA). [9] Figure 2.2 shows how Graal plugs into the HotSpot VM and how Truffle acts as a framework to implement interpreters for popular languages.

[4] **https://github.com/oracle/graal**

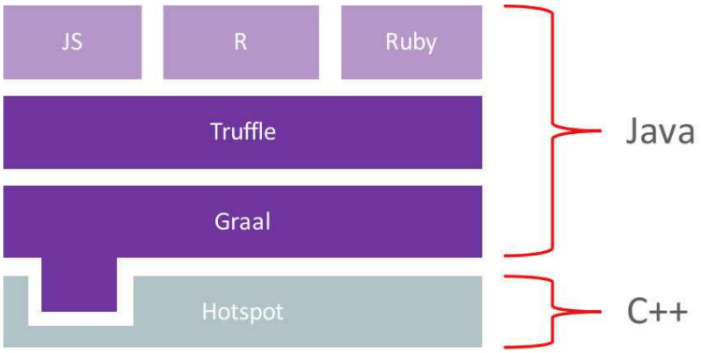

Figure 2.4: Truffle & Graal

2.3.1 Partial Escape Analysis and Scalar Replacement

Escape Analysis (EA) is an optimisation technique that allows the compiler to know whether an object is accessible outside of the allocating method or thread. This is used to enable optimizations like Scalar Replacement: when the compiler detects that an allocated object does not escape it can choose to allocate the entire object storage on the stack, including the header and the fields and reference it in the generated code. This effectively eliminates heap allocation when they are not necessary, greatly improving performance. EA is a complex optimization that was made famous by the JVM but it often does not apply, and when EA isn't enabled, Scalar Replacement fails. The code below shows a trivial code example where EA does not optimize because the control-flow node will be evaluated before the variable assignment. [10]

```
public class ScalarReplacement {
    int x;
    boolean flag;
    // ...
    public int split() {
        MyObject o;
        if (flag) {
            o = new MyObject(x);
        } else {
            o = new MyObject(x);
        }
        return o.x;
    }
}
```

Listing 4: Trivial EA failure example.

This case is handled well by Graal. Instead of doing an all-or-nothing approach to EA, a new algorithm is used that performs control-flow sensitive PEA. It allows Scalar Replacement to be performed on individual branches. Lock Elision is another optimization that can be enabled thanks to EA, it removes superfluous synchronization on non-escaping objects in multi-threaded code [11].

2.3.2 Strengths and drawbacks

Truffle and Graal have a lot to offer:

- Fast compiler creation with the Truffle framework
- Run as fast as any other language without additional work
- Provide debugging features automatically
- Support Profiling, also automatically
- High performance optional Garbage Collector (GC)
- Support low-level languages such as C as well as dynamic scripted languages like Ruby
- Ahead-of-time or just-in-time compilation
- Hotswap of code into a running system [12]

But they do present a couple of drawbacks as well. First, the warmup time can be pretty long; it is the time needed for the language runtime to initialize in order to start executing source code. Secondly, graal has extensive memory usage. This will be taken into account when benchmarking our language.

2.3.3 Alternatives

Truffle and Graal go together but if we had decided to not use Truffle, we would also have had to generate machine code for programs to be executed. This means that we would need to write code to either produce JVM code or assembly for any architecture we could want to run programs on. This is an extremely comprehensive task as there are many architectures in the wild and learning them is very time consuming. For this reason we believe Graal is a perfect fit for our language as we want to avoid spending time implementing things that have been done before by countless other projects.

2.4 Related work

The following projects using Truffle and Graal are of interest to us as we can use them as inspiration or reference where the official documentation lacks. They are actively being maintained:

- TruffleRuby, also four times faster than JRuby.
- SimpleLanguage, a small language to showcase most of Truffle's features. Our language will be inspired by this.
- Graal.JS, a Javascript implementation whose speed matches V8.
- Sulong, allowing Graal to interpret LLVM bytecode meaning running C/C++/Objective-C/Fortran/Swift and potentially even Rust.

Chapter 3

Case Study

In this chapter we discuss the different versions of the languages we're going to implement incrementally. Each version will present new challenges that will be described **Chapter 4**, but for now we simply describe the features we would like to implement and how we see the language evolve over time.

Although we haven't named our language yet we assign .uri as extension for source files. We hope that the implementer will eventually produce a proper name for the language and its compiler.

3.1 Initial Language Properties

3.1.1 Types

The language we implement will have strong dynamic typing, meaning the type checking is done at runtime and the compiler will not try to do automatic conversions between types. The reason for this choice is down to the implementer's taste. Modern languages using mostly dynamic types also motivated the choice. The types we first chose to implement are the following:

- Number: 64 bit numbers, using the Java long primitive type.
- Boolean: using the Java boolean primitive type.
- String: using the Java String type.
- Object: key value stores using the object model provided by Truffle.
- Null: a singleton with null as only value.

Type definitions using the Truffle abstractions are found in the URITypes.java file in the source tree. They are defined with the *@TypeSystem* Truffle annotation:

```
@TypeSystem({long.class, URIBigNumber.class,
boolean.class, String.class, URIFunction.class, URINull.class})
```

Listing 5: TypeSystem configuration.

3.1.2 Functions

In our language, functions are implemented as a TruffleObject. Functions can be redefined, meaning functions can change at run time. Functions are also defined as a literal node and are treated as first-class citizens. They can also be used as function return values. An example is shown in the code below. A few built-in functions are implemented as well: readln to read from standard input, writeln to write to standard output, NanoTime to get nanosecond System time useful for benchmarking.

```
def ret(a) { return a; }
def dub(a) { return a * 2; }
def inc(a) { return a + 1; }
def call(f, v) { return f(v); }
def retfun(f) { return f; }

def main() {
  println(ret(42));
  println(dub(21));
  println(inc(41));
  println(call(ret, 42));
  println(call(dub, 21));
  println(call(inc, 41));
  fun = retfun(dub);
  println(fun(22));
}
```

Listing 6: Functions as first-class citizens.

3.1.3 Control Flow and Operators

The language features basic arithmetic and logical operators: +, -, *, /, logical and, logical or, ==, !=, $\leq$, $\leq$, $\geq$, $\geq$. Variables are only visible within the context they were defined in.

The *if*, *while* nodes along with *break* and *continue* and *return* are also implemented.

3.1.4 Grammar

```
Uri ::= {Function}
Function ::= "def" IDENTIFIER "(" [IDENTIFIER {"," IDENTIFIER}] ")" Block
Block ::= "{" {Statement} "}"
Statement ::= "while" "(" Expression ")" Block
            | "break" SEMI
            | "continue" SEMI
            | "if" "(" Expression ")" Block ["else" Block]
            | "return" [Expression] ";"
            | Expression ";"
            | "debugger" ";"
Expression ::= LogicTerm {"||" LogicTerm}
LogicTerm ::= LogicFactor {"&&" LogicFactor}
LogicFactor ::= Arithmetic [("<"|"<="|">"|">="|"=="|"!=") Arithmetic]
Arithmetic ::= Term {("+"|"-") Term}
Term ::= Factor {("*"|"/") Factor}
Factor ::= IDENTIFIER (MemberExpression)?
         | STRING_LITERAL
         | NUMERIC_LITERAL
         | "(" Expression ")"
MemberExpression ::= (
                       "(" [Expression {"," Expression}] ")"
                     | "=" Expression
                     | "." IDENTIFIER
                     | "[" Expression "]"
                     )
                     [MemberExpression]
```

Listing 7: Imperative language grammar.

At this stage, we have a very simple imperative programming language featuring functions, basic types and basic control-flow constructs.

3.2 Building an Object-Oriented Language

Objects and their relations are a natural way for us humans to think about programming problems, and for this reason it is desirable for our language to feature the Object-oriented Programming (OOP) paradigm.

3.2.1 Grammar Changes

The grammar for our language is updated as shown below.

```
Uri ::= {Function | Class}
Class ::= "class" IDENTIFIER ["extends" IDENTIFIER] "{" {MemberDecl} {Function} "}"
MemberDecl ::= IDENTIFIER ["=" Expression] ";"
Function ::= "def" IDENTIFIER "(" [IDENTIFIER {"," IDENTIFIER}] ")" Block
Block ::= "{" {Statement} "}"
Statement ::= "while" "(" Expression ")" Block
            | "break" SEMI
            | "continue" SEMI
            | "if" "(" Expression ")" Block ["else" Block]
            | "return" [Expression] ";"
            | Expression ";"
            | "debugger" ";"
Expression ::= LogicTerm {"||" LogicTerm}
LogicTerm ::= LogicFactor {"&&" LogicFactor}
LogicFactor ::= Arithmetic [("<"|"<="|">"|">="|"=="|"!=") Arithmetic]
Arithmetic ::= Term {("+"|"-") Term}
Term ::= Factor {("*"|"/") Factor}
Factor ::= IDENTIFIER (MemberExpression)?
         | STRING_LITERAL
         | NUMERIC_LITERAL
         | "(" Expression ")"
MemberExpression ::= (
                       "(" [Expression {"," Expression}] ")"
                     | "=" Expression
                     | "." IDENTIFIER
                     | "[" Expression "]"
                     )
                     [MemberExpression]
```

Listing 8: OOP language grammar.

To make the grammar difference more clear as reading Extended Backus-Naur Form (EBNF) isn't always easy, here are the relevant changes in the form of railroad diagrams:

First the program rule was changed to accept class definitions as well as functions:

Figure 3.1: Uri program rule without classes and with classes

Then two more rules were added to deal with class definitions as shown in figures 3.2 and 3.3.

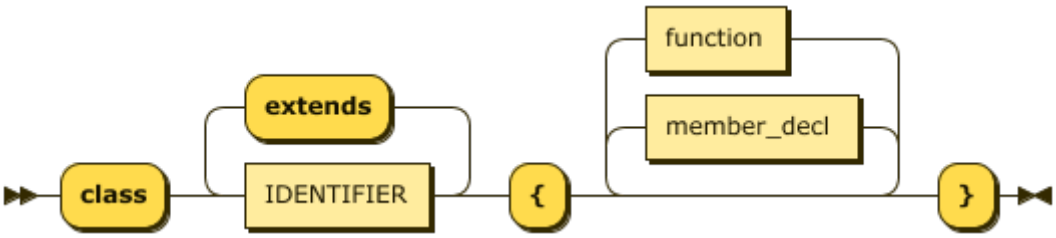

Figure 3.2: Uri class rule

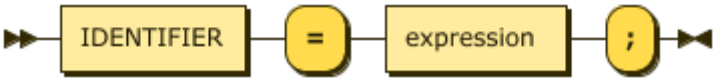

Figure 3.3: Uri class member rule

These diagrams show that the grammar didn't change much to allow our language to parse classes.

3.2.2 The Class system

This updated language features classes with fields and methods. An example of a class is shown in the code below.

```
class A {
    a = "Hello";
    def hello() { return this.a + "World"; }
}
def main() {
    c = new(A);
    println(c.hello());
}
```

Listing 9: Classes and methods.

The class system is optional to keep our language backwards compatible with the first imperative version.

3.3 Metaprogramming

A powerful desired feature in modern languages is metaprogramming, which allows controlling parts of the program reflectively at runtime like defining new methods and getting or setting instance or class variables and dealing with dynamic method sending. We will use Ruby as inspiration for our implementation, a popular language for metaprogramming. Truffle and Graal have been designed to make it easy to

implement efficient metaprogramming [13]. In this section there will be no grammar changes, but the dynamic nature of our language will increase.

3.3.1 Metaprogramming Patterns

The metaprogramming patterns we wish to implement are the following:

1. **Method sends**: this feature allows to dynamically call a method by it's name as a literal string. It is useful when the method call depends on some condition. The code below shows an example.

```
class MyClass {
    def message1() { /* method body */ }
    def message2() { /* method body */ }
    def message3() { /* method body */ }
}
def main() {
    mc = new MyClass();
    mc.message3(); // normal method call
    n = 3;
    // dynamic method call through metaprogramming
    mc.send("message" + n);
}
```

Listing 10: Dynamic method send.

2. **Hook methods**: they allow to affect what happens when a method is called. For example, we might want to execute some specific action when a missing method is called. An example is shown below.

```
class Foo {
    def foo() { println "foo"; }
    def method_missing(function, args) {
        println("This method is missing...");
    }
}
class Bar {
    def bar() { println "bar"; }
}
def main() {
    // bar() is not a method of class foo,
    // method_missing will catch the call instead.
    f = new Foo();
    println(f.bar());
}
```

Listing 11: Dynamic method proxy.

3. **Code evaluation**: we want to be able to evaluate a string as our language source code at runtime. This is useful for template systems. An example is shown below.

```
def main() {
    eval("def foo() { return 14 + 2; }");
    println(foo());
}
```

Listing 12: Dynamic code evaluation.

4. **Variable access**: as for method sends, we want to be able to read and write class fields as instance variables using a dynamic value as a name. An example is shown below.

```
class A {
    some_str = "Hello";
}
def main() {
    a = new A();
    // sets instance variable some_str to "World"
    a.var_set("some_str", "World");
    // gets instance variable some_str
    println(a.var_get("some_str"))
}
```

Listing 13: Class instance variable access.

3.4 Feasibility

Wrapping up this chapter, we feel like the language we have defined is feasible, as dynamic languages with classes, objects and metaprogramming exist in good numbers: python, ruby, crystal, PHP... The papers we have referenced describe Truffle as being an ideal choice to implement such dynamic languages, giving us confidence in our ability to complete the work.

We wish to make our language distinct by syntax and semantic choices though. The next chapter focuses on its implementation.

Chapter 4

Development

4.1 Implementing a Truffle Interpreter

This section describes what it takes to build an interpreter with Truffle to be run in GraalVM. First, we discuss the project structure and the different Java classes involved. We then discuss the transitioning towards an OOP language and the challenges that it encompasses. Finally, we discuss the lessons learned in developing with such a framework.

4.1.1 Project structure

The java source structure of the language contains 10 packages, the reasoning behind this is explained in the following chapter. We will also describe notable classes within these packages.

The source code for our interpreter is structured as follows:

package name	–	use
root package	–	language main, exception handling
builtins	–	builtin language functions such as println
nodes	–	abstract nodes, root nodes
nodes.access	–	nodes for object data retrieval
nodes.call	–	function invocation node
nodes.controlflow	–	control flow nodes such as while, if
nodes.expression	–	expression nodes such as arithmetic and comparison operators
nodes.interop	–	GraalVM language interoperability specification
nodes.local	–	function or class local variable access
parser	–	grammar and other parser related classes
runtime	–	uri types, function registry, context

Uri

The top-level package **uri** contains the main language class *URILanguage*, extending *TruffleLanguage*. It registers the language name and ID, a mime-type and a few Truffle configuration options. It also launches the parsing, deciding to start from a main function if it exists or if we simply start registering every function.

At the top the *URIException* package can also be found. It contains methods to print adequate error messages. The language just aborts during unexpected operations so the class prints the error and where it comes from within the source code.

Nodes

The **uri.nodes** package contains base nodes used during the program's semantic evaluation.

- *URIEvalRootNode*, the first class called when evaluating a program, performs two tasks:

  1. Lazy registration of functions on first execution. This fulfills the semantics of "evaluating" source code.
  2. Conversion of arguments to types understood by uri. The uri source code can be evaluated from a different language, i.e., the caller can be a node from a different language that uses types not understood by uri.

- *URIBinaryNode*, abstract class for operations that take a left and right argument. An example specialization of it is the *URIAddNode*, performing addition.

- *URIExpressionNode*, an abstract class for all nodes that produce a value, using type specialization.

- *URIRootNode* the root node for all functions, builtin or user-defined.

- *URIStatementNode* abstract class of all nodes as all nodes can be used as statements (not returning anything), even expressions.

- *URITypes* the type system of the language uri, where we define available types, type conversions and checks. Type conversions for examples include the method *castBigNumber* where we allow the primitive *long* type to be used as an uri BigNumber.

```
@TypeSystem({long.class, URIBigNumber.class, boolean.class,
String.class, URIFunction.class, URINull.class})
```

Listing 14: URI Type system.

Nodes.access

The **uri.nodes.access** package contains the classes allowing to read and write properties to objects using a Truffle Polymorphic inline cache. This is a step we made towards a fully OOP language and will be explained later.

- *URIReadPropertyNode* The node for reading a property of an object. When executed, this node:
  1. evaluates the object expression on the left-hand side of the object access operator;
  2. evaluates the property name;
  3. reads the named property.
- *URIWritePropertyNode* The node for writing a property of an object. When executed, this node:
  1. evaluates the object expression on the left hand side of the object access operator;
  2. evaluates the property name;
  3. evaluates the value expression on the right hand side of the assignment operator;
  4. writes the named property.

Nodes.call

The **uri.nodes.call** package contains the class *URIInvokeNode*, used for function invocation. Since uri has first class functions, the target function can be computed by an expression. This node is responsible for evaluating this expression, as well as evaluating the arguments. The actual dispatch is then delegated to a chain of *URIDispatchNode* of the same package that form a polymorphic inline cache.

Nodes.controlflow

The package **uri.nodes.controlflow** contains nodes different controlflow features:

- *URIBlockNode* statement node that just executes a list of other statements.
- While loops are implemented using the following classes:
  1. *URIWhileNode* statement node that uses a Truffle *RepeatingNode* for optimization reasons;
  2. *URIWhileRepeatingNode* the loop body of a while loop. A Truffle framework LoopNode between the URIWhileNode and URIWhileRepeatingNode allows Truffle to perform loop optimizations, for example, to compile just the loop body for long running loops;
  3. *URIBreakNode* allows breaking out of a loop by sending an exception to the parent *URIWhileNode*;
  4. *URIContinueNode* allows continuing execution, bypassing further instructions in the loop by sending an exception to the parent *URIWhileNode*.

  The following code illustrates the inner workings of a while node in the language.

```
public boolean executeRepeating(VirtualFrame frame) {
    if (!conditionNode.executeBoolean(frame)) {
        /* Normal exit of the loop when loop condition is false. */
        return false;
    }
    try {
        /* Execute the loop body. */
        bodyNode.executeVoid(frame);
        /* Continue with next loop iteration. */
        return true;
    } catch (URIContinueException ex) {
        /* Continue with next loop iteration. */
        return true;
    } catch (URIBreakException ex) {
        /* Break out of the loop. */
        return false;
    }
}
```

- *URIIfNode* a statement that evaluates the condition, executes the then-branch on success or the else-branch otherwise. Note that with our grammar the else-branch is optional.

– *URIReturnNode* throws an exception to the parent function node, the exception carries the return value or the type URINull if return is used without an argument.

– *URIFunctionBodyNode* the body of a user-defined function. It handles the return value of a function: the return statement throws an *URIReturnNodeException*. If the method ends without an explicit return, return URINull type.

Nodes.expression

The **uri.nodes.expression** package contains nodes for common language expressions.

– Constant literals, they are self-explanatory: *URIBigIntegerLiteralNode*, *URIFunctionLiteralNode*, *URILongLiteralNode*, *URIStringLiteralNode*.

– Arithmetic operators:

1. *URIAdd* contains 3 Truffle specializations: it can add primitive *long* numbers, *URIBigNumber* number and it can do string concatenation using a custom Truffle guard. The following code demonstrates this:

```
@Specialization(guards = "isString(left, right)")
@TruffleBoundary
protected String add(Object left, Object right) {
    return left.toString() + right.toString();
}
```

Listing 15: Truffle guard for string concatenation.

2. *URIDivNode*, *URIMulNode*, *URISubNode* all have the same behaviour as java.

– Equality operators:

1. *URIEqualNode* contains specializations for all supported types in our languages. Even functions are supported as they are registered using a canonical *URIFunction* object per function name in the function registry. Note that we do not support the inequality as the equality operator can just be negated using the *URILogicalNotNode*.
2. *URILessOrEqualNode*, *LessThanNode*, behave like their Java counterparts. Note that the greater than and greater or equal than are implemented by negating *URILessOrEqualNode* and *URILessThanNode* respectively.

– Logical operators: *URILogicalAndNode* and *URILogicalORNode* both extend *URIShortCircuitNode*. Logical operations in uri use short circuit evaluation: if the evaluation of the left operand already decides the result of the operation, the right operand must not be executed.

– *URIParenExpressionNode* represents parenthesized expressions; returns the value of the enclosed expression.

Nodes.interop

The **uri.nodes.interop** package contains a class used for converting foreign primitive values to uri values. *URIForeignToURITypeNode*. It notably converts characters to strings using *String.valueof()*. It is used to allow interoperability with other GraalVM guests language, meaning uri can coexist with languages such as javascript, ruby and R.

Nodes.local

The **uri.nodes.local** package contains nodes used to define scopes and handle function variables.

– *URILexicalScope* node to define a lexical scope in the language, it can be a block scope or a function scope.

– *URIReadArgumentNode* node that reads a function argument. Arguments are passed in as an object array.

– *URIReadLocalVariableNode* node to read a local variable from a function's frame. The Truffle frame API allows to store primitive values of all Java primitive types, and Object values. This means that all uri types that are objects are handled.

– *URIWriteLocalVariableNode* node to write a local variable to a function's frame. The Truffle frame API allows to store primitive values of all Java primitive types, and Object values.

Builtins

The **uri.builtins** package contains all of the language's built-in functions such as *println*. All builtin nodes must extend the *URIBuiltinNode*: it gives access to the the interpreter state and allows to execute code returning boolean or long or nothing. Builtin functions are all added in the function registry when the a program

is interpreted. A few interesting builtin functions we implemented are described below.

– *URIDefineFunctionBuiltin* allows defining a function from a string input which will be parsed in the same way as the script is parsed, meaning the same syntax applies. It is one of the meta-programming features of the language. Here's an example:

```
function foo() {
  println(test(40, 2));
}

function main() {
  defineFunction("function test(a, b) { return a + b; }");
  foo();
}
```

Listing 16: define-function.uri

– *URIEvalBuiltin* evaluates source code in any of the GraalVM supported languages, allowing for example to call JavaScript, R or Ruby from our language.

– *URINewObjectBuiltin* creates a new object. This is an OOP feature.

– *URIPrintlnBuiltin* and *ReadlnBuiltin* write or read from the console, respectively.

To demonstrate how the builtin functions are designed with Truffle in our language, let's add a new one. We want to be able to measure time in our language for various benchmarks. We know that Java provides this feature with *System.nanoTime();*.

1. We add a new builtin class to our language in the **uri.builtins** package:

```
@NodeInfo(shortName = "nanoTime")
public abstract class URINanoTimeBuiltin extends URIBuiltinNode {
    @Specialization
    public long nanoTime() { return System.nanoTime(); }
}
```

Listing 17: URINanoTimeBuiltin.java.

2. We make sure it gets added to the function registry at runtime, in *uri.runtime.URIContext.java*'s *installBuiltins* method:

```
installBuiltin(URINanoTimeBuiltinFactory.getInstance());
```

Note that the class *URINanoTimeBuiltinFactory* is generated automatically by Truffle. *installBuiltin* will take care of registering the function in the registry with its arguments so that it can be accessed like any other user-defined function in our language.

3. Profit!

```
function main() {
  before = nanoTime();
  println("Time before calculation " + before);
  println(1+2);
  after = nanoTime();
  println("Time after calculation " + after);
  println("It took " + (after - before) + "ns.");
}
```

Listing 18: nanoTime.uri.

Runtime

The **uri.runtime** package contains runtime classes for the language:

- *URIContext* the runtime state during execution, used by functions. It contains the function registry.
- *URIFunction* represents a function. On the Truffle level, a callable element is represented by a call target. This class encapsulates a call target, and adds version support: functions can be redefined at run time.
- *URIFunctionRegistry* manages the mapping from function names to function objects.
- types *URINull*, *URIBigNumber*.

Parser

The package **uri.parser** contains the ANTLR grammar, a factory to help generating nodes *URINodeFactory* and a helper class *URIParseError* as well as the two ANTLR-generated classes *URILexer* and *URIParser*. The parser is explained further in **Chapter 4.2**.

Putting it all together

The main entry point for the language is found in the top-level class *URILanguage* where the Truffle registration is found. The method *parse* is able to interpret code

snippets such as *println*(”2”); as well as source files containing functions and classes. A helper program *URILauncher* is used to handle user input and start the parser. The following code shows how to launch the interpreter through GraalVM's Polyglot APIs. Error checking is omitted.

```
if (args[0] == null) {
    org.graalvm.polyglot.Source source = Source.newBuilder("uri",
        new InputStreamReader(System.in), "<stdin>").build();
} else {
    org.graalvm.polyglot.Source source = Source.newBuilder("uri",
        new File(args[0])).build();
}
org.graalvm.polyglot.Context context =
    Context.newBuilder("uri").
    in(System.in).out(System.out).options(options).build();
org.graalvm.polyglot.Value result = context.eval(source);
System.out.println(result.toString());
```

Listing 19: URILauncher.java

To conclude this section, we feel like the Truffle framework is well documented enough that implementing nodes for our language was a task that ended up being a good learning experience albeit sometimes referring to other projects such as TruffleRuby for inspiration on basic node operations, such as making sure we didn't forget about operator precedence or basic control flow principles. A striking example was the omission of the *else* block in the *if* statement three months into the development, later painfully realizing this when a basic test would simply not work.

4.2 Implementing the parser

In this section we discuss how we make the parser we generated from our ANTLR grammar produce an AST to interpret programs using the nodes we defined in the previous section.

The **uri.parser** package is structured as follows:

- ANTLR files: **URI.g4** the grammar itself, **URIParser.java** the ANTLR-generated URI parser and **URILexer.java** the ANTLR-generated URI lexer;
- Our Java classes: **URINodeFactory** a factory for the nodes to use in the parser, this makes the grammar file easier to follow and **URIParseError** an error class to be used in the parser.

Everytime we modify the grammar, we must run the ANTLR program to generate a new version of the parser and lexer files. As described in the background chapter, extending the grammar to accept classes was easy thanks to ANTLR's IntelliJ plugin. We also made sure that the grammar would also accept programs from our previous language iteration to stay backwards-compatible. ANTLR links rules with methods from our node factory to generate the node links forming the AST. The node factory contains the methods to generate all of the nodes. For example, when parsing a function, our interpreter will run the following methods:

- *startFunction* which will create a new frame descriptor, initialize an empty list of method statement nodes and run *startBlock*;
- If there are arguments, the *addFormalParameter* method is run, adding arguments as assignment nodes to local variables in the function, adding to the list of method statement nodes;
- *startBlock* initializes a new lexical scope;
- when the method is finished, meaning there are no more statement nodes to parse, the *finishMethod* method is run, taking a list of body statement nodes from the method *finishBlock*, checking for errors initializing a function body node for the method nodes and finally a root node for the function.

Other nodes are parsed in a similar way, ANTLR and Truffle make it very easy to generate the AST. We will not describe the parser further as its developement was not of particular methodological interest, which is our main focus.

4.3 Implementation of the object-oriented paradigm

In this chapter we discuss the features we want to implement and then show how it was done practically.

4.3.1 Incremental development

Now that the time has come to implement OOP features in our language we must find out how Truffle will allow us to do so without reinventing the wheel. A quick look in the Truffle Javadoc[1] shows us two packages of interest **com.oracle.truffle.api.object** and **com.oracle.truffle.api.object.dsl** we haven't used yet. They both seem to contain classes to facilitate dealing with objects, such as *Shape*, *Property*, *Layout*. They most probably use the OSM used to store objects we defined in **Chapter 2**. In the section below we look further into these Truffle features and how we used them to implement objects in our language.

4.3.2 Object-oriented concepts

We define the object-oriented paradigm as follows: a programming paradigm based on the concept of objects which can contain properties in the form of fields and methods which can modify the object's fields with which they are associated by using the notion of "self". Objects in our language are instantiations of classes which define their type. We also want to make sure we don't break the existing language features.

The following code snippet shows usage of the object properties feature, the first object-oriented feature we mean to implement. This program should print 12.

```
class A { num = 12; }
def main() { a = new(A); println(a.num); }
```

Listing 20: class-field.uri

To implement this, we need to be able to store properties and methods in objects. The relevant code for object handling in the source of our language is found in the **uri.nodes.access** package. Truffle's OSM(Object Storage Model) provides an API

[1] https://www.graalvm.org/truffle/javadoc/overview-summary.html

to deal with objects which we can use it to implement our object storage. The API code relevant to our interests is found in the **com.oracle.truffle.api.object** package.

- *Shape* objects create a mapping of *Property* objects to *Locations*.
- *Shape*'s *newInstance()* method returns a *DynamicObject*, an abstract class extending *Object* from Java and implementing *TruffleObject*.
- *Property* objects represent the mapping between property identifiers (keys) and storage locations.
- *Location* objects represents the data.

Storing and accessing values in the object requires using *Shapes*'s *defineProperty(Object key, Object value)* and *getProperty(Object key)*.

With *Shape*'s *getPropertyList()* method we are able to extract information about the data encapsulated within an object, static information of the object's identity, we can then define a registry *URIClassRegistry* of user-defined classes associated with their name to use when instantiating a new object. Calling $a = new(A)$; will return a new *Shape* object with the properties of A, as per *URIClassRegistry*'s *lookup(String name)* method. a will then have fields described in the class definition of A and it's methods.

Now that we have described how we can store and access properties we must now make the interpreter parse classes, evaluate member statements to populate the class registry. When running the interpreter the parser simply returns all functions analysed and the interpreter creates a call target on the main function wrapped around an *URIEvalRootNode* taking care of registering all of the functions and then calling the main function.

The classes now need to be registered as well. *URIEvalRootNode* is the root node, meaning it will be the first code run trickling down along the program tree's nodes.

For the class evaluation to work as expected, we must define new root nodes around the classes to be able to execute member definition expressions, which will be executed in the lexical scope of the class like a function. The member expressions will result in variables that are defined in a *VirtualFrame*, a frame containing local variables. We can extract information from each frame slot to give identity to classes in the class registry. We must also make sure to maintain a function registry for each class. Conceptually, we have a good idea of how to implement basic object feature, now let's take a look at the implementation.

4.3.3 Parser

The parser needs to be updated to handle parsing classes. Our grammar is already able to scan classes as expected and described in **Chapter 4.3**. The *URINodeFactory* class must be updated to be able to accomodate for class parsing. For each class, the parser will start by calling *startClass*, which will maintain the class' position in the source, a new *FrameDescriptor* to keep fields and methods, a list of *URIStatementNode* for all member definition statements, the method will then call *startMemberBlock* which will simply create a new lexical scope.

At the end of the class, the method *finishClass* will be called, it'll call *finishMemberBlock* which takes all the statement nodes in between and creates an *URIFunctionBodyNode*, except we don't care about return values, which it normally handles. *finishMemberBlock* also takes care of throwing exceptions when nodes different than variable assignment are parsed. Indeed we restrict the class member definition to look like the following:

```
def foo() { println("foo"); }
class A {
    a = 1; b = "2"; c = a; d = foo;
}
```

Listing 21: Class members.

It also takes care of parsing and registering functions with the frame of the class added to its own frame, to give them the proper method functionality commonly found in the OOP paradigm. Function nodes are not executed like member expressions, they are kept simply as literals with access to its class members.

With the new function body node consisting of member expressions, we can now let the parser work on the rest of the source, other functions and classes.

4.3.4 Main root node

The main root node must be changed to execute class member statements before running the code in the function main. For this we extend the *URIEvalRootNode* constructor to take a map of class names associated with their data as assigned by the parser. In *execute*, for each class, we call the function body node consisting of all member definitions. Once it is executed we can extract fields and methods to register

the class in the *URIClassRegister*. When all classes and functions are registered, we can call the main function node.

4.3.5 New object

As we have created new syntax for creating objects (eg. $a = new(A);$), we must create a new built-in function $new()$ that will take care of instantiating a new object by looking into the class registry. We have described how to create built-in functions in **Chapter 4.1.1**.

4.3.6 Class registry

The class registry has a method *lookup(String name, Boolean createIfNotPresent)* that returns a new *Shape* object with the fields and methods associated with the class named in argument in the class map. The method can also be used to create a new empty class entry in the registry as per the $createIfNotPresent$ boolean argument.

4.3.7 Missing features

Our language may contain objects but we didn't implement class hierarchy, visibility operators and volatility status. These features and the reason for their absence will be further described in **Chapter 4.4.1**.

Truffle does not provide an API to deal with object hierarchy and excpicitly state in the documentation that this is deliberate. We feel this is a shortcoming of Truffle as implementing object hierarchy is a burden on the developer for a feature that is common amongst language supporting the OOP paradigm.

4.3.8 Wrapping up

To conclude this section, we can say that the framework Truffle allowed us to achieve our goals of implementing the OOP paradigm in an incremental way. This is greatly facilitated by the great design of the framework. Although it sometimes had a steep learning curve in the early days of our interpreter, using the OSM and the object related API described in the section above was refreshingly intuitive.

4.4 Revisiting the case study

Now that we have implemented the OOP paradigm, let's take a look at the missing features we had defined in the case study in **Chapter 3**.

4.4.1 Object-oriented programming features

Some critical OOP features are absent from the current implementation of our language. Class inheritance was left out as deemed too complicated for the scope of this thesis. Access modifiers (private, protected, public) were not implemented but can easily be represented as additional attributes in a Truffle Shape. Non-access modifiers (static, final, volatile) can be modelled with corresponding storage locations. Multi-threading was also not implemented as Truffle's OSM does not guarantee the thread safety of shape transitions. The GraalVM team is working on a Java implementation using Truffle[6].

4.4.2 Metaprogramming features

There are three features we didn't implement yet. First is variable access through methods like *var_get* and *var_set*. This is very easy to implement as we already simply fetch object variables by key in the OSM as described in the previous section. Similarly, method sends can be easily implemented since we store object methods in a registry where the method name is the key in the map, stored as a string.

Finally, hook methods also are easy to implement. For the *method_missing* example, as we can simply modify our class function registry to return the *method_missing* when calling a missing method.

4.4.3 Case study status

To conclude this chapter, we feel like we have implemented enough interesting features that make our research impactful. There are a myriad of other features we could have implemented from either paradigms we have studied and partly implemented in our language but this work focuses on the incremental nature of the development and we feel like what we have implemented sufficiently pertains to demonstrating Truffle's great design and usability.

Chapter 5

Advanced Graal features

Although our language now has the features we defined in **Chapter 3** and illustrated in **Chapter 4**, there are a few things we can still explore. GraalVM and Truffle offer advanced features for implementing dynamic languages that we feel are worthwhile. In this chapter we explain such features and extend our language with them.

5.1 Polymorphic inline cache

Performance is critical in our language. Indeed, we make use of the GraalVM JIT, the future compiler for all Java platforms which is proven to be fast. Truffle provides an API to cache object values, making access faster. This feature is called a polymorphic inline cache[14]. This technique is commonly used in dynamic languages to achieve a speedup on dynamic property lookup by speculating on the operation stabilising such as the property staying the same type.

5.1.1 Concept and implementation

In our language we can achieve a notable performance gain as the classes are statically typed. Inline caches in Truffle are implemented as nodes in the interpreter's AST[6]. Nodes form a chain of nodes with their Truffle specialization differentiating the types. This chain forms the inline polymorphic cache.

Consider the expression $obj.x$ in our language. The interpreter will return an object read property node *URIReadPropertyNode* where the two children node will be the receiver node obj and the name node x. With an unitialized cache, the node will simply get the value object from the receiver node. The following code and figure below show the case of an uninitialized cache.

```
@TruffleBoundary
@Specialization(replaces = {"readCached"})
protected Object readUncached(DynamicObject receiver, Object name) {
        Object result = receiver.get(name);
        if (result == null) // throw exception
    }
    return result;
}
```

Listing 22: Read uncached property.

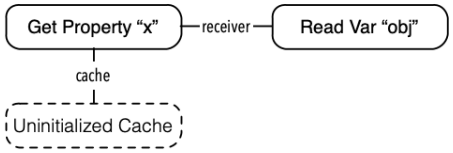

Figure 5.1: Property read with an uninitialized cache.

The node is then inserted in the cache to form a monomorphic inline cache.

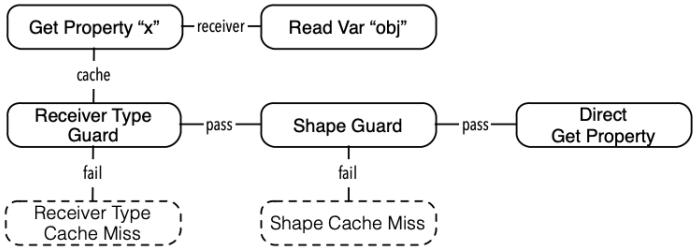

Figure 5.2: Property read with a monomorphic cache.

If the shape guarding check fails meaning the object type is unexpected the cache miss handler inserts a new link in the chain forming a polymorphic inline cache.

The code below shows how the node reads a property when it is cached. Note the use of the previously discussed Truffle *Location* class.

```
@Specialization(limit = "CACHE_LIMIT",
    guards = {"namesEqual(cachedName, name)",
        "shapeCheck(shape, receiver)"})
protected static Object readCached(DynamicObject receiver,
        Object name,
        @Cached("name") Object cachedName,
        @Cached("lookupShape(receiver)") Shape shape,
```

```
        @Cached("lookupLocation(shape, name)") Location location) {
    return location.get(receiver, shape);
}
```

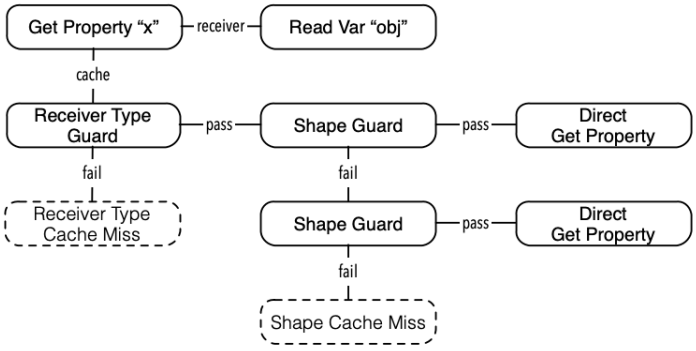

Figure 5.3: Property read with a polymorphic cache.

The number of cache entries must be small in order to ensure positive performance improvements here, the variable CACHE_LIMIT is set to 3. Whenever a chain of nodes is bigger than that limit, the inline cache is considered megamorphic and is rewritten to an indirect dispatch node. The node loads the operation from the object shape and calls it.

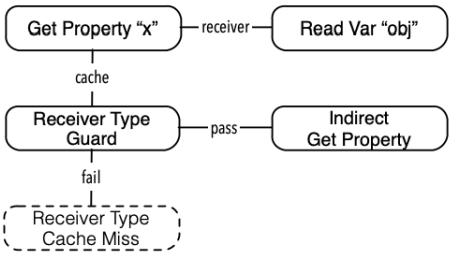

Figure 5.4: Property read with a megamorphic cache.

We also used Truffle's caching capabilities in other places of our language, here are a couple of notable examples:

- *URIEvalBuiltin* evaluates code from any GraalVM supported language. The call target is cached against the mime type and the source code, so that if they are the same each time then a direct call will be made to a cached AST, allowing it to be compiled and possibly inlined.
- *URIDispatchNode* caching the call target of a function allows Truffle to perform method inlining, a pretty important performance improvement.

5.1.2 Evaluation

We collected data to illustrate the impact of polymorphic inline caches on the performance of our language. The following code is inspired from a Ruby fork repository proposing an implementation of an inline polymorphic cache in 2011. It was then of course rewritten in uri.

```
class C1 {
  def m() {
    return 1;
  }
}
class C2
  def m() {
    return 2;
  }
}

def main() {
    o1 = new(C1);
    o2 = new(C2);

    i=0;
    while (i < 6000000) {
        if (i % 2 == 0) {
            o = o1;
        } else {
            o = o2;
        }
        o.m; o.m; o.m; o.m; o.m; o.m; o.m; o.m;
        i+=1;
    }
}
```

Listing 23: benchmark-pic.uri

This test is relevant with regards to what we are trying to benchmark because the object o will change shapes numerous times. When running without the polymorphic inline cache, after running 50 times, the mean total time is $\approx 2.4s$. With the cache, the mean time is $\approx 1.94s$, a $\approx 20\%$ improvement! Interestingly, the Ruby developer behind the test above shared similar results. Monomorphic caches also maintain the same performance, meaning the use of our cache didn't deteriorate the performance of non-cached accesses.

5.2 Polyglot

The GraalVM Polyglot API lets us embed and run code from guest languages in JVM-based host applications. We took interest in this novel feature early in the development process, the reason being twofold.

First, testing: being able to embed uri code in Java greatly helped us during the testing, which is described in detail in **Chapter 6.2**. We made use of the **org.graalvm.polyglot** API, allowing to embed Graal languages in Java applications.

Secondly, language coexistence: by making sure that the uri types are compatible with other GraalVM languages and inversely through the use of the **com.oracle.truffle.api.interop** API. For example our language does not define a character type, in our **uri.nodes.interop** package, the *URIForeignToURITypeNode* class does the following:

```
@Specialization
protected static Object fromChar(char value) {
    return String.valueOf(value);
}
```

To make sure that characters can be passed to our language, as strings, which uri supports. Objects in our language are compatible because we made use of the Truffle Dynamic object storage model described thouroughly in **Chapter 4.3**.

The code below shows how we can embed uri code in a Java program.

```
import org.graalvm.polyglot.*;
import org.graalvm.polyglot.proxy.*;

public class HelloPolyglot {
    public static void main(String[] args) {
        System.out.println("Hello Java!");
        try (Context context = Context.create()) {
            context.eval("uri", "def main() {"
                + "println('Hello Uri!')"
                + "}");
        }
    }
}
```

Listing 24: uri-interop-java.java

We can make our language and Java work together inversely, with the Polyglot API we can provide Java objects to uri in runtime.

There are many other examples of Polyglot usage illustrated on the GraalVM website[1], for example being able to reliably cancel uri tasks after a timeout interval, useful if the code is not trusted (an infinite loop will be canceled). The code listing below shows an example of this.

```
try (Context context = Context.create()) {
    context.initialize("uri");

    Timer timer = new Timer(true);
    timer.schedule(new TimerTask() {
        @Override
        public void run() {
            context.close(true);
        }
    }, 1000);

    try {
        String lessTrustedCode = "def main() {" +
                                 "while(true) {"; +
                                 "}";
        context.eval("uri", lessTrustedCode);
        assert false;
    } catch (PolyglotException e) {
        assert e.isCancelled();
    }
}
```

Listing 25: not-trusted.uri

We though this advanced feature was cool and unique so we made effort in implementing it for our language. In retrospect the great GraalVM documentation made it quite simple and it ended up being very useful for our testing environment.

To conclude this chapter, we learned a lot about exciting features to use in our language by perusing papers and documentation. We will hint at possible further developments in the conclusion of this thesis.

[1] https://www.graalvm.org/docs/reference-manual/embed/

Chapter 6

Difficulties and validation

6.1 Difficulties Encountered

We hit a few bumps along the road on the way to completing this work. Implementing an interpreter with the tools we chose was not an easy task! This chapter highlights some of the difficulties encountered; they demonstrate situations common in the world of language and software engineering in general. These problems were, in retrospect, valuable lessons for a future software engineer.

6.1.1 The ups and downs of working with young software

As the title implies, the work for this thesis gives us experience in using young software. When we started our work the GraalVM project we used as the framework for our programming language development was only in its first pre-release and there were only a handful of conference papers about it. Since then it has almost reached a first stable version, a lot of documentation has been written and a neat user-friendly website[1] was designed. At the time of writing this document GraalVM has reached release candidate 16, a whole lot of versions and 11631 commits in roughly 13 months. This means a lot of features have been added, a ton of bugs fixed, all of it towards the maturation of GraalVM to a stable framework.

It has been a challenge following its development, as breaking updates were common. This isn't surprising for the release candidate model that Oracle follows for GraalVM. The following section describes notable changes in the last year concerning the GraalVM SDK and Truffle which we use from the Graal repository that impacted the development of our language:

[1] **https://www.graalvm.org/**

Added support for logging in Truffle languages and instruments.

This was a useful new feature when developing new languages. For example we added the following code to report when a function is being interpreted in uri/runtime/URIFunction.java:

```
private static final TruffleLogger LOG =
TruffleLogger.getLogger(URILanguage.ID, URIFunction.class);
// in setCallTarget method:
LOG.log(Level.FINE, "Installed call target for: {0}", name);
```

This feature was used in a few nodes to help with debugging and testing.

Added new execution listener API that allows for simple, efficient and fine-grained introspection of executed code.

Execution listeners allow to instrument the execution of guest languages. For example, it is possible to attach an execution listener that is invoked for every statement of the guest language program, similar to how a debugger would single-step through the program.

```
Context context = Context.create("uri");
ExecutionListeners listener = ExecutionListeners.newBuilder()
          .onEnter((e) -> System.out.println(
                  e.getLocation().getCharacters()))
          .statements(true)
          .attach(context.getEngine());
context.eval("uri", "i = 0; while (i < 2) { i++; }");
listener.close();
```

The above code should yield the following result:

```
    i = 0
    i < 2
    i++
```

Again, this helped us debug our parser and interpreter greatly during development since we could introspect various parts of our language such as statements, expressions or roots of methods, functions.

Added static member to class objects that provides access to the class's static members.

This allows getting class static members directly, as shown in the following unit test:

```
final Source src = Source.newBuilder("uri",
    "def getValue(type) {" +
    "o = new(type); o.a = 10; " +
    "return o.value;}",
    "testObject.uri").buildLiteral();
context.eval(src);
Value getValue =
    context.getBindings("uri").getMember("getValue");
Value ret = getValue.execute(new TestType());
Assert.assertEquals(20, ret.asLong());
```

This feature was very useful when we first implemented objects in our language, as we could verify objects and their properties were working as intended.

Major Refactoring

In february, a new version released included a major revision of the Truffle interoperability API, Graal SDK was renamed to GraalVM SDK, the option to set the encoding when loading source code was added. The changes are far too many to list and the resulting code difference is of 5109 lines! The full list is available on the GraalVM Github release notes.

The impact on our code was big as their refactoring was extensive. We ended up needing a couple of days simply to rename classes and changing Truffle syntax around, compile testing, again and again until all our tests ran as expected.

We decided to freeze the code at version RC15 to avoid breakage after early May 2019 when the work should be about finished. At the time of this writing, three more release candidates were released. This was a good lesson in the difficulty of following immature software as it can lead to a lot of maintenance work. On the other hand, the upside of this process is getting exciting new features to work with as described above.

6.1.2 Truffle difficulties

Working with Truffle was not without its hiccups; we can criticise the design choices behind some of the features in the framework. Some issues we had can be attributed to our misunderstanding of a very large framework, others to what we feel are poor design choices from the team of developers.

Object features

The Truffle Javadoc specifies explicitly that it does not provide an API to handle object hierarchy; a feature we counted on as basic inheritance is paramount to the OOP paradigm. The Truffle developers describe how it is a task that the programmer should take care of, we feel it is disingenuous to offer objects in an API made to develop dynamic languages and expect us to implement object hierarchy given that the whole point of Truffle is getting rid of a plethora of homegrown object system implementations.

Sparse documentation

In the early days of the development Truffle was still in its infancy therefore there was not much in the way for us to understand how to use the framework. Documentation was sparsely scattered among developer papers, the official Javadoc and a couple of blogs. Since then, at the time of writing this, the developers have launched a website with more documentation and made the Javadoc for their APIs more rich. This is of course a problem related to the young age of the software as described in the previous section.

API complexity

We feel that some parts of the Truffle API are over-engineered. This may be explained by the fact that the developer team behind Truffle also works on GraalVM, a whole compiler with an immense amount of complexity. An example that we feel illustrates this is the way Truffle registers nodes we defined in Java. First, we must create an abstract class extending a Truffle *Node*, such as a *DivNode* where we must use the following Truffle annotations:

- @NodeInfo(shortName = "/"), the syntax for a division node;
- @Specialization's for types where we want to allow division to occur: long's and *URIBigNumbers*, methods that perform the division and error checking;
- @Fallback to make sure a node is still reached when trying to divide unsupported types.

So far so good, it is as we described in the introduction of this thesis. Now if we want to use the class to produce a node we can't directly instantiate it since it is abstract. Truffle will actually generate a class called *DivNodeGen* with a constructor for the node and a few unreadable methods handling with polymorphic specializations. We can then use *DivNodeGen.create()* with the arguments being the left and right hand side arguments for the division operator.

Truffle generating code is nice because it takes care of a lot of boilerplate code by itself. This becomes an issue when a bug appears as having to inspect machine-generated code is much more difficult, though the framework tries to insert comments to aid debugging. The following code snippet shows an extract of the method *executeGeneric_generic1* part of the generated *DivNodeGen* class, serving as type check to execute the appropriate division method.

```
if ((state & 0b10) != 0
    /* is-active div(URIBigNumber, URIBigNumber) */ &&
    URITypesGen.isImplicitURIBigNumber((state & 0b11000) >>> 3
    /* extract-implicit-active 0:URIBigNumber */,
    leftNodeValue_)) {
    URIBigNumber leftNodeValue__ =
        URITypesGen.asImplicitURIBigNumber((state & 0b11000) >>> 3
        /* extract-implicit-active 0:URIBigNumber */,
        leftNodeValue_);
    // more unreadable code...
}
```

As we can see, the code is quite cryptic. This caused us grief when we had a bug in the method reading a property from an object; we had forgotten to use a guard to make sure that the property name was properly matched against the one in the object's cache as property names can be arbitrary URI objects.

Thread safety

As documented in the official Truffle Javadoc, the objects API we have used does not provide thread-safe objects. This feature is still missing as of the time of writing this thesis; this blocks the ability to implement threading in our language as we cannot offer a broken feature.

To conclude this section, albeit we voiced strong opinions about some of the design choices we feel like the framework was a fit for the goals we achieved. We think Truffle is essentially a one of a kind all around Swiss knife tool for creating interpreters.

6.2 Testing a language

A very crucial part of implementing an interpreter for a new language in an incremental manner is making sure even the most basic features are not broken every time we add new things. In this chapter we describe the testing infrastructure we setup to make development easier along the way.

6.2.1 Unit Tests

We used Maven as a build tool for the project along with its plugin SureFire allowing to execute tests before creating a jar archive and deploying. It is also used to produce an XML or text file report of the testing phase that contain stack traces and may be used by Continuous Integration (CI) platforms such as Jenkins. The testing phase takes a few minutes to execute fully, which Maven allows to easily disable when we want to check whether our Java code simply compiles, saving a significant amount of time.

6.2.2 Program Validation

The folder *tests* at the project root contains 35 URI programs and their output. In the *error* subdirectory 17 other URI programs can be found. The former 35 programs are pass tests as described in the introduction, meaning they are expected to run successfully and return the same output as in the output file. The fibonacci example below illustrates this. The main function runs fib(i) with i ranging from 1 to 10, omitted from the code listing below.

```
def fib(num) {
  if (num < 1) {return 0;}
  n1 = 0;
  n2 = 1;
  i = 1;
  while (i < num) {
    next = n2 + n1;
    n1 = n2;
    n2 = next;
    i = i + 1;
  }
  return n2;
}
```

Listing 26: fib.uri.

The test case URISimpleTestSuite uses URITestRunner to create JUnit test cases dynamically. It first load the .uri program and its corresponding .output file as expected output, then runs the program. The following code snippet shows the crux of the URITestRunner class:

```
context = Context.newBuilder().
    in(new ByteArrayInputStream(testCase.
    testInput.getBytes("UTF-8"))).out(out).build();
PrintWriter printer = new PrintWriter(out);
run(context, testCase.path, printer);
printer.flush();

String actualOutput = new String(out.toByteArray());
Assert.assertEquals(testCase.name.toString(),
    testCase.expectedOutput, actualOutput);
```

Listing 27: URITestRunner.

The 17 error programs are tested in the same way, meaning that the output file contains the expected error, as demonstrated by the brief example below.

```
def main() {
    break;
}
```

Listing 28: invalid-break.uri.

```
Error(s) parsing script:
-- line 2 col 3: break used outside of loop
```

Listing 29: invalid-break.output

Pass tests and fail tests make 54 test cases, running our Maven test target returns the following with SureFire.

Running uri.test.URISimpleTestSuite Tests run: 54, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.992 sec

In this section we showed a few examples of our testing strategy, an integral part of our development process. We feel like we have learned a lot about TDD when working on rapidly growing Java code bases. The amount of Truffle API to aid testing as well as good IntelliJ integration of JUnit made the testing bearable and the incremental development practical.

Chapter 7

Conclusion

7.1 Objectives

The goal of this thesis was to show that we can make compiler development incremental and manageable in time when the code base grows as features are added given that an appropriate set of tools are used. We showed that with the Truffle framework paired with the GraalVM compiler we could create an interpreter for a language that is built in a way that facilitates growth in a sane way. We called this incremental compiler development and chose to implement a new language with these tools to validate the relevance of our research question and the appropriateness of the tools considered to address that question.

We first analysed and motivated the choice of tools we were going to use to achieve this, before defining a case study where we decided which features we were going to implement in our language. This case study contained the addition of the OOP programming paradigm as the main incremental growth. It also described a few metaprogramming features we felt would make our language nice. We then went into great detail how we used Truffle to implement our language, providing insight on the ease of the language's development as it grew. Next we described advanced features that we feel were worth implementing as they helped creating a good testing environment. Then we discussed the difficulties we encountered along the way and the validation strategies we set in place.

Not only did we manage to build the language while almost feature matching the case study, we described a way to make a difficult task such as writing a compiler easy and manageable. We showed that it is worth taking the incremental approach we came up with on a large code base with the tools Truffle and GraalVM which we vouch for wholly.

7.2 Future work

While we feel like we have fulfilled the case study feature-wise there are certainly more features our language could have. Although we did implemented objects, a few critical things were left out; class hierarchy, interfaces, object field visibility. These features are needed if anyone is going to use our language seriously. Secondly, the ability to execute concurrent code is very important when developing real-world applications. Thus, having threading would be great but Truffle does not offer thread-safe objects yet. Truffle offers instrumentalization features that could allow us to write an Integrated Developement Environment (IDE) with useful debugging features such as breakpoints. This could also be an interesting path of future work.

A way forward could be to take another incremental step by implementing a new paradigm altogether like logic programming, concurent programming, ... and have our language become multi-paradigm much like Oz. It is important to note that Truffle and GraalVM are still young, therefore many exciting new features will show up in future releases.

7.3 Lessons learned

The work that this thesis encompassed opened our eyes to the world of big Java code bases which was unknown to us before. It was a good learning experience as there were many roadblocks and failures that sometimes - more often than not though, were followed by success that happened after we spent sometimes days in documentation, papers or source code of other similar projects. We learned valuable lessons in software maintenance and evolution when we faced difficulties using a young framework that was very much a moving target or when documentation was not available or even when we realised, perhaps too late, that a feature we expected to be in the framework was not present.

Incidentally, our struggling allowed us to better understand the design of compilers as we wandered around source code for other languages such as Ruby, Perl6 and PHP.

To conclude this thesis, we think that the subject was worth our time as satisfying results were attained and hope that the readers will have got new insight from them. Thanks for reading.

Bibliography

[1] Michel Larabel. *GCC Soars Past 14.5 Million Lines Of Code*. URL: **https://www.phoronix.com** (cit. on p. 1).

[2] Matt Might. *What every computer science major should know*. URL: **http://matt.might.net/articles/what-cs-majors-should-know/** (cit. on p. 1).

[3] Stefan Heule. *How Many x86-64 Instructions Are There Anyway?* URL: **https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/** (cit. on p. 2).

[4] William Campbell, Swami Iyer, and Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. 2012 (cit. on p. 2).

[5] Thomas Wurthinger et al. "Self-optimizing AST Interpreters". In: *SIGPLAN Not.* 48.2 (Nov. 2012), pp. 73–82. ISSN: 0362-1340. DOI: **10.1145/2480360.2384587** (cit. on p. 8).

[6] Andreas Woss et al. "An Object Storage Model for the Truffle Language Implementation Framework". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '14. Cracow, Poland: ACM, 2014, pp. 133–144. ISBN: 978-1-4503-2926-2. DOI: **10.1145/2647508.2647517** (cit. on pp. 10, 36, 37).

[7] Matthias Grimmer et al. "High-performance Cross-language Interoperability in a Multi-language Runtime". In: *SIGPLAN Not.* 51.2 (Nov. 2015), pp. 78–90. ISSN: 0362-1340. DOI: **10.1145/2936313.2816714** (cit. on p. 11).

[8] Michael L. Van De Vanter. "Building Debuggers and Other Tools: We Can "Have It All"". In: *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '15. Prague, Czech Republic: ACM, 2015, 2:1–2:3. ISBN: 978-1-4503-3657-4. DOI: **10.1145/2843915.2843917** (cit. on p. 11).

[9] Thomas Wurthinger et al. “One VM to Rule Them All”. In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. Indianapolis, Indiana, USA: ACM, 2013, pp. 187–204. ISBN: 978-1-4503-2472-4. DOI: **10.1145/2509578.2509581** (cit. on p. 11).

[10] Aleksey Shipilev. 2018. URL: **https://shipilev.net/jvm-anatomy-park/18-scalar-replacement/** (cit. on p. 12).

[11] Lukas Stadler, Thomas Wurthinger, and Hanspeter Mossenbock. “Partial Escape Analysis and Scalar Replacement for Java”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’14. Orlando, FL, USA: ACM, 2014, 165:165–165:174. ISBN: 978-1-4503-2670-4. DOI: **10.1145/2544137.2544157** (cit. on p. 13).

[12] Tonis Pool. *Generic Reloading for Languages Based on the Truffle Framework*. 2016 (cit. on p. 13).

[13] Chris Seaton. *AST Specialisation and Partial Evaluation for Easy High-Performance Metaprogramming*. 2016. URL: **http://chrisseaton.com/rubytruffle/meta16/meta16-ruby.pdf** (cit. on p. 20).

[14] Urs Hölzle, Craig Chambers, and David Ungar. “Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches”. In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP ’91. London, UK, UK: Springer-Verlag, 1991, pp. 21–38. ISBN: 3-540-54262-0. URL: **http://dl.acm.org/citation.cfm?id=646149.679193** (cit. on p. 37).

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**