JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Faculty of Engineering
and Natural Sciences

Truffle/C Interpreter

Master's Thesis

submitted in partial fulfillment of the requirements
for the academic degree

Diplom-Ingenieur

in the Master's Program

Computer Science

Submitted by
Manuel Rigger, BSc.

At the
Institut für Systemsoftware

Advisor
o.Univ.-Prof. Dipl.-Ing. Dr.Dr.h.c. Hanspeter Mössenböck

Co-advisor
Dipl.-Ing. Lukas Stadler
Dipl.-Ing. Dr. Thomas Würthinger

Xiamen, April 2014

Contents

Abstract

Truffle is a novel framework for the implementation of programming languages, and allows to model interpreters based on an abstract syntax trees (AST). However, until now, this language had only been facilitated by dynamic languages, which highly profit from the run-time specialization offered by the Truffle concept.

The Truffle/C project aims to explore, how a statically typed language such as C, that has traditionally been compiled to machine code for execution, can be implemented using the Truffle framework, and how its implementation performs compared to the compiled approach taken by GCC.

This thesis presents an overview of the Truffle/C implementation and concepts. It explains, how Truffle/C uses a modification of Clang, to generate a Truffle/C file with a custom format and platform independent content from a C source file. The thesis also explains, how the Java side of Truffle/C reads this file and generates an executable Truffle AST from it.

The thesis shows how data types and data can be mapped from C to Java. This includes how unsigned primitives or structures can be implemented, that do not exist in Java. It presents, how Truffle/C implements the various operations, including the goto operation, that Java also does not support on the language level.

This thesis concludes with an evaluation on three parts. It consists of the evaluations of the performance on common benchmarks, the completeness with GCC test cases, and a comparison of the implementation of Truffle/C to the Truffle implementations for dynamic languages.

Kurzfassung

Truffle ist eine neuartige Sprache, mit der Programmiersprachen als ausführbare Abstrakte Syntaxbäume (ASTs) modelliert werden können. Implementierungen zielten bis jetzt darauf hin ab, dynamisch typisierte Sprachen ausführbar zu machen. Diese Sprachen profitieren nämlich zusätzlich von der Laufzeitspezialisierung, die Truffle mit sich bringt.

Mit dem Truffle/C Projekt versuchen wir C als erste statische typisierte Truffle Sprache zu evaluieren. Mit Truffle/C soll evaluiert werden, wie ein dynamischer Ansatz im Vergleich zu Compilern wie GCC abschneidet. Traditionell wird die Sprache vor der Ausführung zu Maschinencode übersetzt.

Diese Masterarbeit gibt einen Überblick über das Truffle/C Projekt. Die Arbeit zeigt, wie Truffle/C eine Modifizierung von Clang verwendet, um eine C-Datei in ein eigens dafür entworfenes Truffle/C-Format zu konvertieren. Weiters präsentiert sie, wie die Java Seite der Implementierung die Datei liest und daraus einen ausführbaren AST produziert.

Diese Arbeit stellt außerdem vor, wie Truffle/C die C-Datentypen und Datenstrukturen in Java nachbildet. Damit präsentiert sie auch, wie das Projekt die in Java nicht vorhandenen vorzeichenlosen numerischen Datentypen und Strukturen realisiert. Schließlich zeigt sie, wie Truffle/C die C Operationen und damit das in Java ebenfalls nicht vorhandene Goto implementiert.

Schlussendlich präsentiert die Masterarbeit, wie Truffle/C bei Geschwindigkeits-Benchmarks abschneidet, wie viele und welche der GCC Testfälle unterstützt werden und wie sich Truffle/C von den Truffle Implementierungen für dynamische Sprachen unterscheidet.

Chapter 1

Introduction

This chapter introduces basic vocabulary, to explain the motivation for implementing a C interpreter in Truffle. It gives an introduction to C and Java as the most relevant languages for Truffle/C. The chapter also delimits the scope and structure of the thesis.

"C is quirky, flawed, and an enormous success." [34]
Dennis Ritchie

1.1 Motivation

Since Dennis Ritchie first implemented C in the 1970s [34], researchers published numerous papers regarding different aspects of the C language and language implementers developed many C compilers until today.

However, like most prevalent language implementations, the optimizations for the construction of high quality code or fast execution are deeply rooted within the compiler or interpreter of the language. The language framework Truffle with the Graal compiler addresses this issue and enforces a separation of the implementation of the language semantics and the optimizations performed on it. The framework allows an implementation of a fast C interpreter, that does not have to be concerned with efficient code generation, intermediate representations, or optimizations.

Truffle allows to model a language implementation as an Abstract Syntax Tree (AST). An AST interpreter is an intuitive way to implement a language, since a language implementer can always

concentrate on implementing one specific operation as a node. Truffle is written in Java, hence allowing all the features Java offers. A language implementer can use the high level language constructs and libraries from Java to implement language semantics. She does not have to regard garbage collection since this is automatically performed by the JVM. It is easy to debug a language implementation, since programmers can use the debug infrastructure of Java. More generally, a language implementer can use all the mature productivity tools that are available for the Java language.

Truffle has been only evaluated on script languages so far [40], where languages can additionally profit from type specialization mechanisms. The goal of the implementation of Truffle/C is to evaluate Truffle as a framework for statically typed languages, specifically for C. Experimentation on a static language provides the Graal developers with feedback on the applicability of the optimizations on C.

1.2 Goals and Scope

The goal of this project is to implement a C interpreter that is written in Java and builds on the Truffle framework. For preprocessing the C code the project should use the Clang parser written in C++ to generate a simplified representation of a C source file. The interpreter should be capable to execute established C benchmarks, mainly from the "The Computer Language Benchmarks Game"[1].

Besides implementing the language, the thesis should identify which constructs of C can be modeled in Java.

Explicit non goals are completeness with respect to the C specification. Reaching completeness requires a development effort which exceeds the scope of a Master's thesis. However, this thesis shows that Truffle/C already supports a major subset of C. Another non goal is peak performance of executed code. Performance relies on the one hand on using Truffle in the intended way, on the other hand on the optimizations, that Graal performs. The benchmark results in the evaluation also show a comparable peak performance to GCC in some of the benchmarks.

1.3 From C to Java

C is a language that Dennies Ritchie first implemented in the 1970s as a system implementation language which is close to the machine [34]. While it is generally possible to write platform independent

[1] The Computer Language Benchmarks Game: `http://benchmarksgame.alioth.debian.org/`

C code, there are still platform dependent details. One example is the data types, which vary in C depending on the platform. Additionally, compilers usually first compile C code to assembler code, which is then platform specific. Java is more strictly specified and the language is platform independent, i.e., it can run on any platform where a JVM is supported. While Java follows the "Write once, run anywhere" paradigm, one can attribute "Write once, compile anywhere" to C, if the program only uses platform independent features. In contrast to C, the data types have a fixed bit width in Java and do not contain unsigned types. Java is a high level, object-oriented language, which was introduced by Sun Microsystems in 1995.

C does not have a strict type system. It is possible, to interpret any memory area as a value of a certain type, belonging to the program space. This is not the exception, but C programmers commonly use it to emulate subtyping [37]. In contrast, Java has a strict type system. An access in Java is always "safe"; the program theoretically cannot crash the JVM through illegal memory accesses [20] (which is not true for Truffle/C, since it uses the restricted `Unsafe` API).

C is usually first compiled to assembler code, which is then executed. Because of the simplicity of C and the operations which are close to the machine, compilers for the language can be very simple. However, C programs can be hard to optimize, since pointer analysis has to be used to determine aliasing. In contrast to Java, every pointer can theoretically point to any value. Without this analysis, many optimizations cannot be performed and the compiler has to place conservative assumptions [39]. Memory addresses can be computed, which complicates this problem. Java has different challenges of optimization which are not covered here. In contrast to C, Java programs usually run on a virtual machine. When the interpreter executes code paths often enough, the virtual machine lets parts of the program compile.

Since C is an old programming language and developers and standardization organizations tried to keep backwards compatibility throughout its existence, it has many deficiencies. In contrast to Java, C for example still supports the goto statement, which allows the program to arbitrarily jump to a location in the code. This statement is considered deprecated [14] in most code, however, a Java interpreter in C also has to support it.

The XJ311 committee released the first standard specification in 1983 [34] for ANSI C. Before that, only an informal book known as K&R served language implementers as a source. The next standard C99 in 1999 introduced new features and also made C stricter in some aspects. The newest standard since 2007 is C11.

An industrial quality interpreter or compiler cannot only target the latest standard. Compilers like GCC even support syntax and semantics of language features, that were supported in the K&R C time. Listing 1.1 shows an old function notation, which exists since K&R and which compilers such

as GCC still support until today.

```
int func(a)
int a;
{
    return a + 1;
}
```

Listing 1.1: Valid K&C Notation for Function Parameters

Additionally, the C specification allows freedom in some decisions. For example, the order of evaluation of subexpressions and the order of side effects are often unspecified [23, 6.5.3]. Listing 1.2 shows an example, where the industry quality compilers GCC and Clang differ. Clang outputs the value 2 for this example, while GCC outputs 1.

```
int main() {
    int a = 0;
    int res = a++ | ++a;
    printf("%d", res); // 1 or 2?
}
```

Listing 1.2: Function With Different Result in GCC and Clang

Truffle/C does hence not follow a standard but tries to support the test cases of GCC. Since Truffle/C supports features up to standard C99, this thesis uses the C99 standard to quote less ambiguous details.

1.4 Structure of the Thesis

Truffle/C is an on-going effort. Active developers on the Truffle/C project are my colleague Matthias Grimmer and me. We split our theses describing Truffle/C into two parts, as shown by Figure 1.1:

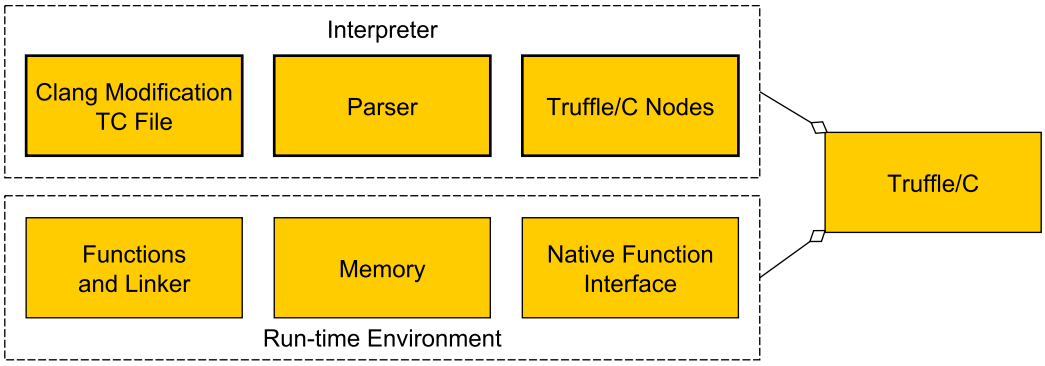

Figure 1.1: Split of the Work on Truffle/C

My colleague explains in his thesis [21] the runtime environment, shown in the lower half of the image. He explains the native function interface for Graal, which allows for efficient calls to functions from native C libraries from Java. Connected with that, he explains how we model functions in Java for calling Truffle/C and native functions. He also explains the memory model which we use in Truffle/C.

My part targets the interpreter and compile-time environment, shown in the upper half of the image. This thesis hence talks about how we generate Truffle/C files using Clang. It also presents how the parser processes the file to generate the Truffle/C AST. It explains the general architecture of Truffle/C and the nodes that implement the operations.

Some parts of the theses are overlapping. For example, this thesis cannot explain the construction of the nodes concerning the memory, without shortly talking about the memory concept. However, both theses tried to avoid explanations of overlapping details, where possible.

After the introduction in this chapter, Chapter 2 explain the major components that are involved in the development and during the execution of Truffle/C. Then, Chapter 3 briefly present the high level architecture of Truffle/C and also explain the roles of the components of the previous chapter related to Truffle/C. Chapter 4 to Chapter 8 give a more detailed view on Truffle/C. Chapter 4 deals with the Clang modification and the Truffle/C file, that it produces from the C file. Chapter 5 shows the implementation of the data types in Truffle/C. Chapter 6 bases on the previous chapter and explains the implementation of structures, unions, arrays, enumerations, literals and where and how the runtime stores them. Chapter 7 demonstrates, how Truffle/C builds nodes for binary and unary operations. Chapter 8 shows the approach, that Truffle/C takes for control structures and includes optimizations and branch probability profiling. Chapter 9 presents the components of Truffle/C again in a bigger picture, by revising some of the concepts in an example program. Chapter 10 presents the result of a performance, completeness, and an evaluation of Truffle/C as a Truffle language. Chapter 11 and

Chapter 12 present future work and related research. The thesis concludes with the findings of the project in Chapter 13.

Chapter 2

State of the Art

This chapter presents the main components, that are relevant for Truffle/C. It talks about the compiler Graal, the concept of the Truffle framework, as well as the compiler front end Clang.

2.1 Graal

Graal is a just in time (JIT) optimizing compiler written in Java. One possibility to facilitate it is through the Graal VM, which is a modification of the Java HotSpot VM, in which Graal is available as a compiler alongside client and server compiler.

Its intermediate representation (IR) is a graphed-based IR with the Java objects as graph nodes. A programmer represents the control-flow and data-flow dependencies between the nodes via Java annotations on fields of node classes.

Graal uses aggressive and speculative optimizations. For example, it cuts of cold branches that likely do not appear in the execution. When a speculative assumption is invalidated, a deoptimization returns control to the interpreter. To be able to do so, Graal saves information on where to return the execution and how to reconstruct the variables. [15, 16]

2.2 Truffle

Truffle is essentially a language for modeling and implementing languages using Java as a base. This means that a programmer can use the existing Java's standard libraries, debug infrastructure, memory management, and productivity tools to implement an own language. A programmer uses the Truffle framework to implement an executable abstract syntax tree (AST). This AST consists of nodes which have `execute` methods, that locally implement the operations. For a `double` addition in C, the implementation merely has to offer a simple addition node as Listing 2.1 shows. While AST interpreters are intuitive to implement, they have a high performance overhead because of the virtual method dispatches between the nodes [43]. Programmer should implement Truffle nodes in a specific way, so that they avoid boxing, record execution statistics and adapt to the execution state of the program. This not only allows more efficient interpreter performance, but improves later compilation, which Graal handles for Truffle in a special way.

```
@NodeChildren(value = {@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class DoubleAddNode extends DoubleNode {

        @Specialization
        public double executeDouble(double left, double right) {
            return left + right;
        }
    }
}
```

Listing 2.1: Truffle/C Node with Truffle DSL Usage

2.2.1 Rewriting and Specialization

Truffle AST nodes have a reference to their parent nodes and offer a `replace` method, with which a node can replace itself at its parent. This mechanism allows the AST to specialize on the current execution state. Generally, it allows the node to specialize on a subset of the semantics of an operation, that the node should handle. Dynamic languages use this mechanism heavily, as a variable can take on any type. This also implies, that an `add` operation for a dynamic language has be able to handle differently typed input operands and produce outputs of different types. Instead of handling all the different possible operations in one node, an AST node is expected to only handle those, which occur during the execution. An `add` node for a dynamic language might start as an uninitialized node, that can handle no case. If the node receives two integer operands, it would replace itself with a node, which only handles integer operands. Every execution it first has to check if the assumption, that both operands are integer, is still valid. If it is not valid any more, e.g., because the `add` operation

suddenly receives two string operands, it replaces itself with a more generic node, that can handle both or all cases. The goal is, that before compilation the AST becomes stable, i.e., stops performing rewrites, but at the same time contains only nodes that handle the semantic subsets for the current execution.

Another concept of specialization are polymorphic inline caches: Polymorphic inline caches are for example useful for virtual function calls and follow a similar concept as type decision chains for field accesses in dynamic languages. For a function call, a node starts again in an uninitialized state. When the node calls a function the first time, the node saves a function identifier as target. Upon the next call, it directly checks if the target is still this function. If this is the case, it saves a lookup. Otherwise, it would additionally save the second function to perform a chain of checks upon the next executions. The node extends the inline cache, until the checks are too many, i.e., the call is too polymorphic - and rewrites to a generic node that always performs a lookup.

For local variables, Truffle provides a `Frame` in which a guest language can save local variables. The language implementation addresses the `Frame` with a `FrameSlot`, through which the language can also specialize on reads and writes on the `Frame`. A language typically creates a `Frame` on every function call.

Truffle nodes usually avoid boxing in their execute method. While for the generic case, an operation might return an `Object`, specialized nodes for integer can directly return a Java **`int`** and thus avoid the boxing. To do so, a parent node expects the specialized version, e.g., an **`int`**, by executing the specialized method that returns an **`int`**. If the child node cannot return an **`int`**, e.g. because the operand types change, it throws an `UnexpectedResultException`, that the parent node catches to rewrite to the new data type, that the child node specifies.

2.2.2 Truffle DSL

Truffle DSL uses annotations on classes, fields, and methods, from which an annotation processor infers further classes. It relieves the programmer from having to write the boiler plate code, and allows her to concentrate on implementing the semantics. Since C is statically typed, it does not need some of the features like rewriting for type specialization. Thus, Listing 2.1 only has a single specialization for the **`double`** type, as the annotation tag `@Specialization` specifies. Apart from that, the two node children `@NodeChild("leftNode")` and `@NodeChild("rightNode")` produce `DoubleNode` classes.

In the Truffle/C project, the code generated by the annotation processor amounts to over 50,000 lines of code, while the code written by a programmer only makes up 24,000.

2.2.3 Control Flow

While Truffle/C often represents local control flow like loops or if statements directly using Java equivalents (see Chapter 8), a programmer should implement non-local control flow such as function returns or breaks as Java exceptions that inherit from `ControlFlowException`. Graal acknowledges this exceptions as normal control flow, and not as deoptimization points [42].

2.2.4 Profiling and Inlining

The Truffle AST can collect profiling information and should record, e.g., how often a loop inside a node executes or what branch probability a conditional statement has. This helps the Truffle framework to guide when it should inline a function and also helps Graal in generating optimized code [42].

Truffle performs inlining already on the AST level, to be able to react to profiling pollution. For dynamical or object-oriented languages, the arguments of a call can be polymorphic. If there are for example two call sites, where one of the callers always passes a string and the other always an `int`, the call would be polymorphic. However, when the code surrounding these function calls gets hot, Truffle clones and inlines the functions, which makes the call site monomorphic.

2.2.5 Partial Evaluation and Compilation

When the compilation eventually triggers, the Graal compiler will assume that the target AST is constant and performs no rewrites any more. It inlines all the `execute` methods of the target AST, replaces the rewrite logic with deoptimization points, and optimizes and compiles the code under this assumption. This inlining step could be considered to be a form of partial evaluation.

After partial evaluation, Graal processes the AST interpreter in its IR and performs additional optimizations such as inlining and in particular global optimizations. While the AST nodes only perform local optimizations, Graal can apply optimizations over whole ASTs [42].

Graal employs Escape Analysis to optimize away the allocation of the `Frame`, that the host language can use for storing its local variables, and connects the read of a variable to its last write. Thus, the array for storing this variables only exists virtually in the compiled code and produces no overhead [42].

If an assumption is invalidated, i.e., a node rewrite happens, a deoptimization point causes the compiled code to go back to interpreted mode, rewrite the node, and continue profiling until the AST is stable

again.

2.3 Clang

Clang is a front-end for the LLVM compiler and supports C, C++, Objective C, and Objective C++. It provides useful error messages, is fast and has a low memory usage. It targets to be compatible with GCC - also with its undocumented features - and hence fits well for Truffle/C's quest to be compatible with GCC [3].

Chapter 3

Architecture

This chapter provides a rough sketch of how the components of Truffle/C work together. It explains, how the Clang modification produces a Truffle/C file, and how the Java side processes it, to eventually generate the Truffle/C AST to be executed.

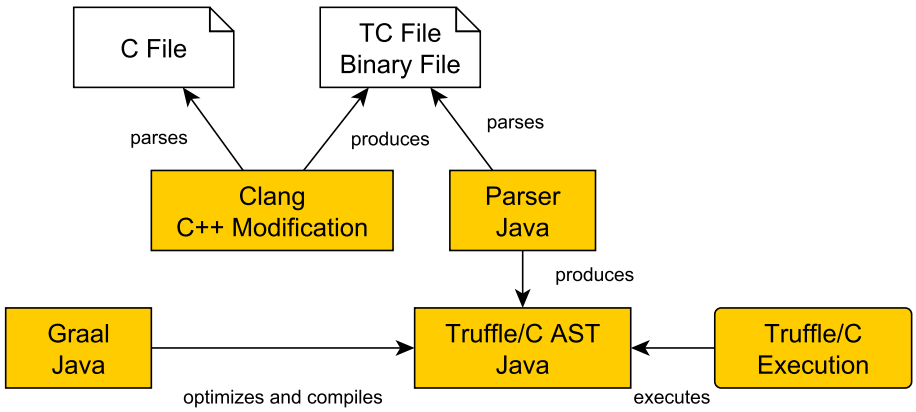

Figure 3.1: Truffle/C Architecture Overview

Figure 3.1 shows the workflow with the most important architectural components and artifacts of and around Truffle/C. The Clang modification first parses one or more C source or header files and generates an AST out of them. It writes this AST to a binary file with a custom Truffle/C format.

On the Java side, the parser reads one or more Truffle/C files and translates them into executable Truffle ASTs. When Truffle decides, that parts of an AST have been executed often enough, it lets Graal optimize and compile these ASTs.

3.1 From Clang to Java

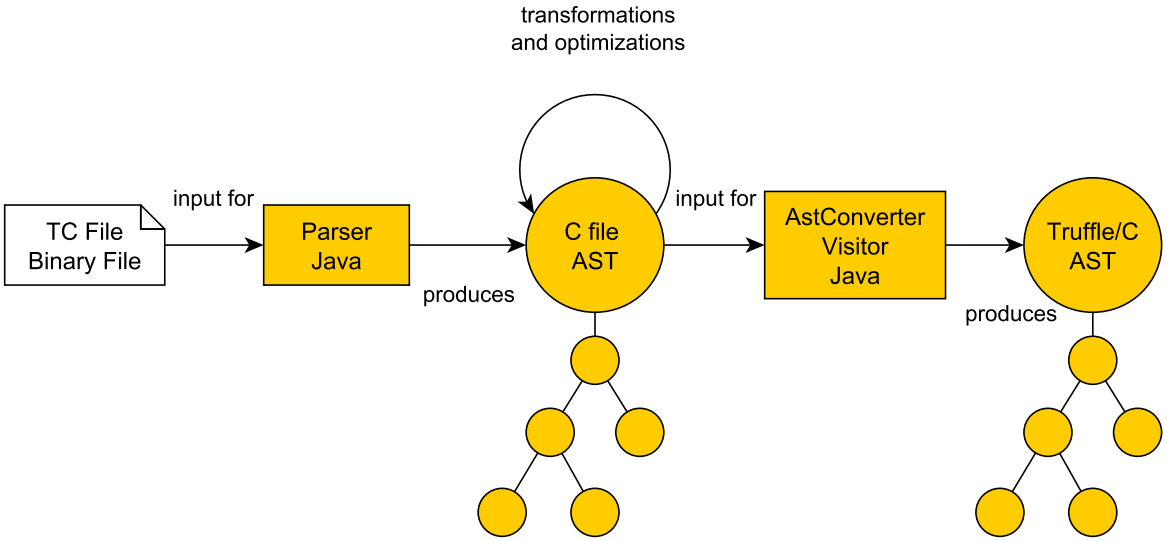

Figure 3.2: From the Truffle/C File to the Truffle/C AST

The previous Figure 3.1 suggests, that the parser at the Java part of Truffle/C directly produces the Truffle AST from the Truffle/C file. However, the implementation actually requires another temporary structure. As Figure 3.2 shows, the parser uses the Truffle/C file to produce an intermediate AST, that is conceptually equivalent to the AST present in the file. At the same time the parser builds up symbol tables, with information about types and data structures. Finally, the `AstConverterVisitor` traverses the intermediate AST to build the Truffle/C AST.

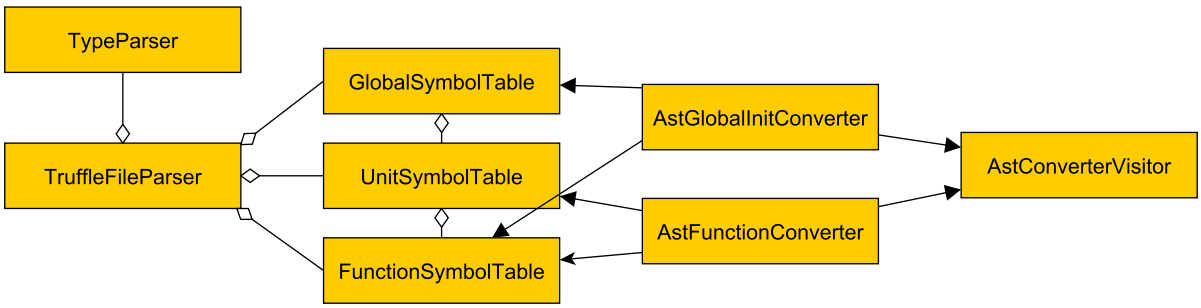

Figure 3.3: Important Classes for the Truffle/C Node Construction

Figure 3.3 shows the main classes, that the process of reading the Clang file requires. The `TruffleFileParser` parses the Truffle/C file and builds up the symbol tables. The `GlobalSymbolTable`

contains several instances of `UnitSymbolTable`. A unit refers to a single output file by the Clang modification, which can again consist of several C files. The `GlobalSymbolTable` itself helps to locate `extern` variables, which do not necessarily have to lie inside the same unit. The `UnitSymbolTable` contains the other "global" information such as global variable declarations, `extern` variable declarations, enumerations, records, and type definitions, as well as the necessary mapping of data to memory. The `UnitSymbolTable` also contains multiple instances of `FunctionSymbolTable`. A `FunctionSymbolTable` exists for every function and contains the static and non-static local variables, the compound literals, as well as other information that is local to the function. The Clang modification uses the custom Clang way to output types as strings. On the Java side, a `TypeParser` processes these type strings which are used for almost every node in the Truffle/C file.

3.2 Node Construction

Only after the parser constructed the intermediate format and the symbol tables, the `AstConverterVisitor` in Figure 3.3 can process the intermediate AST. The visitor processes the AST in a single pass and produces the nodes from the bottom to the top. Both the `AstGlobalInitConverter` and `AstFunctionConverter` use the symbol tables and the `AstConverterVisitor`, to build the Truffle nodes. The `AstGlobalInitConverter` builds those nodes, that Truffle/C needs to execute before calling the main function. The `AstFunctionConverter` constructs the nodes for the name of the function, that the linker passes. This split design allows lazy construction of functions: As long as the execution does not need a function, the `AstFunctionConverter` does not have to construct nodes for it. This design is also applicable for the construction of the intermediate format, since we designed the Truffle/C format to support lazy loading of functions (see Chapter 4). Since peak performance is the main goal and not start up time or memory efficiency, the `TruffleFileParser` does not yet support to lazily read the Truffle/C file.

3.3 Runtime

During run-time, Truffle/C informs the Truffle framework about execution statistics and branch probabilities (see Section 8.4). Truffle/C also guides and implements inlining. When a function gets hot, Truffle uses Graal to partial evaluate, optimize, and compile the AST.

Chapter 4

The Truffle/C File

This chapter explains the Truffle/C file format and what considerations determined its design. It also compares the Truffle/C file format to related formats such as ELF and Java class files.

4.1 Truffle/C File Format Goals

In order to comprehend the design of the file format, it is useful to first understand the requirements on this format:

- ① *Platform Independence*: Platform independence of the format and its content is the most important property. The Truffle/C file should be platform agnostic and not specify alignments or bit widths of data types. Instead, the interpreter should decide platform dependent details before execution.

- ② *High Level Information*: The file format is expected to contain all the high level information, that the C file contains. This high level information is needed as debug information, as well as to construct nodes directly corresponding to the high level statements in the code. Thus, it would for example not be preferable to lower a loop to a conditional backjump, as a conditional goto with a label might have the same lowered representation, thus shadowing the programmer's original intention.

- ③ *Lazy Parseable*: The file format should not restrict that functions can be parsed lazily. This also means that, e.g., global variables referenced by functions should be parseable without having

to construct the AST of other functions not yet needed.

- ④ *Extensible*: The format should be future proof and backward compatible. It should be possible to include optional information, that cannot be interpreted by all versions and is thus ignored.

- ⑤ *Debug Information*: It should be possible, to append optional debug information such as line numbers for statements from the source file.

- ⑥ *Compactness*: The format should be reasonable compact to ensure a fast start-up time.

- ⑦ *Comprehensibility*: The format should be easy comprehensible by an interpreter. The interpreter should not have to infer types or implicit conversions, to fully benefit from the Clang front end.

- ⑧ *Low Coupling*: There should be a high level of independence between the Clang and Java layer. The Clang layer should simplify the presentation and make operations explicit. The Java layer has to determine the platform dependent properties and then construct the fitting nodes for it.

While ④-⑦ are generic requirements, which apply for most file formats and architectures, ①-③ are uniquely connected to Truffle/C: Object files like ELF do not usually account for this requirements, which made the introduction of the Truffle/C format necessary. Some of these requirements also contradict each other, like ⑥ and ⑦. Providing more explicit information in ⑦ will increase the size of the format in ⑥. The design of the format tries to make a compromise between all these demands.

4.2 Truffle/C File Format 1

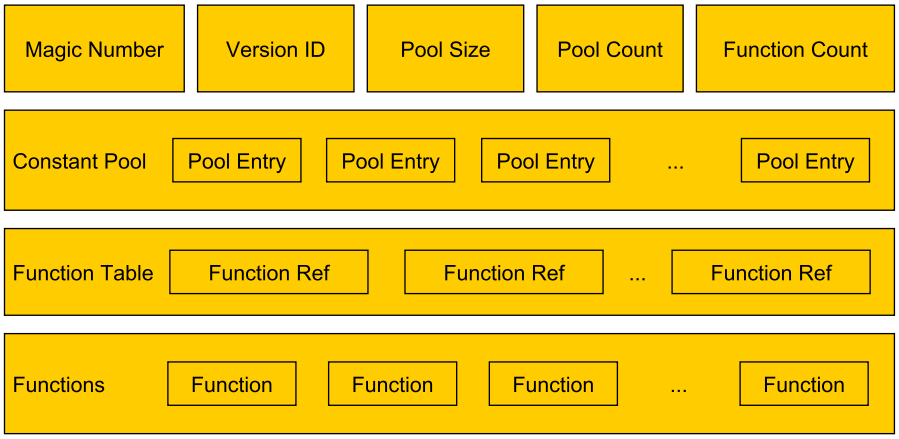

Figure 4.1: Truffle/C File Format 1

The file format is identified by the first four bytes of the file. This *Magic Number* as shown in Figure 4.1 should be equal to `0xDEADFACE`. Then a *Version ID* follows, which helps the versioning of Truffle/C.

4.2.1 Constant Pool

The next two items require prior explanation: Truffle/C uses a *Constant Pool* to store content like strings or numbers in it. The actual information like function data only references entries in the *Constant Pool*. If for example a string occurs several times in the original content, the *Constant Pool* only stores it one time and contributes to the compactness as stated in ⑥. The *Pool Size* stores the byte size of the Constant Pool, while the *Pool Count* stores the number of entries. Since a string or other entries can have a variable number of bytes, it is not possible to directly infer the *Pool Size* from the *Pool Count*. Taking into account consideration ③, the file format also includes *Pool Size* to be able to jump over that part in the file, while at the same time being able to allocate an array of *Pool Size* for the mapping of the *Constant Pool* indices to the content.

The *Constant Pool* supports at its current states the Java types **`byte`**, **`char`**, **`short`**, **`int`**, **`double`**, **`long`**, and `String`. These type-value pairs, as shown in Figure 4.2, are put into the *Constant Pool* as items of *Pool Entry*, with a *Type Tag* and *content[]*, which has a constant size for number types but a variable size for `String`.

4.2.2 Function Table

The file contains a field *Function Count* for the number of C functions in the file, as seen in Figure 4.1. Then, the functions are not sequentially aligned but a *Function Table* follows. This again helps to jump over the file content to support consideration ③, without having to process the function data to determine the end of a function. The *Function Table* consists of multiple items of *Function Ref*. To fully identify the function, the *Function Name*, containing the name of the function as a string, is sufficient, since C does not support function overloading. To be able to non-sequentially read a function from the file, the *Function Ref* contains the *Offset* for the offset of the function data starting from the *Function Table* and the *Data Length* for the function data size in byte.

4.2.3 Functions and Attributes

Eventually, the function information is present in *Functions* which consists of multiple *Function*. These functions, which correspond to actual C functions, are shown in Figure 4.2. Each *Function* is represented as an AST and consists of multiple *Function Data*. Each *Function Data* corresponds to one node in the AST and has a *Node Name*, that denotes the name of the node. Then a *data[]* item with a fixed length which is dependent on the *Node Name* follows. This *data[]* could for example be an integer value for an integer literal or for a statement block the number of children. Since not all possible instances for *Node Name* are implemented yet, a full list is not given here.

At the end of each *Node Data*, there is an optional sequence of *Attribute*, accounting for extensibility and thus consideration ④. An *Attribute Tag* identifies what kind of attribute is attached. Then, the *Attribute Size* denotes the length in byte of the content in *data[]*. The important characteristic while handling an *Attribute* is, that an interpreter has to ignore unknown attributes and use the *Attribute Size* item to jump over the unknown content. Thus, later implementations of Truffle/C can for example attach line numbers as attributes to provide debug information such as stated in ⑤.

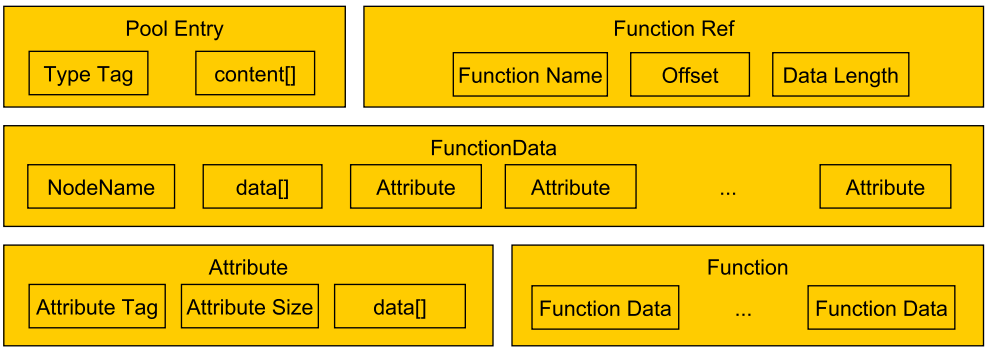

Figure 4.2: Truffle/C File Format 2

4.3 Truffle/C File Considerations and Comparison

4.3.1 Java Class File and Truffle/C File

The Truffle/C file format bears a strong resemblance to the Java class file format. The first Truffle/C prototype even directly generated Java class files, since a prototype Truffle/Java implementation already provided the necessary infrastructure for reading such a class file with an attached Truffle AST. This approach enabled a compact implementation, since there was no redundancy like there is in the implementation now, where essentially a simplified representation of the internal Clang AST is rebuilt on the Java side. However, there was also a strong coupling between the two layers, since the Clang modification already had to know the name and structure of the classes. Due the high maintenance effort, we decided for the present solution with an independent Truffle/C format, which also fits to consideration ⑧.

The constant pool [29, JVM 4.4] is essentially a residual from this first implementation. It is now also used for the node data in the *Function* items. The Truffle/C format also incorporates the Java attributes [29, JVM 4.7] to guarantee extensibility. Also Java specifies, that for unknown attributes, the parser has to "silently ignore those attributes" [29, JVM 4.7]. Such as the initial implementation could use a custom attribute to attach the node AST in the Java class file format, future extension can make use of this attribute architecture in Truffle/C.

Truffle/C uses an indirection for the functions in form of a *Function Table* and *Functions* to support lazy construction of nodes for functions as in consideration ③. The Java class file format, however, was designed to be processed at once, because "the Java Virtual Machine needs to verify for itself that the desired constraints are satisfied by the class files it attempts to incorporate. A Java Virtual Machine

implementation verifies that each class file satisfies the necessary constraints at linking time" [29, JVM 4.10]. The Java class file thus lacks a lookup similar to the *Function Table* and has its method aligned sequentially in a *method_info* structure, with the code attached as an attribute [29, JVM 4.7.3]. Truffle/C specifies no such check at the moment, and thus fails at execution time, when a function contains wrong content.

Regarding other aspects, the Truffle/C also follows a similar design philosophy as the Java class file format: Such as the bytecodes, the *Function Data* items are platform independent and contain no platform dependent nodes as expressed in consideration ①. Its design also incorporates that the names of identifiers are included per default, so they can be used as debug information as in consideration ② and ⑤.

4.3.2 ELF and Truffle/C File

On Linux, the ELF format is the prevalent choice for generating binary files for C programs and is also the output format of GCC. Most distinctively, the format does not comply with consideration ①, the platform independence. ELF uses flags to indicates whether the code uses a 32 or 64 bit format, if the data had been saved in little or big endianness, and targets a certain ABI and instruction set architecture [1]. The code in an ELF file is (after linking) already machine code and contains all the alignments. In contrast, Truffle/C computes these alignments before the execution of the function, and not already after compiling.

Generally, the code in the ELF file specifies exactly what to execute. The content in the Truffle/C format in the functions is rather a description of the statements contained in the file. For example, when an ELF file contains code to initialize a struct, the Truffle/C file highlights information of the struct fields, including the value to which they have to be initialized.

One feature of reading ELF files similar to Truffle/C is when using shared libraries, where the Procedure Linkage Table (PLT) adds an indirection for function calls. The first time a function is called, the linker is called instead to look up and bind the function [28]. This lazy mechanism is comparable to the Truffle/C format, where function calls denote the function as a `String`, that essentially point to the *Function Table*.

4.4 Clang Modification Truffle/C File

One of the additional challenges when implementing the Clang modification, was the complex API of the internal AST: Clang is not only designed for C, but also for other languages of the C family C++, Objective C, and Objective C++. Thus, some of the API functionality is not applicable for C, and calling the "wrong methods" can result in wrong results, crashes, or unexpected behavior.

The Truffle/C Clang modification uses the internal tree of Clang which is subject to modification and not clearly documented. In particular, it is hard to see, what combinations of nodes can occur in C, and which cannot. The Clang modification is essentially a modification of a debug output tool (class `ASTDumper`), which is sometimes incomplete or does not output the desired result from the perspective of Truffle/C. One main effort is to keep the changes as minimal as possible, since it seems likely, that the output mechanism has to be updated with major Clang versions, as we already had to do once.

```
int sum(int arr[], int n) {
  int sum = 0;
  int i;
  for (i = 0; i < n; i++) {
    sum += arr[i];
  }
  return sum;
}
```

Listing 4.1: Simple C Function with Corresponding Truffle/C File in Listing 4.2

```
FunctionDecl 93105984 sum (E) int (int *, int) 2  impl
  ParmVarDecl 93105664 arr int * nonstatic nonextern 0
  ParmVarDecl 93105776 n int nonstatic nonextern 0
  CompoundStmt 4
    DeclStmt 1
      (A) VarDecl 93106176 sum int nonstatic nonextern 1
        (C) IntegerLiteral int 0
    DeclStmt 1
      (A) VarDecl 93106336 i int nonstatic nonextern 0
    ForStmt 4
      BinaryOperator int '=' 2
        DeclRefExpr Var 93106336 i 0
        (C) IntegerLiteral int 0
      BinaryOperator int '<' 2
        (D) ImplicitCastExpr int 1
          (B) DeclRefExpr Var 93106336 i 0
        (D) ImplicitCastExpr int 1
          (B) DeclRefExpr ParmVar 93105776 n 0
      UnaryOperator int postfix ++ 1
        (B) DeclRefExpr Var 93106336 i 0
```

```
    CompoundStmt 1
      CompoundAssignOperator int += 2
        (B) DeclRefExpr Var 93106176 sum 0
        (D) ImplicitCastExpr int 1
          ArraySubscriptExpr int 2
            (D) ImplicitCastExpr int * 1
              DeclRefExpr ParmVar 93105664 arr 0
            (D) ImplicitCastExpr int 1
              DeclRefExpr Var 93106336 i 0
ReturnStmt 1
  (D) ImplicitCastExpr int 1
    DeclRefExpr Var 93106176 sum 0
```

Listing 4.2: Truffle/C File of Listing 4.1

For the C code in Listing 4.1, the Clang modification produces a binary file with a structure equivalent to Listing 4.2. As I explained before, the file uses a very expressive format and the strings in this listing are actually present in the binary file. All the variable declarations include an ID as shown in (A). If a node later on references this variable such as in (B), the ID helps to identify the variable again. All nodes have as list element an `int` value, that tells the number of children. This is often redundant, such as for the `int` literals in (C), which cannot have a child. The Clang modification helps to identify all implicit conversions such as in (D), which Truffle/C uses to generate Casts, as Chapter 5 explains. The notation that the `TypeParser` has to parse is, e.g., visible in the function signature of (E). The function with signature `int (int *, int)` expects an `int*` and an `int` also returns an `int`. Section 6.3.2 explains, why the first parameter does not have type `int[]`.

Chapter 5

Truffle/C Data Types

In the implementation of Truffle/C, the mapping between the data types of C and Java constitute the basic Truffle/C node classes. This chapter introduces two approaches for such a mapping.

An important architectural part in Truffle/C is the mapping of data types from C to Java. The C standard offers a distinction between the rank of a data type, i.e., the type specified on the language level and the representation of this data type in the implementation [23, C99 6.3.3.1]. When implementing a C interpreter in Java, it is essential to consider how to map the rank of the language data type to the Java (primitive) type and how to perform conversions between them.

In terms of conversion, the implementation specifically has to support both implicit casts and explicit casts. Listing 5.2 shows an explicit cast. An explicit cast is directly specified in the program code. Listing 5.1 shows an implicit cast. An implicit cast is not directly specified in the program code, but by the implicit type conversion rules of the programming language.

C, as well as Java, only defines implicit upcasts and no implicit downcasts [23, C99 6.3.1.8.1] [19, JLS 5]. However, in C, many compilers perform type conversions implicitly for mismatching types, as long as a common type can be found. This implies, that Truffle/C also needs to support implicit downcasts. Figure 5.1 shows the conceptual AST of Listing 5.1 where C compilers have to perform both an implicit conversion to a higher and a lower rank: Since the plus operation requires at least an `int` rank, the specification requires the conversion of the `short` and `char` variables to a higher-ranked `int`. Common C compiler then assign the `int` result of the addition to a lower-ranked `char`. The data type is truncated, thus transgressing the C specification, which is what many compilers support.

```
short var_a;
char var_b;
char res = var_a + var_b;
```

Listing 5.1: Implicit Upcasts and Downcast

```
int var = (int) 1.3f;
```

Listing 5.2: Explicit Downcast

Since Clang's AST already contains nodes for implicit casts, the parser does not need to find out, where implicit casts occur. Truffle/C hence can treat implicit and explicit casts in the same way.

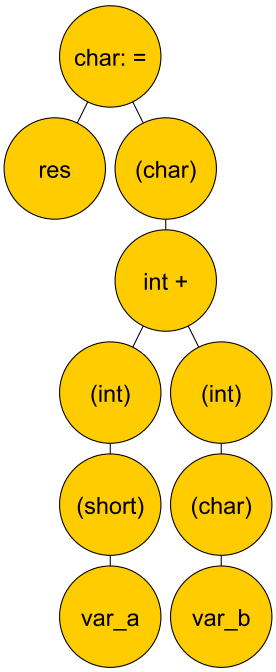

Figure 5.1: Clang AST of Listing 5.1 with Explicit Nodes for the Casts

The implementation should support all primitive and non-primitive types. A challenge regarding primitive types is, that C supports signed as well as unsigned types. However, Java only supports signed types. Truffle/C needs to perform unsigned operations and should exhibit the correct sign extension behavior. A problem with non-primitive types such as arrays, union, structs and others is, that these types can be reinterpreted as other non-primitive types through casts, since they are essentially just memory regions. Truffle/C also needs to support these arbitrary casts between complex types.

5.1 Data Type Hierarchy: Boxing, Upcasts and Downcasts

One of the goals when implementing nodes is that they are compact and only contain the code for the specific case of execution. As a concrete example, let us consider the plus operation: Plus is defined for many data types, so a node could contain methods for all the possible data types. But this would make the compilation slower and would produce suboptimal code. To be as compact as possible, we split these methods in separate nodes, so that for every data type a special version of plus exists. A generic execute method that returns an `Object` is also not applicable, since we want to avoid boxing in our interpreter, as Section 2.2 explains.

The following section presents two different approaches on how to implement a data type hierarchy, which avoids boxing and is as minimal as possible regarding the cases, which it has to handle. It also explains, how Truffle/C implements conversion between the types. Listing 5.1 serves as an example to explain the architecture.

5.1.1 The Implicit Approach

One approach is to implement a hierarchical type system. This approach can use the implicit widening primitive mechanisms [19, JLS 5.1] in Java. An example for this widening conversion is, that Java allows a substitution of an `int` in a context where a `long` is expected. Since the numeric value is preserved in an upcast (this does not necessarily hold from `float` to `double` [19, JLS 5.1.2]), Java performs an implicit conversion. We can facilitate this by arranging the node hierarchy according to the implicit widening conversion in Java.

This approach is shown in Figure 5.2. The left side shows an excerpt of the Java class hierarchy, and the right side the nodes Truffle/C would construct for the example in Listing 5.1.

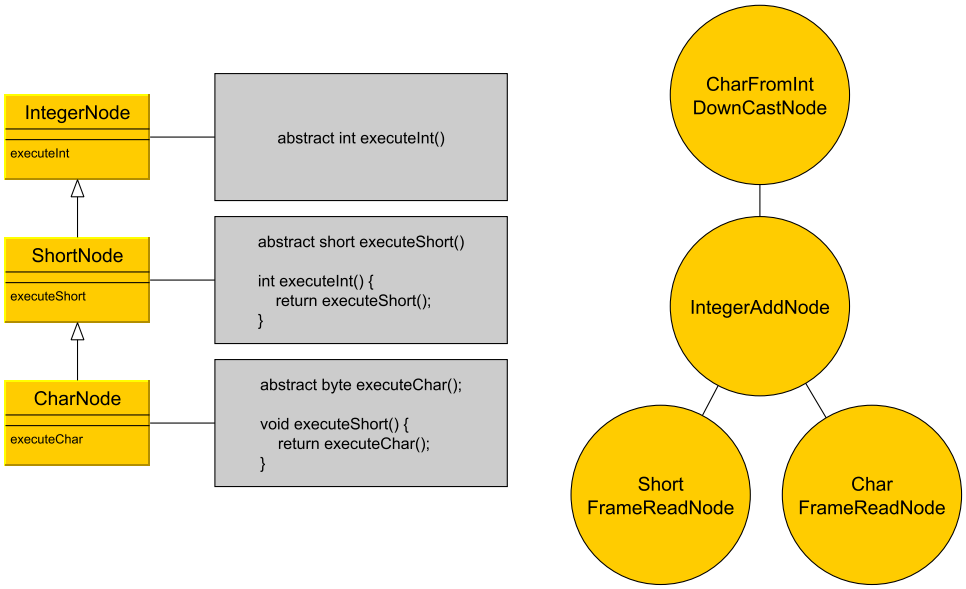

Figure 5.2: Truffle/C Nodes from the Implicit Approach for Listing 5.1

Truffle/C can treat the `CharNode` as a `ShortNode`, since the Java widening conversions can implicitly convert a **`byte`** (a **`byte`** in Java has eight bit and hence better represents the C `char` than the Java **`char`** with sixteen bit) to **`short`**. The same relationship holds between the `ShortNode` and `IntegerNode`. Since the relationship is transitive, the **`byte`** can be upcasted to **`int`**.

When the **`int`** addition expects its left and right side to be of the result type **`int`**, it can just call the `executeInt` of its children. For the left child, the `executeInt` will propagate to `executeShort`. For the right child, the `executeInt` will propagate to `executeShort` and then to `executeChar`.

However, this approach still needs an explicit node, which handles downcasts. Thus, the `CharFromIntDownCastNode` converts the **`int`** Java type to a **`byte`** Java type during execution by truncation.

This approach has the advantage that it facilitates a subset of implicit conversions both common to Java and C. This allows to omit the nodes in the AST for arithmetic upcasts, keeping the AST compact. However, this approach makes it hard to find errors: When debugging, the programmer has to keep in mind the implicit conversions performed during execution. Also, the debugging depth in terms of methods is higher, when converting from a low ranked data type to a high ranked data type. While the conversion from `ShortNode` to `IntegerNode` requires only one method to be called, the conversion from `CharNode` to `IntegerNode` requires two method calls. However, since of the transitivity of upcast conversions, we could reimplement all execution methods of the superclasses in the lower ones to avoid this problem. Another problem is, that the implementation is more error prone, than

the second approach in the next section. When constructing nodes in the parser, Java allows any subclass like `CharNode` or `ShortNode`, where an `IntegerNode` is expected. In the initial implementation it happened, that Truffle/C did not create necessary cast nodes, but the node construction did not fail since inheritance allowed the substitution with a subtype. Because of the disadvantages described here, Truffle/C uses the "Explicit Approach" instead of the "Implicit Approach" described here.

5.1.2 The Explicit Approach

Figure 5.3 shows the Java classes for Listing 5.1 on the left side, the nodes Truffle/C constructs on the right side.

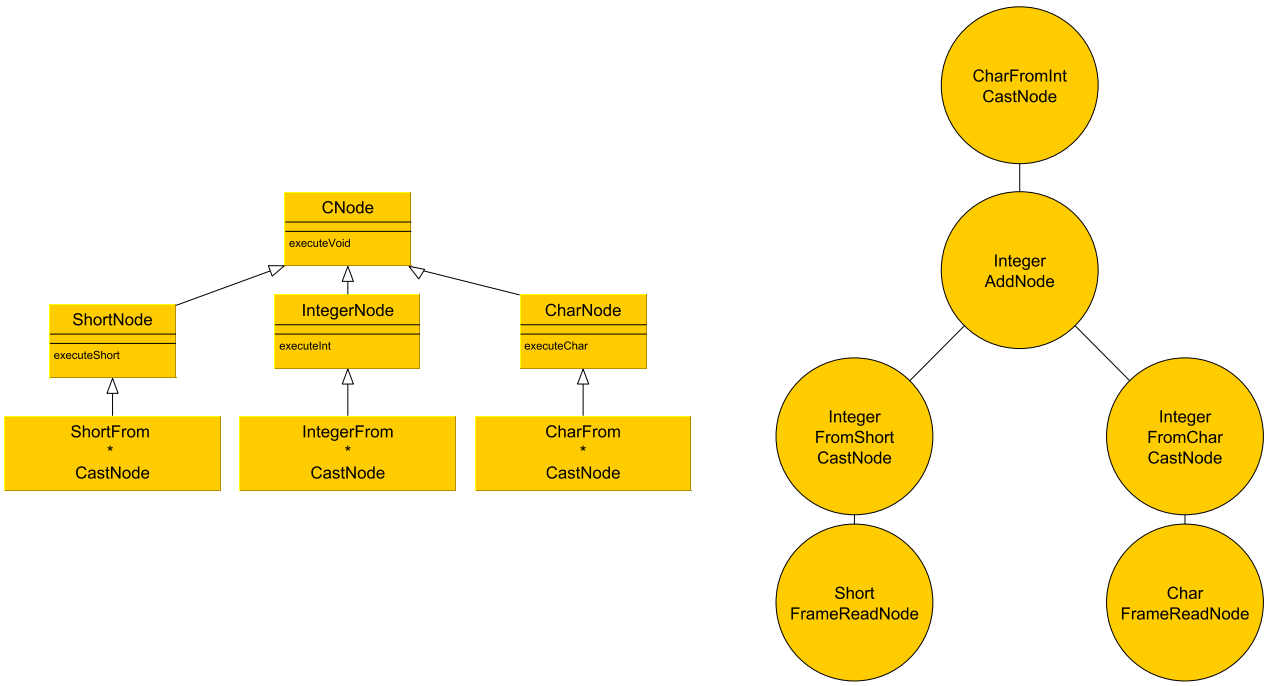

Figure 5.3: Truffle/C Nodes from the Explicit Approach for Listing 5.1

The "Explicit Approach" has nodes for each conversion. Whenever an explicit or implicit cast to another type should be performed, Truffle/C inserts a cast node into the AST, that performs the conversion. For upcasts, the cast nodes can return the value directly, facilitating again the implicit upcast semantics of Java. The downcasts are implemented in the same way as in the "Implicit Approach".

The advantage of this approach is, that each conversion is expressed explicitly in the Truffle/C AST. This improves the debuggability but makes the AST less compact. A major advantage is, that it is

easier to find errors during node construction, since a wrong node type results in a `ClassCastException`, where before, the execution silently failed with a wrong result. In the following, the thesis assumes the implementation of the "Explicit Approach".

5.2 Signed and Unsigned

Besides signed data types, C also supports unsigned integer data types. Unsigned data types have the same amount of storage and the same alignment requirements [23, C99 6.2.5.6]. However, the interpretation of the bit values in this storage with bit width N differs: C interprets unsigned values reaching from 0 to $2^N - 1$. Signed values divide the space into negative and positive numbers from $-1 * 2^{N-1}$ to $2^{N-1} - 1$. Arithmetic operations on unsigned types differ from operations on signed types. Also, the sign extension of unsigned types differs from signed semantics.

Truffle/C handles unsigned types by differentiating the operations between signed and unsigned types, thus constructing different operation nodes. Both unsigned and signed integer type operations of the same type inherit from a common node. A common node implies, that Truffle/C uses the same Java type to represent the unsigned and signed values. However, Truffle/C constructs different arithmetic nodes, depending on whether the C type is signed or unsigned.

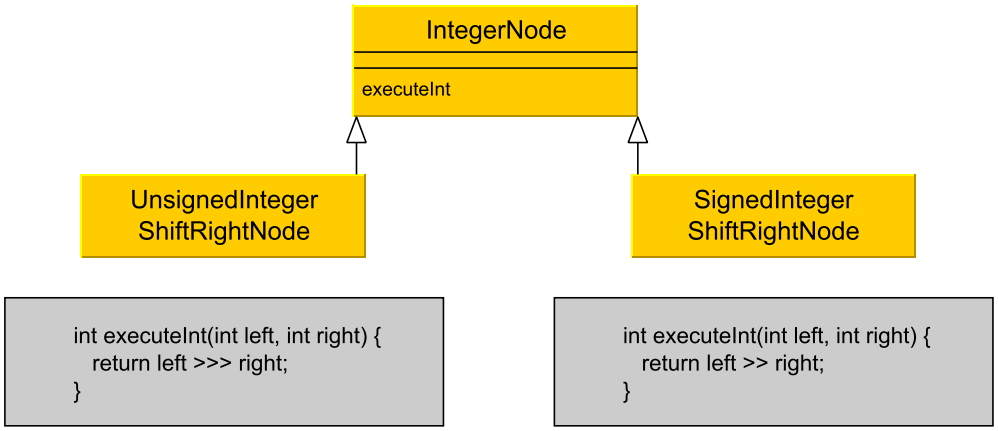

Figure 5.4: Signed and Unsigned Integer Right Shift Class Diagram

Figure 5.4 shows this concept on the `int` right shift operator: Both `UnsignedIntegerShiftRightNode` and `SignedIntegerShiftRightNode` inherit from `IntegerNode`. The `executeInt` method returns a Java `int` for both nodes. However, the implementation of the method differs, to produce the right result for the unsigned, respective the signed type.

Type	s to s	s to u	u to u	u to s
upcast	✓ ①	✓ ①	✗ ②	✗ ②
same type ③	✓	✓	✓	✓
downcast ④	✓	✓	✓	✓

Table 5.1: Case Distinction for Conversion between Types

A special case is the conversion between signed and unsigned. Table 5.1 shows the possible cases. The conversion can either be an upcast, downcast or a conversion within the same type. The table also specifies whether the conversion is from signed to signed, unsigned to unsigned, signed to unsigned or unsigned to signed. Truffle/C does not have to handle case ① specially, since Java automatically sign extends for upcasts. Listing 5.3 illustrates case ①, where C has to perform a sign extension, when upcasting from a signed variable.

```
short var_a = -1;
unsigned int var_b = var_a;
printf("%d", var_b); // -1
```

Listing 5.3: Upcast from Signed with Sign Extension

```
unsigned short var_a = -1;
unsigned int var_b = var_a;
printf("%d", var_b); // 65535
```

Listing 5.4: Upcast from Unsigned without Sign Extension

However, for case ②, Java also sign extends, which is the incorrect semantics for upcasts from unsigned. Listing 5.4 shows, that C performs no sign extension, when upcasting from an unsigned variable. However, when assigning a **`short`** to an **`int`** in Java, the **`short`** is sign extended, which results in a wrong result for a negative value.

Case ③ does not require special treatment, since a conversion between the same type is just a reinterpretation of the bit content. Also, Java already handles case ④ in the same way as C does, because they both truncate the value on a downcast.

To correctly implement an upcast from unsigned in Java, we need to truncate the sign extension that Java implicitly performs. For converting a value from a source type with bit width N to the target type, we mask it with $2^N - 1$. This means, that the target type value is the same, as if we copied the bits from the source type to the target type, without sign extending them.

Figure 5.5 shows the truncation as nodes for the example in Listing 5.4: `ShortFrameReadNode` reads the value of the variable as a **`short`**. Then, the `IntegerFromShortCastNode` uses the implicit Java arithmetic widening conversion to automatically upcast it to an **`int`**. However, because sign extension can occur, the parent node masks an eventual sign extension.

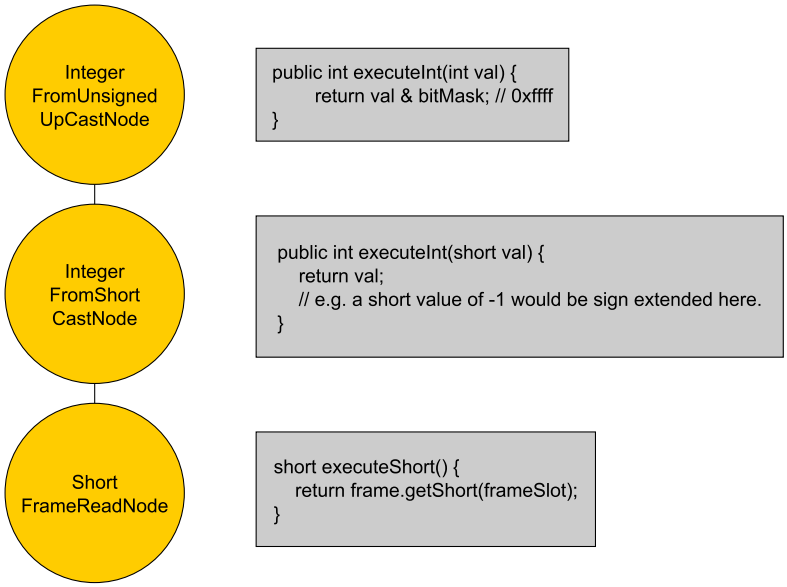

Figure 5.5: Truffle/C Nodes for the Truncation for Unsigned Upcasts of Listing 5.4

5.3 C Data Types and Truffle/C Base Nodes

For mapping the data types from Java to C, the bit width should be the same as on the platform, to support interoperability between native functions and Truffle/C. Table 5.2 shows the mapping between the standard data types and the assigned nodes with return value type of their execute method in Java.

All these nodes except `ConditionNode` inherit from `RawDataNode`, whose `executeRaw` method returns a **`long`** value. Since it is possible to represent all the values within the bits of a **`long`**, having a method like this is useful for passing arguments to functions. When a function is called, this method can execute each node argument and the calling node can pass the arguments to the callee as a Java **`long`** value.

5.3.1 Primitive Types

The base nodes include the representations of C primitives which are `CharNode`, `ShortNode`, `LongNode`, `FloatNode`, and `DoubleNode`, such as shown in Figure 5.6. They simply map to the corresponding C type, and use the same bit width in their `execute` method as the corresponding C type.

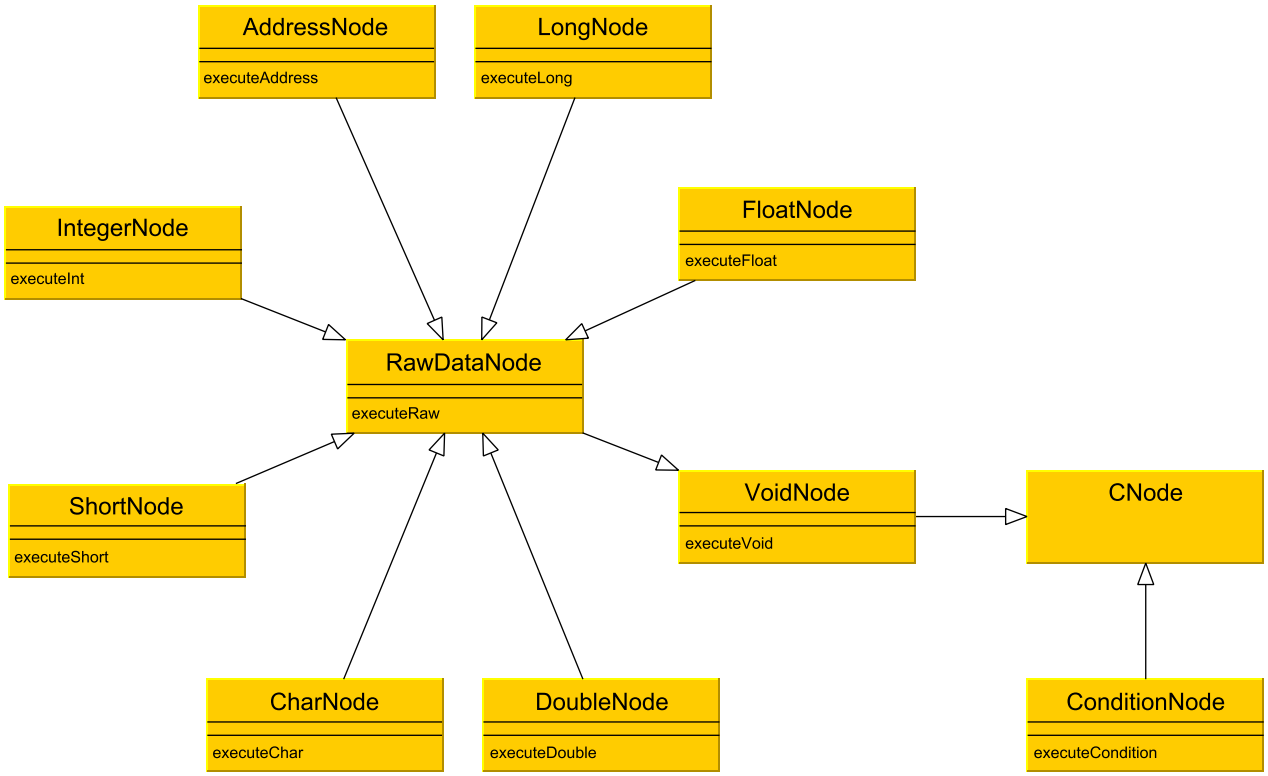

Figure 5.6: Truffle/C Base Nodes Class Diagram

5.3.2 Condition Type

Truffle/C maps conditions of loops or other conditional control flow statements to a `ConditionNode`. In C, a `bool` data type exists since C99 in `stdbool.h` [23, C99 7.16]. However, since the condition can be any data type, e.g., also a `long`, Truffle/C uses an artificial data type to represent a condition in control statements. For every C data type, a subclass of `ConditionNode` converts a value of the given data type into a **`boolean`** value. The artificial data type prevents a combinatorial explosion of nodes, where every control statement (and every specialization of it) would require a node for every possible data type of the condition.

5.3.3 Pointer Type

Besides the common primitive types, there is also an `AddressNode` as shown in Figure 5.6. The `LongNode` as well as the `AddressNode` both return a Java **`long`** value in their `execute` method. Truffle/C uses `LongNode` to represent C **`long`** variables. In contrast, it uses `AddressNode` in a context, where an

C Type	Canonical Type	GCC AMD64 Bitsize	Truffle/C Node	Java primitive
`char`	signed char	8	`CharNode`	**`byte`**
`signed char`				
`unsigned char`	unsigned char			
`short`	signed short	16	`ShortNode`	**`short`**
`short int`				
`signed short`				
`signed short int`				
`unsigned short`	unsigned short			
`unsigned short int`				
`int`	signed int	32	`IntegerNode`	**`int`**
`signed int`				
`unsigned`	unsigned int			
`unsigned int`				
`long`	signed long	64	`LongNode`	**`long`**
`long int`				
`signed long`				
`signed long int`				
`unsigned long`	unsigned long			
`unsigned long int`				
`long long`	signed long			
`long long int`				
`signed long long`				
`signed long long int`				
`unsigned long long`	unsigned long			
`unsigned long long int`				
`float`	float	32	`FloatNode`	**`float`**
`double`	double	64	`DoubleNode`	**`double`**
`long double`	long double	128		

Table 5.2: Mapping of C Data Types to Platform Dependent GCC Bit Widths and Truffle/C Nodes

address is expected. Examples for such contexts are pointer arithmetic or pointer variables. The pointer addresses are not virtual, since the `Memory` implementation returns the "real" addresses of the data objects.

5.3.4 Other Data Types

All data types exhibit the same semantics as the GCC AMD 64 implementation, except for `long double`. Representing `long double` as a `double` with half the bit width does not contradict the C standard. However, interoperability with native libraries is not given regarding this data type.

Apart from the data types shown in the table, Truffle/C also partially supports specific type extensions in header files of the platform. This includes for example the `long128` data type [5, GCC 6.8], which is preliminarily only supported by a 64 bit **`long`** instead of a 128 bit type. Additionally, the C99 standard introduced a `bool` data type which Truffle/C treats as a type definition for `int`. Besides the arithmetic primitives, Truffle/C also supports pointers as described before and the "`void` data type" (it is not actually a data type [23, C99 6.3.2.2, 6.2.5.19]).

5.3.5 Type Definitions

Truffle/C supports type definitions, by explicitly outputting them in the Clang AST. During construction of the nodes, Truffle/C resolves references to types, that refer to a type definition. The advantage of not directly outputting the original type already in the Clang AST is, that Truffle/C can print informative error messages.

Chapter 6

Data and Data Structures

Basing on the previous chapter, this chapter explains the different kind of variable types, and how Truffle/C handles compound objects. Also, it shows how Truffle/C handles strings and literals.

Truffle/C supports most of the data structures of C, such as different kinds of literals, compound objects such as structures, unions, and arrays, as well as enumerations. Truffle/C has a relatively complex memory management to allow for an efficient handling of addresses and pointers, as well as easy interoperability to native functions.

6.1 Frame and Memory

The assigned storage of a variable can either be in the `Frame` or in the `Memory`.

The frame concept is part of the Truffle API and ensures a fast access to variables in compiled code. It contains the type specialized variable values of local variables. In interpreted mode, the variable values lie in an array, whereas this array does not exist in compiled code. Graal eliminates the array by performing an Escape Analysis after partial evaluation and connecting each read on the array with the last write [38, 42].

The memory concept is part of Truffle/C and is described in more detail by my colleague [21]. In the memory concept, local or global variables are stored in the native heap. Because of the optimized frame accesses, as many variables as possible should be in the `Frame`. However, there are two exceptions where this is not possible or desirable:

- An address-of operator `&` is used to obtain the address of a variable.

- The variable is a composite and a caller passes it to a native function.

In the first case, Truffle/C has no other choice than to put the variable into the `Memory`. Truffle/C cannot put it into the `Frame`, because it needs a memory address to refer to it. The programmer might use its address to perform pointer arithmetic or pass it to a native library.

Regarding the second case, Truffle/C always places composites such as structures, arrays and unions into the `Memory`. This allows a simple implementation and enables interoperability between Truffle/C and native libraries via the native function interface [22]. If Truffle/C would not place composites into the `Memory`, nodes would have to convert them whenever calling a native function.

6.2 Variables

Truffle/C represents variables by a location identifier, that other nodes can use to read the value from the `Memory` or `Frame`. Thus, different read and write nodes are needed depending on whether the location identifier is an address or a `FrameSlot`. A location identifier can be produced by one of the three following nodes:

- `GlobalMemoryAddressNode`: Truffle/C uses this node to represent globally allocated storage, where the interpreter allocates the `Memory` already during node construction. The address is hence constant during the run-time of the program.

- `LocalMemoryAddressNode`: This kind of node is used for compounds and for non-static function variables, when an address-of operator occurs inside the function and targets the variable. The `Memory` allocates the variable once per function call. Instead of an absolute address, it contains a relative offset from which it computes the absolute address during run-time.

- `FrameSlotNode`: This node returns the same `FrameSlot`. A `FrameSlot` is an identifier for the `Frame` provided by the Truffle API. Every function call allocates a new `Frame`.

For global and static variables, Truffle/C always uses the `GlobalMemoryAddressNode`. When the linker loads a function it executes its static initialization block. A global block executes its initializations once before executing the main function.

6.3 Compound Objects

6.3.1 Structures and Unions

Structures are objects with members that are aligned in a sequence, while the members of an union overlap [23, C99 6.7.2.1]. While Truffle/C needs to align the members of a structure according to the ABI, it can assign the union members to an offset of zero (but potentially has to align members of its members).

There are several ways to initialize structures and unions as shown by Listing 6.1. For the initialization of `t1` and `t2`, Truffle/C takes each subobject of the initialization list to assign it consecutively to the members of the structure. As a structure can again contain compound objects like the array `c`, Truffle/C has to perform assignments recursively while following the different alignment requirements [23, C99 6.7.8.(17,20)]. The listing shows that for `t2` and `t4` only some members have assigned initializations. The standard specifies implicit initializations which are zeros for these members [23, 6.7.8.(19,20)]. This also applies for the `t4` initialization, where the identifiers explicitly denote their member targets. The initialization value can also come from an arbitrary expression, such as from a function call, a compound literal, a primitive value, or another structure. In the case of `t3`, the initializer copies the `Memory` content with the size of the structure from the expression. In other cases, such as for a compound literal, it is sufficient to copy the pointer.

```
struct data {
  long a;
  int b;
  char c[10];
  int d;
};

struct data2 {
  int a;
} t1 = {1};

int main() {
  struct data t2 = {1, 2, {1, 2, 3}};
  struct data t3 = t1;
  struct data t4 = {.b = 5, .c = {4, 5, 6}};
}
```

Listing 6.1: Different Kinds of Initializations for Structures

For passing structures by value, Truffle/C recursively extracts the members of a structure (e.g., also all members of an array) and passes them via the `Frame` to the receiving function. In the function,

initialization nodes copy the values from the `Frame` to the `Memory`. On first glance this seems to be a high overhead, but after compilation putting a value into the `Frame` is "free" and the compiled code copies the values without writing them to the `Frame`. Truffle/C must copy the values in each case, since writes on one structure should not change the values in the other. This implementation eases the handling of the native side, since the ABI also requires that the caller passes the members of the structure sequentially.

For returning structures per value, the caller allocates storage to hold the structure and then passes the pointer to the structure callee. When the callee wants to return a structure, a node inside the callee copies the structure to the address the caller previously passed. This also conforms to the concept that the ABI specifies.

Bit fields are a feature that requires more implementation effort. Bit fields allow to read and store numerical values in a more fine grained and compact manner. In practice, a programmer can use all the numerical data types (the standard just defines three [23, C99 6.7.2.1.4]) in signed and unsigned variants to qualify them as a bit-field. The compiler is then expected to fill the according data type with the consecutive bit fields up until the next value does not fit any more into the variable of the specific data type. Most other alignment decisions are implementation dependent [23, 6.7.2.1.(4-13)] and Truffle/C implements them in the same manner as GCC. Bit fields can be mixed with non bit field members, and together with the ABI for non bit field structure member, compiler implementers have to take special care regarding the alignments.

When reading or writing a bit field, the Truffle/C nodes require a bit offset and bit length. Bit field nodes also require the address of the structure member, which can be the same for multiple bit fields. The nodes use the bit offset and bit length to form a bit mask to read or write the individual bits. When reading from a bit field Truffle/C also constructs special casts. If the bit field is signed, the leading bit has to be used for sign extension.

Listing 6.2 shows an example use of bit fields. The variables from `a` to `d` have the bit widths of five, one, three, and seven respectively. The unnamed field with a bit width of zero indicates, that for a possibly following bit field, the compiler should start the allocation in a new memory area according to the data type that the programmer specified [23, C99 6.7.2.1.11]. Thus, the first variable `a` resides alone in the first `char Memory` area. The variable `b` together with `c` consecutively occupy the bits in the second `char`. Since the seven bits do not have enough space in the second `char`, Truffle/C places them in the third `char`. Although the initialization assigns the values thirty, one, eight, and three, the output values differ from the ones the programmer assigned to them. Table 6.1 helps to explain this behavior: The two's complement bit representation of `30` with five available bits is the negative value `-2`. When Truffle/C reads the value, it sign extends the `int` with the leading bit, which is `1`. Since variable `b` only has one bit to store values, its value range reaches from `-1` to `0`. Thus, also here the

0	1	2	3	4	5	6	7
a							
1	1	1	1	0	✗	✗	✗
b		c					
1	0	0	0	✗	✗	✗	✗
d							
0	0	0	0	0	1	1	✗

Table 6.1: Memory Content Table for the Bit Field Structure in Listing 6.2

value of `1` overflows to `-1` and Truffle/C sign extends the number to an `int` of value `-1`, as mentioned in the previous paragraph. For `c`, Truffle/C writes `0`. The value `8` occupies four bits and Truffle/C has to perform bit masking to ensure that it does not overwrite neighboring bits. Only variable `d` does not overflow its value range and hence stores and outputs `3`.

```
struct data {
  char a : 5;
  char : 0;
  char b : 1;
  char c : 3;
  char d : 7;
};

int main() {
  struct data t1 = {30, 1, 8, 3};
  printf("%d %d %d %d", t1.a, t1.b, t1.c, t1.d); // -2 -1 0 3
}
```

Listing 6.2: Structure with Bit Field Members

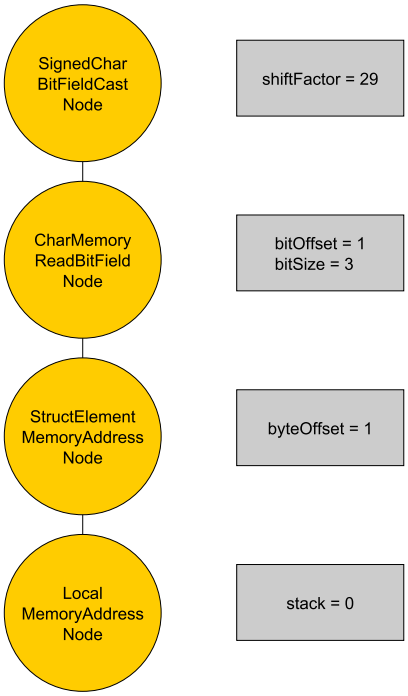

Figure 6.1: Truffle/C Nodes for a Read on Bit Field `c` of Listing 6.2

Figure 6.1 shows the nodes Truffle/C constructs when producing a read on the bit field `c` of the previous example. A `LocalMemoryAddressNode` points to the beginning of the structure. From there, we have to select the correct byte which is the second one and thus has an offset of one. Next, the correct bits are selected by a bit offset of one and a bit length of three. Truffle/C reads the whole byte, and then uses a bit mask to extract the right bits. The last step that Truffle/C has to do is to perform the sign extension, which first uses a left shift and then a right shift of a factor of twenty-nine in the previous example.

6.3.2 Arrays

Truffle/C performs the initialization of arrays by providing a node to initialize a certain `Memory` area. It executes the children and then consecutively stores the results in the memory area. Truffle/C supports both one and multi-dimensional arrays, as well as variable length and known constant size arrays. For known constant size arrays the size is a constant integer expression, while for variable length arrays it is not. Listing 6.3 shows the two kinds of arrays: `arr1` receives its size argument per function parameter and is thus a variable length array, and `arr2` has a constant integer expression and is thus a constant size array. Truffle/C notes for constant size arrays such as `arr2` that the function has to allocate additional `Memory` every call during the construction of the nodes. However,

for `arr1`, Truffle/C assumes only the space of a pointer during construction time. During runtime, a node initializes this pointer with the variable size storage, which is essentially a call to `malloc`. All accesses to the variable size array first dereference the pointer, so that the behavior is transparent to the program. It is important that a node frees this dynamically allocated space again after leaving the scope of its declaration [23, C99 6.2.4.6].

```
int func(int n) {
  int arr1[n];
  int arr2[10];
}
```

Listing 6.3: Function with a Variable Length Array

A special case of arrays is when they are function parameters or return values from functions. Although the function signature might suggest that a caller can pass or return them by value, an array as a function parameter is always a pointer [23, C99 6.7.5.3.7]. The different signatures in Listing 6.4 are thus all pointers to arrays. Clang, and thus also Truffle/C, treat parameter arrays as pointer.

```
void func(int arr[10]);
void func(int arr[]);
void func(int *arr)
```

Listing 6.4: Function Signatures with Arrays as Expected Parameters

Another special case since C99 are flexible array members. A flexible array member can only occur in a structure and has an incomplete array type. Truffle/C basically ignores this field by giving it an address with storage size of zero. Its purpose is to have a variable size object in the structure. A programmer can reserve space for this object while allocating the structure by `malloc`. When space additional to the size of the structure is allocated, then the flexible array member behaves as if the space would belong to it [23, C99 6.7.2.1.16]. However, Truffle/C does not support passing structures with flexible array members per values to functions. To implement this, it essential to follow the ABI implementation for native compatibility.

Listing 6.5 shows how a programmer can use a flexible array member: The structure packet actually has a size of four, thus only including the `int` field. When the programmer allocates a structure by `malloc` and adds additional storage, this storage belongs to the data array. In the example, the program can use the array simply as `char data[5]` in most of the cases. Since the allocated storage exists anyway, the flexible array member essentially only needs to provide an address to access this memory, as long as no caller passes it to another function by value.

```c
struct packet {
  int length;
  char data[];
};


int main() {
  printf("%d", sizeof(struct packet)); // 4
  struct packet *p = malloc(sizeof(struct packet) + sizeof(char[5]));
}
```

Listing 6.5: Usage of Flexible Array Member

6.4 Literals and Enumeration

6.4.1 Number Literals and Enumerations

For numeric literals, the value of the respective number is stored and directly returned upon execution. Such a node exists for every numeric primitive type.

Enumeration constants, the member of enumerations, are simply named `int` constants [23, C99 6.2.5.16]. A programmer can also assign `int` values to enumeration members, in which case the following members obtain their constant by consecutively adding one to the previous constant [23, C99 6.7.2.2]. Listing 6.6 shows the declaration and usage of an unnamed and named enumeration. The first value of both enumerations is zero. The named enumeration then specifies the second member to be zero, thus having the same value as the first member.

```
enum kind {
  _int, // 0
  _double = 0,
  _short, // 1
  _float // 2
};

enum {
  ok, // 0
  invalid // 1
};

int main() {
  enum kind type  = _short;
  int code = ok;
}
```

Listing 6.6: Manually and Automatically Assigned Enumeration Constants

Truffle/C stores the constants for each enumeration member. When the program references an enumeration member, it instead constructs an `int` constant node with the respective value.

6.4.2 String Literals

String literals require a more complex implementation: Truffle/C, as well as C [23, C99 5.2.1.2], treat strings not as high level objects but as arrays of `char`. Truffle/C also has to add a zero byte to `char` arrays, since they are by convention terminated by `'\0'` [23, C99 5.2.1.2]. Truffle/C has to discern whether the left side of a string literal assignment is an `char` array or a pointer to `char`. The initialization of the array `arr3` in Listing 6.7 uses the string literal to "initialize the elements of the array" [23, C99 6.7.8.14], hence leaving the content modifiable. On the other hand, the interpreter initializes `arr1` and `arr2` to point to an object with the `char[]` value. The specification defines that modifying such an array is undefined [23, C99 6.4.5].

```
int main() {
  char *arr1 = "abc";
  char *arr2 = "abc";
  char arr3[] = "321";
}
```

Listing 6.7: Mutable and Immutable String Constants

Accordingly, Truffle/C handles the two cases differently: For string literals assigned to pointer, Truffle/C reserves and assigns static space during the construction of the nodes. Truffle/C only allocates a string literal once and assigns the same address on consecutive allocations. Thus `arr1` and `arr2` are pointing to the same address. On the other hand, Truffle/C constructs a node for `arr3`, that copies the four characters (including `'\0'`) to the `Memory` space starting from `arr3`. The actual assignment copies the string at runtime and hence stays modifiable.

C also supports wide chars that are not implemented yet in Truffle/C. However, wide chars follow the same concept as chars and thus would mainly differ in the usage of the Java data type to represent them.

6.4.3 Compound Literals

Compound Literals are initializer lists preceded by a cast. They can be used, e.g., for constructing array, structure and union literals. According to the standard, they either have static or automatic storage lifetime, depending on whether they are global or local. Truffle/C respectively constructs either a local or global `Memory` node, and allocates the `Memory` once global or per function call. A `CompoundLiteralNode` has a block of values and uses the address of the allocation block as an offset, to write the content of the values to the `Memory` location. To conform to the native ABI, it is essential that these values have the correct alignment. As Listing 6.8 shows, the implementation thus has to regard the type of the compound literal. While both assignments have the same initializer list, the first compound literal is a structure, while the second one is array. The ABI requires that the structure values are aligned at offsets 0, 8, and 16, while the `int` array must be aligned sequentially (at offset 0, 4, and 8).

```
struct test {
  int a;
  long b;
  int c;
};

int main() {
  struct test t = (struct test) {1, 2, 3};
  int *p = (int[]) {1, 2, 3};
}
```

Listing 6.8: Compound Literal Assignment to a Structure and an Array

Chapter 7

Operations

This chapter introduces how Truffle/C implements operators. A short presentation of the different kinds of reads and writes that Truffle/C has to produce is followed by the common arithmetic, logical, and bitwise unary and binary operators. The chapter also shows the implementation of the member access operators.

7.1 Reads and Writes

Since C includes the dereference and address-of operator, it raises the question at which point in time during the construction of the nodes Truffle/C can produce a node to read a value. A first approach might be to immediately read the value when it sees a dereference operator and otherwise return the address without reading. Figure 7.1 shows that this approach does not work for the expression `&(*result)`. The expression `&(*result)` where `result` is of type `int *` first dereferences the pointer and then takes the address again, resulting in the same address value as if not applying the operators at all. When Truffle/C encounters the variable, it would first construct a node for the memory location. After seeing the dereference operator, it would follow with a node for reading the value from the respective location. However, when it processes the address-of operator, it should again go one step back to just get the address and not read the value. At this time, it already set up the Truffle/C node and just deleting it would not be a clean solution.

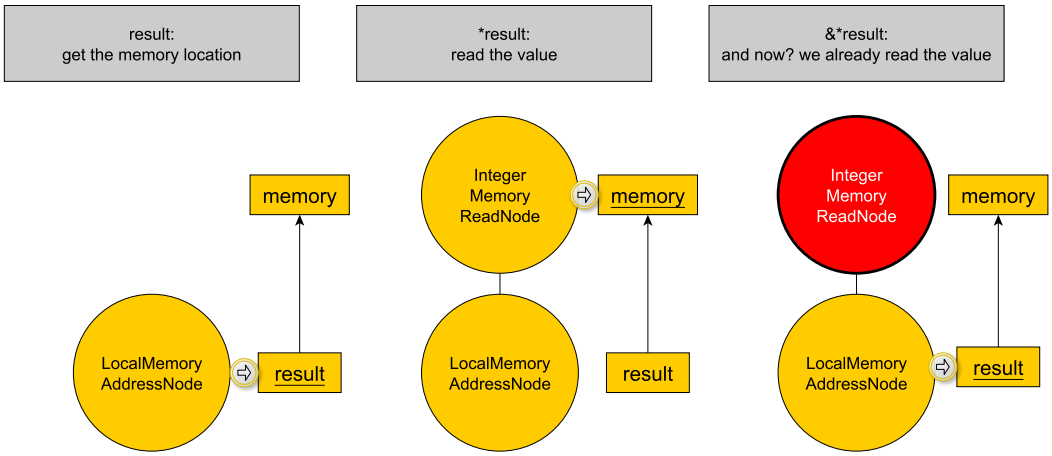

Figure 7.1: Naive Approach for Reading Values (Not Working)

One working approach is to always read "one level higher", i.e., when we know if we need an address or a value. This way, we also solve the issue of whether we want to receive a lvalue or an rvalue. When referencing a variable, Truffle/C cannot determine at the processing level of the variable in the AST, if it should construct a node to read the value or return the address: When the variable occurs at the left side of an assignment it should return the address, and if it occurs on the right side it should return the value. A simpler approach might also be to save reads symbolically and construct all the reads in the end. However, since in Truffle/C the location of data does not necessarily have to be the same for all reads, Truffle/C implements an approach, where a single location field for the next level is sufficient.

When processing the previous expression `&(*result)` Truffle/C first encounters the dereference operator and notes the potential read, but ignores it after seeing the address-of operator. Truffle/C thus prevents removing a node as the previous strategy would have required.

Figure 7.2 with Listing 7.1 shows an extensive example for the read strategy that Truffle/C employs: When Truffle/C encounters the pointer variable `p`, it first constructs only a `FrameSlot` as identifier, but does not read the address yet. On the next level, Truffle/C knows that it should construct a node for a pointer addition. Only then, Truffle/C creates the node for reading the address from the `Frame` and another one for adding the offset to the pointer (both displayed in one node). When Truffle/C encounters the first dereference operator, it notes that it potentially should read the value, but only reads when it encounters the second dereference operator. Truffle/C creates this second dereference operation when it knows that the expression is a rvalue, which is at the point when it should construct the `int` addition with 2. If Truffle/C would already construct the read one level before at `*(*(p+1))`, it would fail when the expression occurs at the left side of an assignment, as then the address and not the value would be needed.

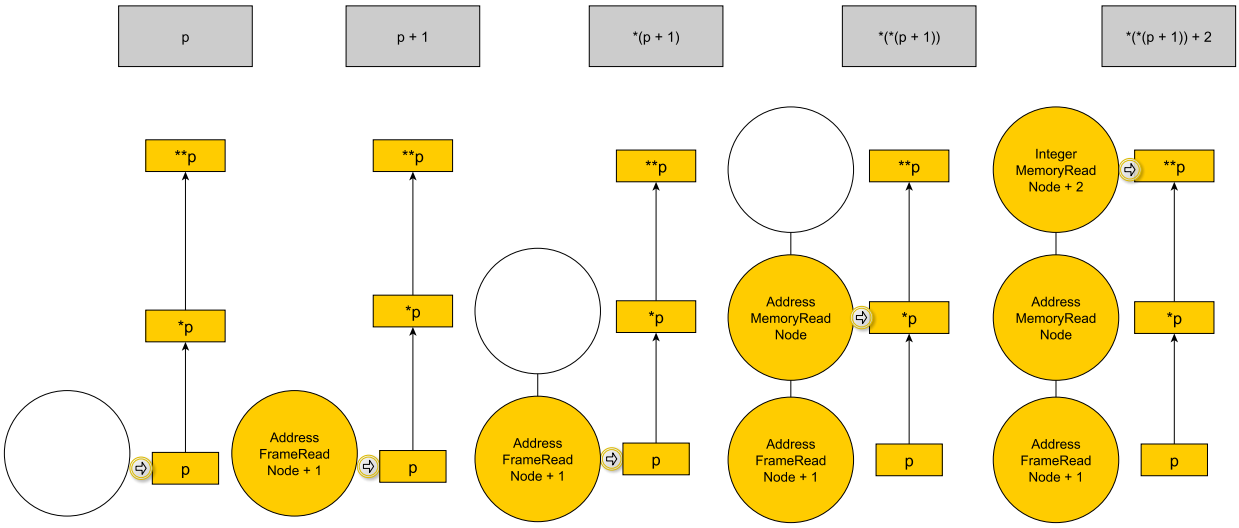

Figure 7.2: Truffle/C Approach for Reading Values Applied on Listing 7.1

```
int **p;
*(*(p + 1)) + 2;
```

Listing 7.1: Reading a Value from an `int **p`

As Section 6.2 explains, Truffle/C uses three different kinds of variable types for which the runtime uses different storage concepts. Figure 7.2 shows, for example, that the address of the pointer is in the `Frame`, while the dereference operation reads the value from the `Memory`.

Figure 7.3 shows the different storage locations for the primitive type variables in Listing 7.2. The nodes depicted are the read nodes that the function call to `printf` receives. Truffle/C reserves space for variable `a` at the node construction time. For the initialization, a global block for the program initializes global variables before the execution of the `main` function starts. Since Truffle/C stores globals in the `Memory`, the read first produces a `GlobalMemoryAddressNode` that contains the absolute address and is accordingly read by a memory read node, which is specialized to Integer. Truffle/C reads the static variable `b` in the same way. However, the linker initializes the variable when it loads the function for the first time. Variable `c` is never the target of an address-of operator and hence Truffle/C assigns it to the `Frame`. A `FrameSlotNode` returns a `FrameSlot` upon execution, which the `IntegerFrameReadNode` uses to read the value from the `Frame`. The next variable `d` is the target of an address-of operator and Truffle/C thus stores it in the `Memory`. However, the `LocalMemoryNode` remembers the address as an offset inside the memory allocated per function call and not as a global address. Truffle/C reads the pointer address for variable `e` from the `Frame`, since the pointer variable itself is never target of the address-of operator. However, the `IntegerMemoryReadNode` then reads the value from the `Memory`.

```
int a = 3;

int main() {
  static int b = 4;
  int c = 5;
  int d = 6;
  int *e = &d;
  printf("%d %d %d %d %d", a, b, c, d, *e);
}
```

Listing 7.2: Function with Writes and Reads with Different Truffle/C Memory Types

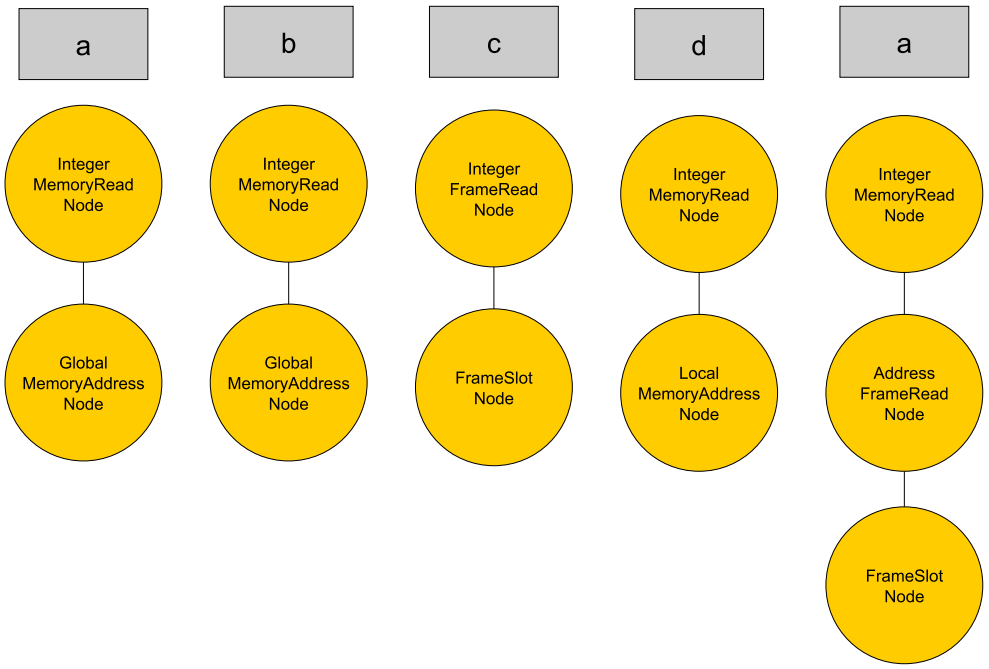

Figure 7.3: Different Truffle/C Read Nodes for the Variables in Listing 7.2

Structures, unions and arrays are stored in the same ways as the primitives, except that they are never put into the `Frame`.

7.2 Signed and Unsigned Operators

Truffle/C does not only have to provide all kinds of reads and writes for `Frame` and `Memory`, but also for the different data types, as well as for signed and unsigned. This also includes augmented assignments and unary increments and decrements. This means that Truffle/C has to provide nodes

for the augmented operators `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, as well as the post and pre increments for all the data types, for both `Memory` and `Frame`. Division, modulo and right shift need augmented assignment nodes in signed and unsigned versions. The signed variant of the right-shift is simply the same Java operator, while in Java `>>>` performs the unsigned right shift operation. For the unsigned division and modulo, Truffle/C casts up to the next higher data type to perform the operation. For `unsigned long` a node implements the binary version of the long division algorithm. The JDK 8 will allow an easier implementation with its new unsigned API which also provides an unsigned **`long`** division[1].

Additionally, bit fields require their own nodes for augmented assignments and unary increments and decrements. Although it results in a high number of nodes, this approach is clean and efficient, and all nodes contain short and non-complex code. Listing 7.3 shows the execute method of a `SignedLongDivFrameAssignNode` node, i.e., the signed version of the `/=` operator that Truffle/C uses on the `Frame`. Truffle DSL provides `val2` as the the result value of the node of the right side of the assignment. For a `Memory` node, the second argument would not be a `FrameSlot`, but a **`long`** with an address, that the node would use to read and store the value from the `Memory`. A bit field additionally has `bitSize` and `bitOffset` arguments, to selectively write certain bits.

```
public final long executeLong(final VirtualFrame frame, FrameSlot slot, long val2)
    {
  final long val = frame.getLong(slot);
  long result = val / val2;
  frame.setLong(slot, result);
  return result;
}
```

Listing 7.3: `SignedLongDivFrameAssignNode` Execute Method

Truffle/C implements the arithmetic and bitwise operators using the same principles as for the augmented assignments.

7.3 Pointer Arithmetic

For the additive operators plus and minus, not both operands are necessarily numeric types. In this case, the standard expects, that an addition of a pointer `p`, pointing to an `int i` produces a result, where the result of the addition points to the `i` th element of where `p` points to [23, 6.5.6.8]. Truffle/C thus calculates the result by taking the pointee type size and multiplying it with `i`, and then adding it to the address of `p`. Figure 7.4 shows the pointer addition (casts omitted for simplicity's sake) of

[1] `docs.oracle.com/javase/8/docs/api/java/lang/Long.html#divideUnsigned-long-long-`

`p + 3`, where `p` is a `short`. Since the pointer addition specifies to take the third element of a `short` array, Truffle/C computes the offset from the pointer `p` as the multiplication of `3*2`.

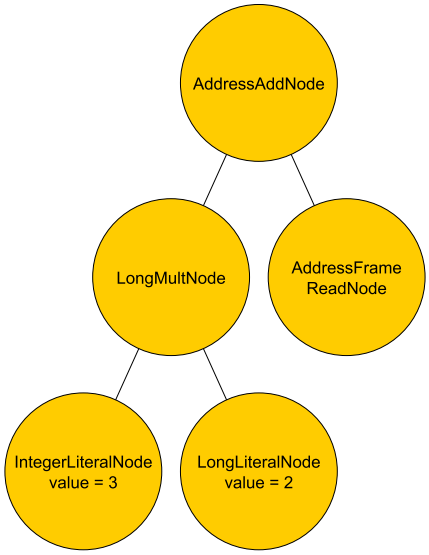

Figure 7.4: Truffle/C Nodes for a Pointer Addition

7.4 Comparison and Logical Operators

The comparison operators are `==`, `=!`, `>`, `<`, `>=`, and `<=`. The result type of a comparison is an `int` and returns one, if the condition is true and zero, if the condition is false [23, C99 6.5.8.6]. If the comparison value is inside a condition of a control statement, Truffle/C wraps it - in the same way as for non comparison operators - in a `ConditionNode` that produces a **`boolean`** value.

For the unsigned comparisons greater than, greater than or equal to, less than, and less than or equal to, the Java comparison operators are not directly applicable. Instead, a case distinction first has to determine the sign. If the sign is equal, Truffle/C can perform a normal comparison, and otherwise has to decide on the basis of the sign. The operators `!=`, `==`, and `=!` have a common version that is applicable for both signed and unsigned.

The logical operators `&&` and `||` perform a short-circuit evaluation where the node only executes the right expression, if the left expression is true in the `&&` case or false in the `||` case.

7.5 Member and Array Operators

For accessing members of a structure or union, Truffle/C has to handle two cases: The target can be either a structure or a pointer to a structure. Listing 7.4 shows these two cases. The programmer has to use a different syntax, namely a `->` if the left side is a pointer and `.` otherwise.

```
struct test {
  int a;
  int b;
};

int main() {
  struct test t;
  struct test *t1;
  int val1 = t.b;
  int val2 = t1->b;
}
```

Listing 7.4: Two Ways for Accessing a Member in a Structure or Union

Figure 7.5 shows the nodes that Truffle/C constructs for these cases. For the case without pointer Truffle/C directly takes the `LocalMemoryNode`, adds the offset to the address with the `StructElementMemoryAddressNode` and then reads the `int`. For the pointer case, an additional read is necessary: The `AddressFrameReadNode` first reads the pointer from the `Frame`, before adding the offset and proceeding as before.

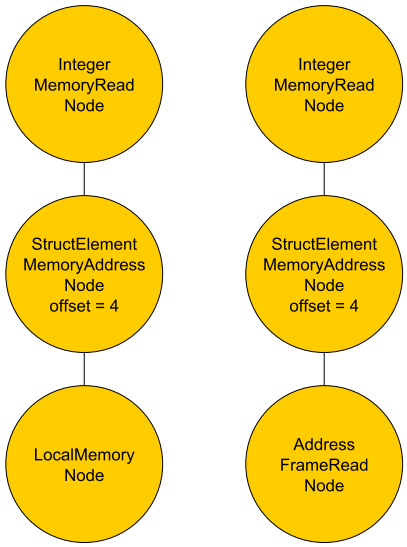

Figure 7.5: Truffle/C Read Nodes for Accessing the Members in Listing 7.4

For reading arrays, a similar concept as for the additive pointer nodes applies: Truffle/C multiplies the value of a subscript with the stride and adds it to the address. Truffle/C processes pointer subscripts from left to right, i.e., the less frequently varying subscripts when sequentially iterating through an array first. Listing 7.5 shows an example of an array read. Truffle/C first assigns values from the one-dimensional initialization list to the two-dimensional array. Then, Truffle/C reads from this array to print the value.

```
int main() {
  int arr[5][2] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int val = arr[3][1];
  printf("%d", val); // 7
}
```

Listing 7.5: Subscript Expression for Array Access

Figure 7.6 shows the nodes that Truffle/C constructs for Listing 7.5. It first constructs the inner subscript for `arr[3]`. On this level, we can see that we have an array that contains five elements which in turn contain arrays with two elements. The result of the first subscript is hence an array of two elements. To select the element from the array, another ArrayElementAddressNode is required.

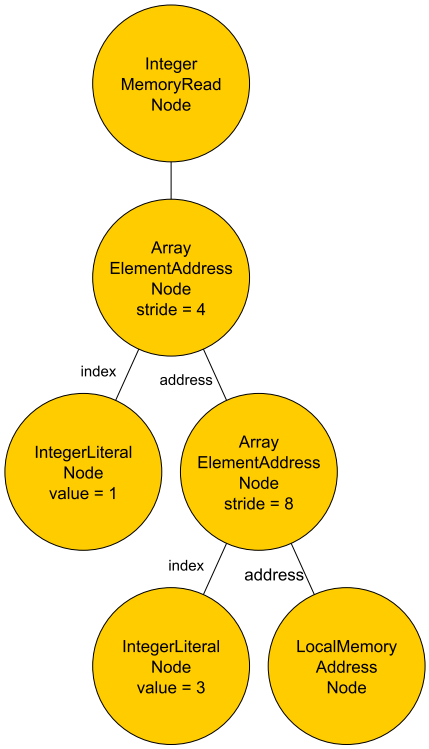

Figure 7.6: Array Access Example

7.6 Other Operators

C also supports other operators or operations such as `sizeof`, `alignof`, and the comma operator.

The `sizeof` operator yields the size of its operand in bytes. Truffle/C constructs a node that returns a literal with the memory size of the type. Truffle/C currently fails for variable length arrays, where it would have to evaluate the operand during run-time [23, 6.5.3.4]. The `alignof` operator returns the platform-dependent alignment of a type.

A less well known operator is the comma operator which first executes its left operand as a `void` expression, and returns the right side of the expression. Listing 7.6 shows an example of its usage. Truffle/C produces a node which executes the two expressions and returns the second one.

```
int main() {
  int t;
  return t = 2, 3;
}
```

Listing 7.6: Usage of the Comma Operator

Chapter 8

Control Structures

This chapter introduces the implementation of common control structures, such as while and for loops, switch, and if operators. Also, it presents how Truffle/C handles goto, which Java does not support at the language level. The last part of this chapter discusses how Truffle/C uses the Truffle API to communicate branch probabilities to Graal.

Many control structures of C can be directly mapped to Java control structures such as for loops, if-else statements, break and continue. The Truffle/C interpreter also has to support the goto statement, which is not implemented in Java. Truffle/C can partially support goto by its implementation that uses Java exceptions.

8.1 If-Else, Ternaries, and Switch

Listing 8.1 shows a very straightforward implementation of an if-else statement in Truffle/C. If the condition is true, the node has to execute the if case and otherwise the else case.

```
@Override
public void executeWithoutLabel(VirtualFrame frame) throws ControlFlowException {
  if (condition.executeCondition(frame)) {
    thenPart.executeWithoutLabel(frame);
  } else {
    elsePart.executeWithoutLabel(frame);
  }
}
```

Listing 8.1: Simple Truffle/C Implementation for If-Else

However, this approach does not benefit from the profiling feedback an interpreter could gather.

```
int func(int a) {
  if (a % 2) {
    return 1;
  } else {
    return 0;
  }
}
```

Listing 8.2: C Function Consisting of an If-Else Statement

Listing 8.2 shows a simple function. If a caller passes random values to this function, then Truffle/C executes both branches. However, if the function always receives even or uneven numbers, then Truffle/C either only executes the then or the else case. Truffle/C exploits this by constructing an uninitialized case and rewriting it on the first execution: If the then branch is executed, the node rewrites to only execute the then case and deoptimize and rewrite to a generic case that handles then and else, if the else branch is executed at a later time. The same logic applies when the else branch is executed first. Figure 8.1 shows the corresponding state diagram.

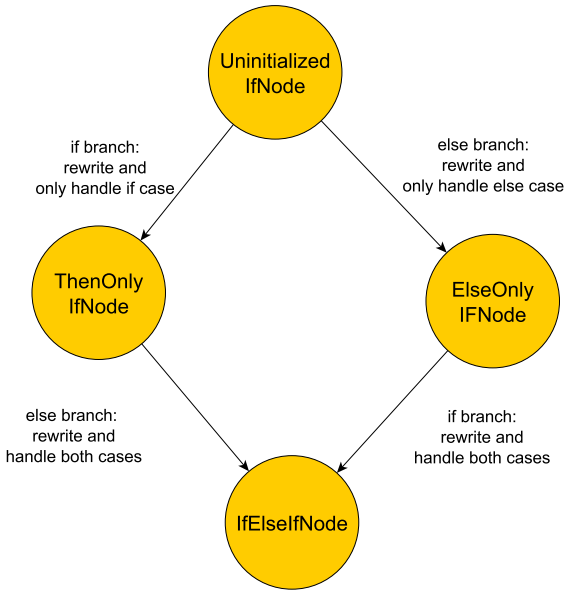

Figure 8.1: State Diagram for the Truffle/C If-Else Specialization

For the function in Listing 8.2, this will only eliminate an additional *movq* instruction and replace it with a deoptimization call. However, in less trivial cases, the machine code can for example do without bigger portions of code for handling error cases or eliminate dead code.

The same logic can be applied for the ternary nodes. The only difference is that ternary nodes return a result and if nodes do not. Switches are implemented in a similar way, so that they only include code for branches that were executed before. Careful tuning is necessary in order to only perform this optimization when it is beneficial.

8.2 Loops

The implementation of loops is similarly straightforward in Truffle/C: We can again map the semantics to the loop statements in Java. However, break and continue statements need to be modeled that interrupt the normal flow in a loop. These two statements are modeled with Java exceptions: A break inside a loop throws a `BreakException` and continue a `ContinueException`. As Listing 8.3 shows, the `WhileNode` has to catch these exception. The continue statement is equivalent to jumping to the end of the enclosing loop [23, C99 6.8.6.2]. The break statement terminates the execution of the enclosing loop (or switch). Thus, the `WhileNode` catches the `ContinueException` inside the end of the loop and the `BreakException` after the loop.

```
try {
  while (condition.executeCondition(frame)) {
    try {
      body.executeWithoutLabel(frame);
    } catch (ContinueException ex) {
    }
  }
} catch (BreakException ex) {
}
```

Listing 8.3: Simple Truffle/C Implementation for the While Loop

An optimization Truffle/C performs, is to specialize on whether a break or continue can occur inside the current loop, because Graal cannot necessarily optimize such a case. If one of the exceptions does not occur, Truffle/C can construct a node, which does not handle the respective exception. After compilation, the machine code thus also does not need code to handle the exception.

8.3 Goto

In C, goto allows unconditional jumps to a label in the enclosing function. Java does not support the goto statement. The goal of our implementation is to show that goto can be implemented with

modest effort and without impacting the performance of other statements. This requirement bases on the assumption that goto is seldom used, after having been discussed and criticized in the past [14]. Today, goto is mainly used for error handling as C provides no other exception handling mechanism. Such an approach is, e.g., recommended for error handling code in the Linux kernel [7, 36]. We did not decide for an approach where goto is transformed to structured control flow [17] but an exception based approach in order to preserve the high level information in the AST.

In the implementation each statement node has an `executeWithLabel` and `executeWithoutLabel` method as shown in Listing 8.4. The execution of a function starts with `executeWithoutLabel` as shown in the first AST (from top to down, left to right) in Figure 8.2. When execution reaches a goto as in the second AST, the node throws a Java `GotoException` which contains an identifier for the target label, that the enclosing function catches as seen in the third AST. The function then rewrites itself to start further executions with `executeWithLabel`. After the rewrite, the `labelId` argument is set to the identifier indicated by the `GotoException` as target and the execution continues with `executeWithLabel` as displayed in the fourth AST, starting from the first statement of the function. Every statement checks if the passed `labelId` allows execution. Since the statements are not allowed to execute, they return the unchanged label as a return result. After a `GotoExeception`, only the label can resume execution and sets the label to an `EXECUTE` label as shown in the fifth AST. Consecutive statements then continue their normal flow when they see the `EXECUTE` label and resume their execution as displayed in the sixth AST. When the function executes the next time, execution immediately starts with `executeWithLabel` with the identifier set to the `EXECUTE` label.

```
public abstract int executeWithLabel(VirtualFrame frame, int labelId) throws
    ControlFlowException;

public abstract void executeWithoutLabel(VirtualFrame frame) throws
    ControlFlowException;
```

Listing 8.4: Signatures of the Two Execute Methods for Statements

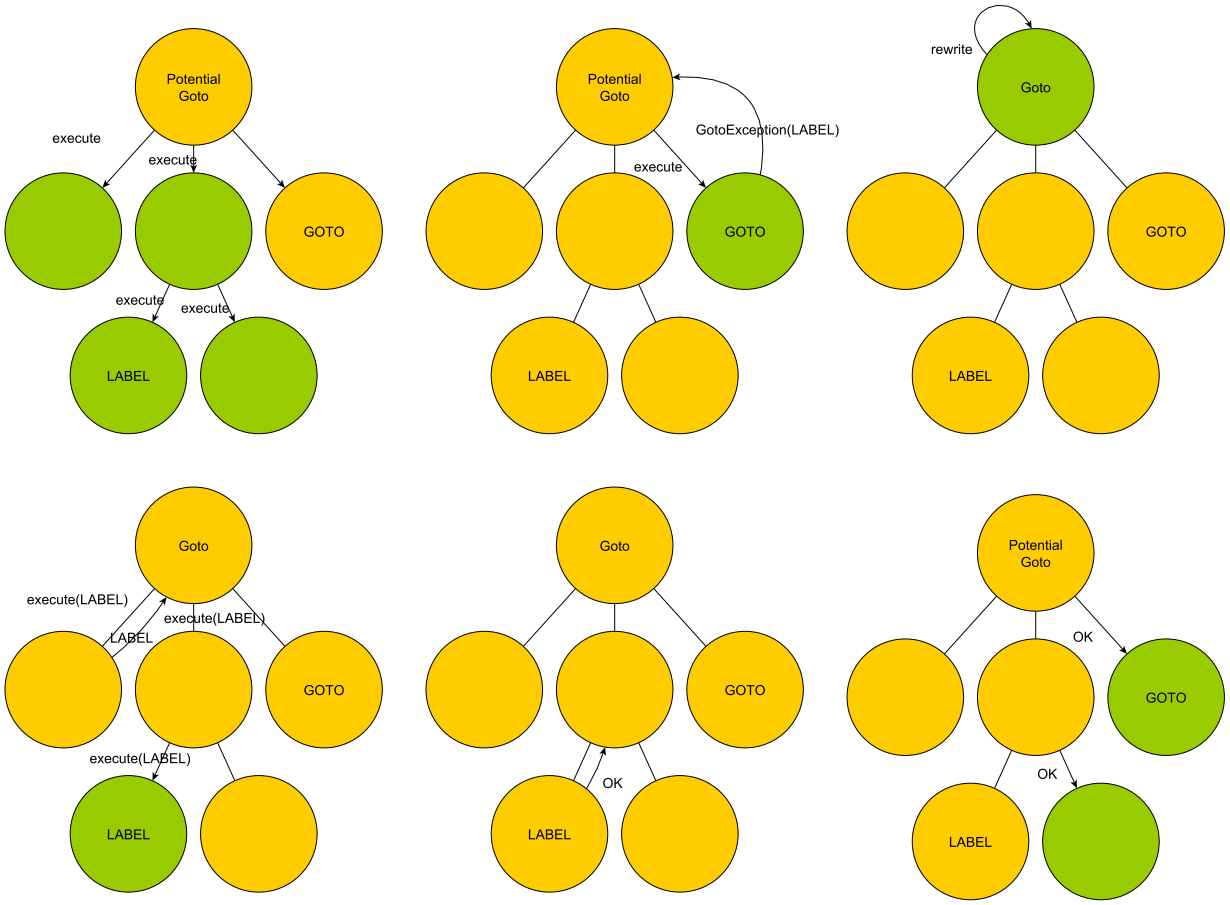

Figure 8.2: Goto Concept Depicted on an Execution in Truffle/C

The statements in the C program are guarded by subclasses of `StatementNode` in the AST. These statement nodes are specialized, to minimize checks and improve performance. One limitation of the goto implementation is, that it only works on the statement level. If an expression contains a goto, it produces a value and hence cannot return a `labelId`. Truffle/C neglects this case.

Listing 8.5 shows an example of the goto mechanism. The top level node Ⓐ speculates on not having to execute goto, although providing a try-catch block to catch a potential `GotoException`. In case valid arguments are supplied and enough memory is available, the goto will never execute and thus the execution will never have to perform additional operations.

The execution starts thus with `executeWithoutLabel`. In case of an error one of the two gotos Ⓒ is invoked and throws a `GotoException`. Ⓐ catches the exception and rewrites itself to a node that expects from then on that further gotos occur. The execution starts again with `executeWithLabel` with the `labelId` targeting the label `error` in Ⓓ. Since the whole block from Ⓑ to Ⓓ does not contain a label, a single check for the `EXECUTE` label is sufficient. As the `labelId` is not `EXECUTE`, Truffle/C

executes the next check at Ⓓ which is the target label. The target label matches and Truffle/C can continue normal execution. It does not need to check the statements after Ⓔ because after the last label in a function structured control flow will continue.

On the next function call execution starts with `executeWithLabel` and a `labelId` with `EXECUTE`. The check at Ⓑ succeeds and the statements are executed. Another check has to be performed at Ⓓ which again succeeds.

```
Ⓐ
int* allocateField(int n, int m) {
  Ⓑ if (n <= 0 || m <= 0) {
    Ⓒ goto error;
  }
  int size = n * m;
  int* alloc = malloc(size * sizeof(int));
  if (alloc == 0) {
    Ⓒ goto error;
  }
  return alloc;
Ⓓ error:
  Ⓔ fprintf(stderr, "field %d %d could not be allocated!", n, m);
  return 0;
}
```

Listing 8.5: Goto and Labels in a Function

8.4 Branch Probability Injection and Feedback

To supply profiling information for Graal, the Truffle API offers an interface to inject branch probabilities. Truffle/C uses these branch probabilities for if nodes, ternaries, and gotos. Listing 8.6 shows the injection of branch probabilities for the generic case of the if node. The if node and others capture only in interpreter mode how often if and else execute. This is achieved by using the `isInterpreter` compiler directive. Graal can use this information to, e.g., move a cold branch to better facilitate the cache.

```
if (CompilerDirectives.injectBranchProbability(thenCount / (double) (thenCount +
    elseCount), /* execute condition */)) {
  if (CompilerDirectives.inInterpreter()) {
    thenCount++;
  }
  // execute if case
  } else {
    if (CompilerDirectives.inInterpreter()) {
      elseCount++;
    }
  // execute else case
}
```

Listing 8.6: Injection of Branch Probabilities

To guide the decision when to inline and compile, loop nodes and gotos report how often they have been executed as shown in Listing 8.7. Also this is only performed in the interpreter. The two variables `incCounter` and `reportLoopCounter` are instances of `Runnable` which implement a method `run` with which they respectively increment an internal counter and report the counter to the Truffle run-time.

```
try {
  while (/* execute condition*/) {
    try {
      CompilerDirectives.interpreterOnly(incCounter); // thread to increment
          counter
        // execute body
    } catch (ContinueException ex) {
    }
  }
} catch (BreakException ex) {
} finally {
  CompilerDirectives.interpreterOnly(reportLoopCounter); // thread to report loop
      iterations
}
```

Listing 8.7: Report of Loop Iterations

Chapter 9

Case Study

To recap some of the concepts and present them in a bigger picture, this chapter presents a small example program. It explains again how the Clang modification generates the Truffle/C file, how the Java side generates the nodes, and how the nodes specialize during execution.

The program in Listing 9.1 sums up fibonacci numbers, stores them in an array, and eventually prints their square root.

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

Ⓔ enum {
  OK,
  NEGATIVE_ARGUMENT,
  NO_ARGUMENT
};

Ⓒ int returnCode = Ⓔ OK;

int fib(Ⓐ int n) {
  if (n <= 1)
    return n;
  return fib(n-1) + fib(n-2);
}

int main(int argc, char *argv[]) {
  Ⓐ int *arr = 0;
```

```
  if (argc <= 1) {
      printf(D "no argument provided!\n");
      returnCode = E NO_ARGUMENT;
      goto exit;
  }
  A int n = atoi(argv[1]);
  if (n <= 0) {
    printf(D "%d is not a positive number!\n", n);
    returnCode = E NEGATIVE_ARGUMENT;
    goto exit;
  }
  A arr = B malloc(n * F sizeof(int));
  A int result = 0;
  A int i;
  for (i = 0; i < n; i++) {
    arr[i] = fib(i + 1);
  }
  for (i = 0; i < n; i++) {
    A int printValue;
    if (arr[i] == 0) {
      printValue = 0;
    } else {
      printValue = pow(arr[i], 2);
    }
    printf(D "values[%d] ^ 2 = %d\n", i, printValue);
  }
exit:
  free(arr);
  return returnCode;
}
```

Listing 9.1: Case Study Complete Program

9.1 Truffle/C File

The first step that Truffle/C performs is a call to the Clang modification with the path to the C file. Clang reads the three included standard headers, and resolves all the macros they contain. Then, the Clang modification traverses the internal Clang AST and writes the AST representation to a single Truffle/C file (see Chapter 4). As Listing 9.2 shows, the type and function definitions from the header files usually constitute the biggest section within the Truffle/C file.

```
...
  FunctionDecl 93962912 malloc void *(unsigned long) 1 decl
    ParmVarDecl 93963072 unsigned long nonstatic 0
  FunctionDecl 93963168 malloc void *(unsigned long) 1 decl
    ParmVarDecl 93962656 __size unsigned long nonstatic 0
  FunctionDecl 93963840 calloc void *(unsigned long, unsigned long) 2 decl
    ParmVarDecl 93964000 unsigned long nonstatic 0
    ParmVarDecl 93964096 unsigned long nonstatic 0
...
```

Listing 9.2: Truffle/C File Header Includes Excerpt for Listing 9.1

The Truffle/C file that the Clang modification produces contains all the information for the node construction. Listing 9.3 shows for example the declaration of the pointer variable with its initializer, which is the result of the `malloc` call.

```
DeclStmt 1
  VarDecl 94039392 ptr struct element * nonstatic 1
    CStyleCastExpr struct element * 1
      CallExpr void * 2
        ImplicitCastExpr void *(*)(unsigned long) 1
          DeclRefExpr Function 93963168 malloc void *(unsigned long) 0
        UnaryExprOrTypeTraitExpr unsigned long sizeof struct element 0
```

Listing 9.3: Truffle/C File Malloc Call for Listing 9.1

9.2 Node Construction

During the construction of the nodes, Truffle/C can see that the variables marked with Ⓐ are not target of an address-of operator. Hence, Truffle/C puts them into the `Frame` and produces the corresponding read and write nodes, as explained in Section 7.1. Truffle/C always assumes that structures, unions, and arrays Ⓑ are in the `Memory`. The content of variable `arr`, i.e., the address of the pointee, is in the `Frame`, whereas the pointee itself (the array allocated in Ⓑ) lies in the `Memory`. Also, the global variable in Ⓒ lies in the `Memory`.

Truffle/C also allocates the string literals in Ⓓ already during the construction of the nodes, as Section 6.4.2 explains.

Truffle/C stores the unnamed enumeration as constant definitions as Section 6.4.1 explains and replaces the assignments of enumeration members with assignments of literals as in Ⓔ. The same happens

with the Ⓕ sizeof operator, as Section 7.6 explains. The literal four serves as replacement for the `sizeof`.

9.3 Execution

The execution of the program assumes an input parameter of `7` and a repeated execution of main, that would trigger compilation of the Truffle/C AST.

```
int main(int argc, char *argv[]) {
  int *arr = 0;
  if (argc <= 1) {
    Ⓐ // deoptimize and replace node
  }
  int n = Ⓒ atoi(argv[1]);
  if (n <= 0) {
      Ⓐ // deoptimize and replace node
  }
  arr =  malloc(n * sizeof(int));
  int result = 0;
  int i;
  for (i = 0; i < n; i++) {
    arr[i] = Ⓓ fib(i + 1);
  }
  for (i = 0; i < n; i++) {
    int printValue;
    if (arr[i] == 0) {
      Ⓑ // deoptimize and replace node
    } else {
      printValue = pow(arr[i], 2);
    }
    Ⓒ printf("values[%d] ^ 2 = %d\n", i, printValue);
  }
exit:
  free(arr);
  return returnCode;
}
```

Listing 9.4: Run-Time Version of Listing 9.1 with Deoptimization Points

Truffle/C starts the execution of the `main` function with `executeWithoutLabel` as Section 8.3 describes. Since the gotos for the error handling never execute, the function never has to invoke `executeWithLabel`. In the compiled code, there is only an exception handler left, to handle a potential `GotoException`, which never occurs for the input argument of `7`. Since furthermore the if statements for the error handling are not invoked, the whole block in Ⓐ (including the gotos) is also not contained in the compiled code. Instead, the if node performs the condition and would rewrite if the condition were true. In compiled code, the rewrite triggers a deoptimization and return to the interpreter to perform the replacement. Since the function logic does not allow that `arr[i]` is zero in Ⓑ Truffle/C can ensure that the if branch is never part of the compiled code. As before, not a static analysis determines this, but the fact that the if branch is only included if it was encountered before.

The linker loads the functions lazily, as soon as the function node executes for the first time. As Chapter 4 presents, the format and architecture support and partly implement lazy reading of the file and lazy construction of nodes. If the linker can find the function in the Truffle/C file, it returns this function, as for example for the `fib` function in Ⓓ. If it cannot find a function, it assumes that the function is native and invokes the native function interface as for Ⓒ with functions `atoi` and `printf`. However, Truffle/C provides intrinsics for the standard library math and memory functions. Instead of a native call, a specially implemented Truffle/C node serves as a replacement. This often makes the execution faster, since Graal can optimize (or apply again intrinsics to) these Java nodes. Figure 9.1 shows this concept, where the specially implemented Truffle/C node does not have to call a native function in order to execute `pow`, `malloc`, or `free`. The only native calls that remain are `atoi` and `printf`. The compiled code can perform these calls by a direct call to the functions, and hence without additional overhead [22].

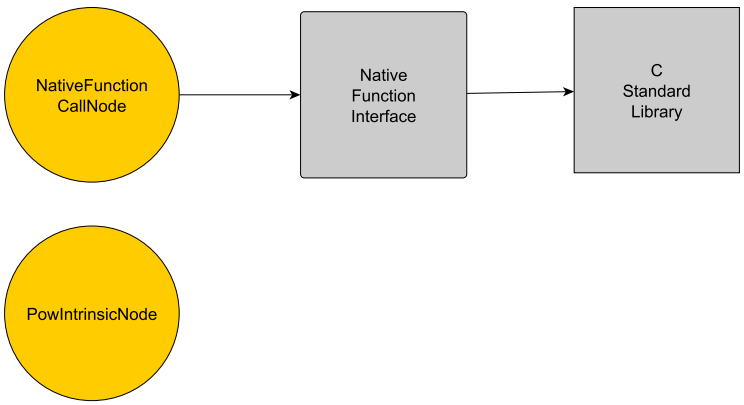

Figure 9.1: Truffle/C Intrinsics

Chapter 10

Evaluation

This chapter evaluates the Truffle/C implementation in three aspects. The first is peak performance on benchmarks, the second one completeness on GCC test cases, and the third one is Truffle/C as a Truffle language.

The benchmarks and test cases were executed by a Lenovo ThinkPad T430s with the CPU Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz and 8 GB of memory. It used an 64 Bit Ubuntu 12.04.2 LTS with an 3.2.0-40-generic kernel version. Truffle/C is of the revision 0c60f3798289 which is not publicly available and Graal version with revision 895f31682b88, which is available from the official OpenJDK Graal repository[1].

10.1 Peak Performance

The performance measurements use different configurations of Truffle/C to show the impact of different optimizations. Figure 10.1 shows the performance of these different configurations on five benchmarks of the Computer Language Benchmarks Game[2].

Our baseline is the configuration where all optimizations are enabled, and in addition to that we also measured the following configurations:

- All optimizations are disabled.
- Only memory intrinsics are enabled.

[1] Graal Mercurial repository: `http://hg.openjdk.java.net/graal/graal`
[2] The Computer Language Benchmarks Game: `http://benchmarksgame.alioth.debian.org/`

- Only math intrinsics are enabled.
- Only control optimizations (branch profiling, control node specializations) are enabled.

The performance is given as a score, computed by the execution time relative to the baseline. Thus, a lower bar means a faster execution time.

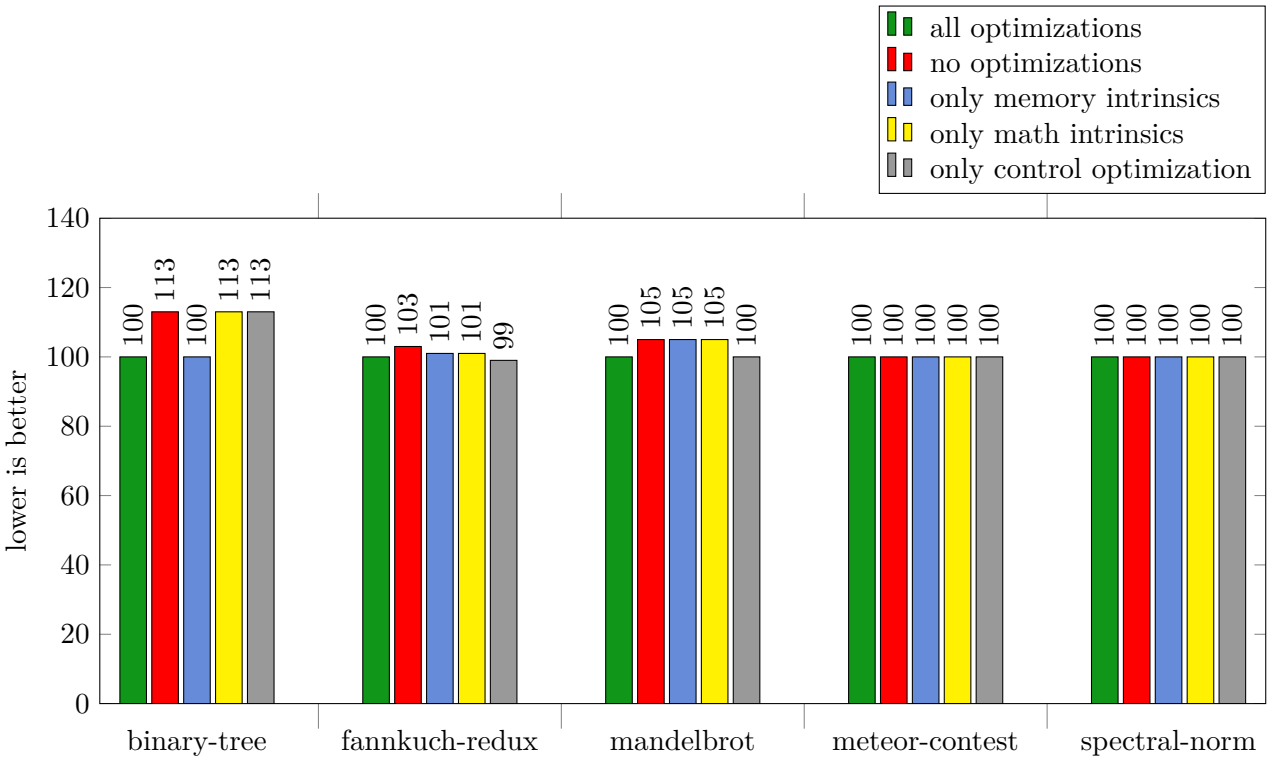

Figure 10.1: Performance Comparison of Different Truffle/C Optimizations on Six Benchmarks

The first observation in Figure 10.1 is that, for most benchmarks, the optimizations do not contribute to a significant performance gain. The two benchmarks *meteor − contest* and *spectral − norm* do not even show any change in the different configurations.

Since benchmarks likely contain little or no dead code, or code which executes only in exceptional cases like error handlers, the control optimizations seem to not significantly contribute to the performance. However, the absence of any gain for some benchmarks seems surprising. For example, the benchmark *meteor − contest* contains seventy-six `if-else` constructs. While forty-two `if-else` nodes contain and execute both `if` and `else`, Truffle/C constructs for thirty-two only the `if` path (i.e. the `if` also does not have an `else` branch in the source). The latter one is an optimization, which only gives a speedup in the interpreter and hence not for the peak performance. However, for two `if-else`, the benchmark actually never executes the `else` branch (i.e. for those which contain an `if` and `else` branch in the source). Truffle/C thus does not let Graal compile these `else` branches. Truffle/C is also able to

construct thirty-eight `while` or `for` nodes that do not need the code for handling a `break` or `continue` and one `for` node that handles `continue` but not `break`.

The benchmark that gains the most from the control optimizations is the *mandelbrot* benchmark with a speedup of 5%. The benchmark contains five `for`-nodes which handle neither `break` nor `continue`. The benchmark that shows a small benefit from the control optimizations is $fannkuch - redux$. For nine out of the ten `while` or `for` loops in $fannkuch - redux$, Truffle/C constructs a version that does not handle `break` or `continue`, while for one case a node is applicable, that only handles a `break`.

Regarding the math intrinsics, only the benchmarks $binary - tree$ and $spectral - norm$ contain one math intrinsic each which replaces the native call to the standard library. They seem to not influence the performance results. Mini benchmarks, that just execute math functions, show that some math intrinsics can give a high speedup but can also negatively influence the performance, depending on the substituted math function.

Only the memory intrinsics seem to highly influence the performance, namely in the benchmark $binary - tree$. This benchmark frequently calls `malloc` (and `free`), where the intrinsics pay off. Not using the intrinsic makes the execution 13% slower.

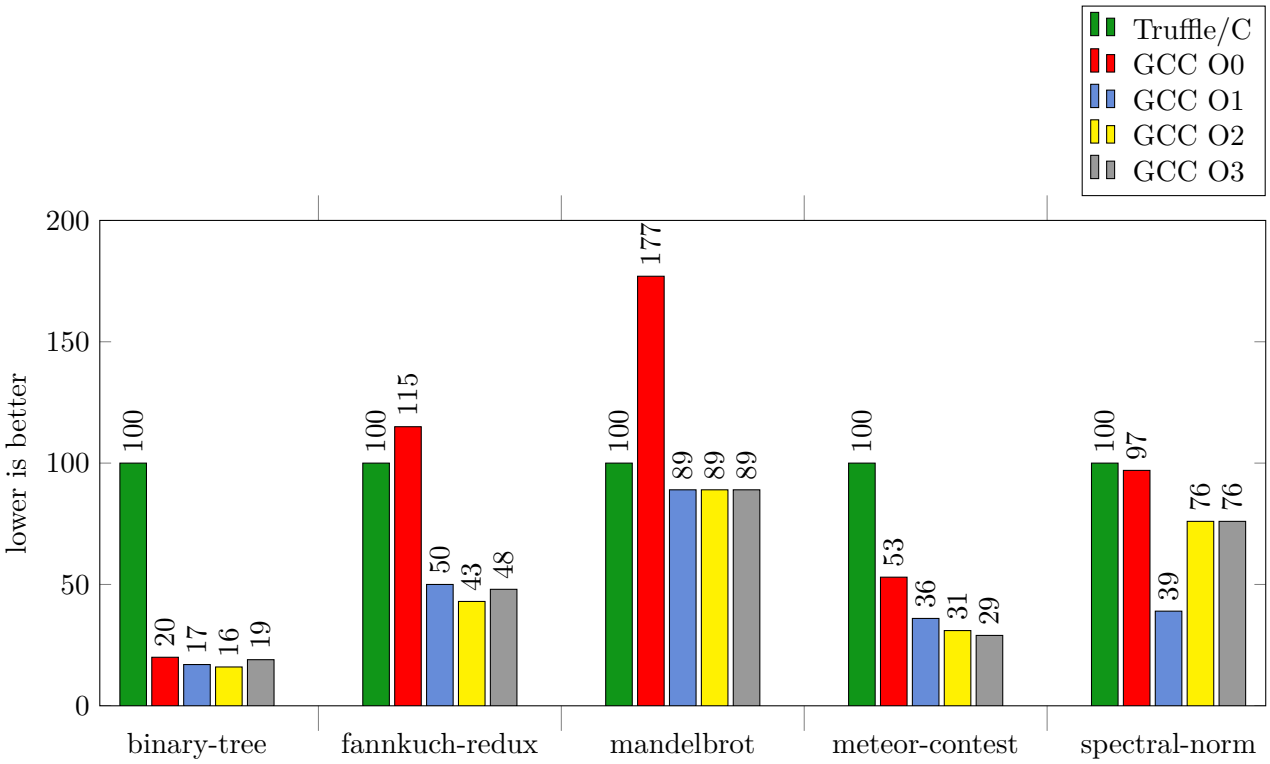

Figure 10.2: Performance Comparison of Truffle/C with GCC on Six Benchmarks

Figure 10.2 shows a performance comparison between Truffle/C and GCC. At the moment, most of the

performance depends on the `Memory` implementation, the fine-tuning of Truffle, and the quality of the compiled code that Graal produces. Since all the components are in active development, sometimes performance regression can occur. The thesis hence refrains from a detailed analysis of the benchmarks at this point. A more detailed analysis of the *mandelbrot*, $fannkuch - redux$ and $spectral - norm$ benchmarks can be found in the thesis of my colleague [21].

The performance of Truffle/C on the benchmarks thus ranges from 1.77 times as fast as unoptimized GCC code for $mandelbrot - redux$ to 5.26 as slow as highly optimized GCC code for binary-tree.

10.2 Completeness

The evaluation of Truffle/C's completeness uses the 1200 test cases from $gcc.c - torture/execute$, a part of GCC's testsuite. This test suite consists of "particular code fragments which have historically broken easily" [6, GCC 7.4].

As the directory name suggests, these test cases have to be *executed* to get a test result, thus differentiating themselves from GCC test that only have to be preprocessed, compiled, or linked. Since executing involves all the steps, and preprocessing as well as compiling bases on Clang's capabilities on a big part, this test suite is expected to produce representative results of Truffle/C capabilities and limits.

As Figure 10.3 shows in the left pie chart, Truffle/C successfully runs 841 out of the 1200 tests (70.08%). The analysis of the 359 (29.92%) remaining tests is divided into three parts, which correspond to three categories. The first group are Truffle/C errors with 186 failing test cases (51.81% of the failing tests). The second group are test cases that exercise the GNU extensions with 116 failing test cases (32.31% of the failing test cases). The third group of test cases can either not be compiled by GCC or Clang and make up 57 test cases (15.88% of the failing tests).

The three groups are test cases, that fail because of Truffle/C errors, that cannot be executed because of built-in functionality, and ones that fail either on GCC or on Clang. Taking only into account the test cases, that Truffle/C *should* be able to execute (i.e., only taking into account the Truffle/C failures) in its fully implemented state, Truffle/C executes 841 out of 1027 test cases (81.89%) successfully.

The finer categorization is not completely accurate, since many test cases fit into multiple categories. One example would be a test, that uses the not yet implemented complex numbers, relies on built-in functions and calculates the wrong result. The decision, to which category a test best fits, was thus sometimes a subjective one.

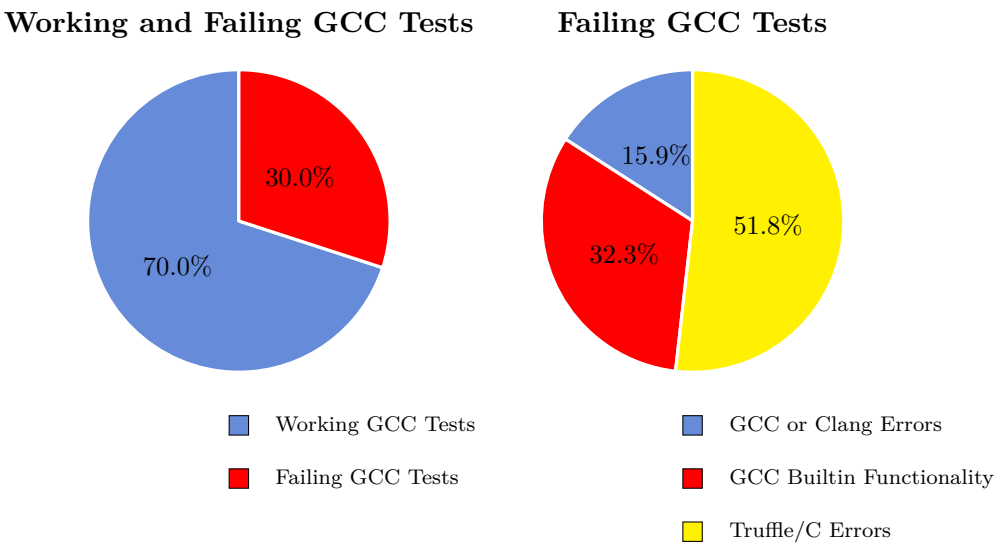

Figure 10.3: Working and Failing GCC Tests

10.2.1 Failing GCC or Clang Tests

First of all, 57 test cases (15.88% of the failing tests) can either not be compiled by GCC or by Clang. While GCC accounts for 9 (15.79%) failing test cases, Clang accounts for 48 (84.21%). These test cases seem to expose "real" bugs, e.g., assertion errors or crashes, and do not indicate trivial errors such as missing includes. To fix this problem, is in the scope of the Clang or GCC developers.

10.2.2 Truffle/C Failures

The test cases of this category should all be executable by Truffle/C in the future. Most "low hanging fruits" have already been fixed. Now, still 186 failing test cases (51.81% of the failing tests) remain. The remaining test cases are often specific problems, that are only exhibited in a special context. They are often not hard to find, but trying to fix them often results in regressions. Figure 10.4 shows the categories of the failing test cases.

28 test cases (15.05%) fail, because of the interface between Clang and the parser of Truffle/C. As Clang developers have not documented the constraints of the internal Clang AST for C, it is difficult to see which nodes and combination of nodes can occur, as well as what information is persistent in each node and which of it is optional. Thus, fixing bugs in this category requires a detailed review of individual test cases and usually needs a change of the Clang modification, as well as the Java parser. An example which occurs multiple times is a K&C notation for function parameters (see Listing 1.1), where Clang does not provide the types.

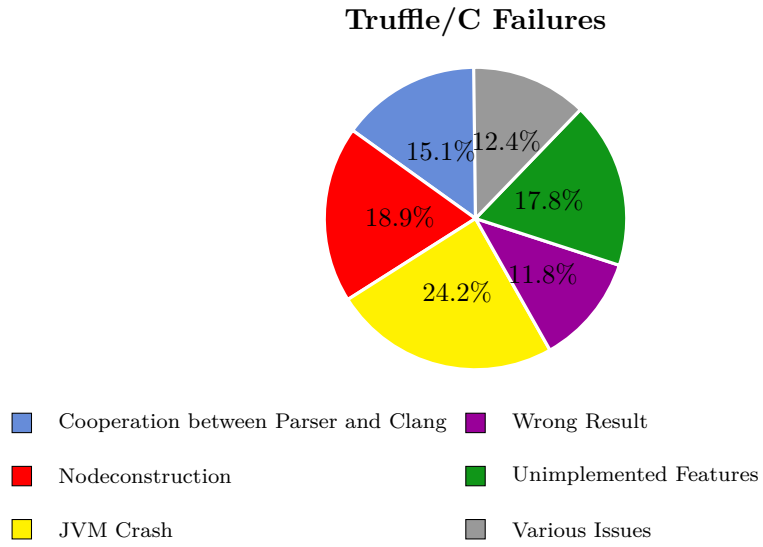

Figure 10.4: Truffle/C Failure Categories

35 test cases (18.82%) fail because of exceptions or failing assertions while constructing the nodes. Additionally, this category also includes some test cases with ASTs that can be constructed, but then fail during execution. These test cases mostly expose, that node constructions methods are not general enough to handle all the cases. Examples are multiple nested composite initializations or not handling function pointers sufficiently in the context of functions.

45 test cases (24.19%) crash the JVM. Crashes in the JVM can occur through the use of `Unsafe`, which Truffle/C uses to implement the `Memory`. This category contains test cases which use arrays, structures, and pointers. Bugs in this category have received little attention, since they are not always deterministic and the ones in other categories are usually easier to find and debug. Fixing bugs in other categories also from time to time resolves bugs in this category. w.

22 test cases (11.83%) compute an unexpected result, but do not crash. Previously, bugs in this category were mostly related to wrong or missing sign extensions for function calls, unsigned operations, bit fields or conversion between signed and unsigned data types. Also, using a wrong Java data type with Truffle DSL resulted in some bugs in this category.

33 test cases (17.74%) account for test cases which use not yet implemented features in Truffle/C. 15 test cases of this category exercise varargs for structures. 11 test cases use complex numbers, and 4 test cases use `stdout`.

23 failing test cases (12.37%) are related to various issues: For 11 test cases, Truffle/C cannot find the type of a variable. In another 7 cases Truffle/C does not free the memory for variable length arrays, e.g., in combination with gotos. In the remaining 5 cases, the linker fails.

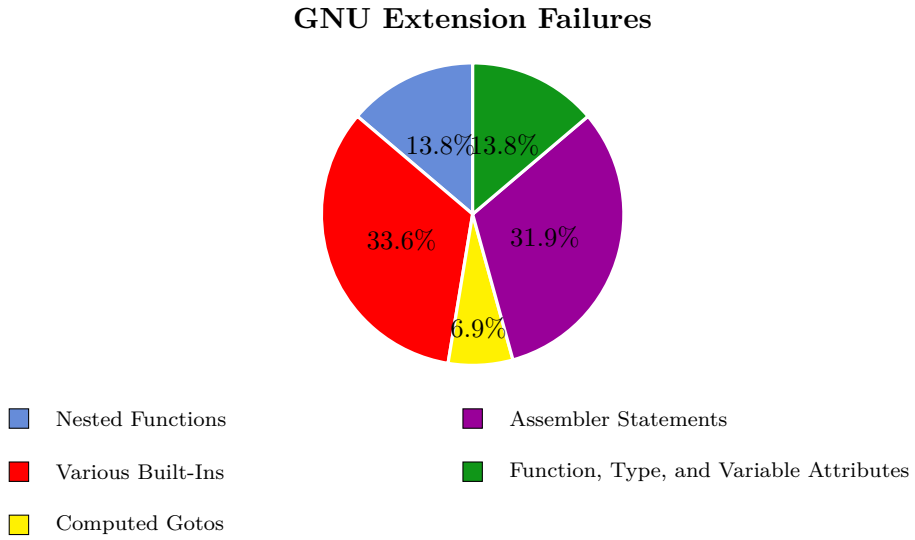

Figure 10.5: GNU Extension Failures Categories

In the test cases, sometimes the keyword `volatile` occurs. This type specifier enforces that reads to a variable are not "optimized away", e.g., for memory-mapped input or output, as well as for asynchronously interrupting functions [23, GCC 6.7.3]. This keyword is just ignored in Truffle/C.

Overall, these test cases do not exhibit essential limitations of Truffle/C. Only 33 test cases exercise not yet implemented features, which are not hard to implement in Truffle/C. However, since Truffle/C is a research prototype we try to keep the implementation simple and not introduce support for language features that are not yet needed. The other test cases relate to bugs which need to be fixed on a case per case basis.

10.2.3 Builtin GCC Functionality

A significant proportion of 116 failing test cases (32.31% of the failing test cases) go onto the account of the GNU extensions [6, GCC 6]. Truffle/C supports some of the GNU extensions, where they could be implemented easily. For example, built-in functions with prefix `__builtin_` often correspond to library functions with their name without the prefix [6, GCC 6.57]. Truffle/C thus maps them to the same non-builtin functions without prefix. Among other extensions, Truffle/C supports empty structures, expression statements, non-constant initializers, and the offsetof operator.

For a mature C interpreter or compiler, it is desirable to also be able to provide the GNU extensions, such as Clang does to a large degree [4]. For example, the standard C library on many Linux systems is the GNU C Library. The GNU C Library provides header files that test via feature test macros

which standards and extensions the programmer wants to enable. If the compiler can support these extensions, the programmer can use a wider range of the standard library [30, GLIBC 1.3.4]. The implementation of the GCC extensions is not within the scope of the thesis, and the test suite does not exercise all these features. Thus, the thesis only provides a sketch on how the implementation of this extensions could look.

37 failing tests (31.90%) rely on the `asm` statement. This statement is an extension and allows to execute assembler instructions in C. It is possible to implement direct support for such assembler instructions in Graal, by implementing a Graal capability, in a similar spirit as the native function interface [22]. The programmer has to specify the operands as C variables, instead of guessing the location of the data [6, GCC 6.41]. This makes an implementation for Graal sensible.

16 tests (13.79%) rely on nested functions. Nested functions are not only local to the block where they are defined, but can also access the variables of the outer frame [6, GCC 6.4]. This requires a change of the Truffle/C API for the `Memory` and `Frame`, to allow access to variables in the outer scope. In case of the `Frame`, Truffle/C needs to save a reference to the outer `Frame`, to be able to traverse the enclosing frames. Truffle implementations for dynamic languages already implement this [33, 41] and we thus expect a small effort to implement nested functions.

8 tests (6.90%) fail because they exercise computed gotos in combination with getting the address of a label [6, GCC 6.3]. Implementing this imposes no problem, since the value of a label is a constant that can be computed during node construction and given as an ID to the `GotoNode`. The computed goto then throws a `GotoException` with this constant value as target. This is an example where the structure of Truffle allows to implement a new feature with minimal effort.

16 tests (13.79%) fail because Truffle/C only partially supports attributes. With GNU C programmers can define attributes on functions [6, GCC 6.30], variables [6, GCC 6.36], and types [6, GCC 6.37], to guide the compiler. Examples of such attributes are inlining hints, specifications of alignments for variables, or the visibility of certain types. Most of these attributes are currently ignored, and Truffle/C prints a warning when encountering them. Some of these attributes are specific low level mechanisms, such as disabling interrupts for special targets in a function. While theoretically, Graal capabilities could provide such support, the implementation of these attributes does not seem feasible. High-level attributes are easier to implement. Some of them can be used for specialized node construction. One example is an attribute which specifies that a function does not return. In this case, a function call node can be constructed which does not handle a `ReturnException`, and results in more compact machine code upon compilation.

39 failing tests (33.62%) use other built-in operators, functions, or types. Examples for operators are offsetof [6, GCC 6.50] to obtain the offset of a field in a structure, or the `__real` [6, GCC 6.10] operator

to extract the real part of a complex number. One of the builtin-functions that does not directly map to standard library function is `__builtin_prefetch` [6, GCC 6.57], to move data into the cache before the program accesses it. Like for attributes, these built-ins would have to be hard-coded in Truffle/C, or implemented in Graal.

Overall, the GNU C extensions did not highlight any significant problems of Truffle or Truffle/C. Some parts cannot directly expressed in Java, like the assembler statements, and thus would require machine code support that would have to be generated by Graal. However, some built-ins are not likely to occur in real world code, thus making an implementation in the near future infeasible. In other cases, the programmer can use built-ins to help constructing specialized nodes for Truffle, while other optimization targeted attributes can be ignored, since programmers often have a poor insight on how their program actually performs.

10.3 Evaluation as a Truffle Language

10.3.1 Truffle/C and Previous Implementations

The first implementation of a Truffle language was TruffleJS for JavaScript, and serves as the first prototype for dynamic languages [43]. Efforts have been set to use Truffle for implementing other languages: The author of this thesis previously built an interpreter for Python using Truffle [33]. The Python implementation for Truffle exhibits similar characteristics as the JavaScript one and uses similar techniques for property access, type and operation optimization. Since then, additional implementations for Python, Ruby, R, and J have become available [40]. All these languages are dynamically typed. Truffle/C is the first major effort to evaluate Truffle on a static language.

JavaScript, as well as other dynamic languages, can extensively apply rewriting: Since data is not statically typed in these languages, the nodes start with an uninitialized state and later rewrite themselves to the actual types encountered.

In contrast, the node types for C are statically known. Instead of starting with an uninitialized state, Truffle/C constructs the nodes according to the static types already when parsing the program. Accordingly, Truffle/C also does not rewrite any nodes because of type mismatches and does not use type decision chains [43]. Truffle/C also does not use polymorphic inline caches, since functions as the only applicable case in C are statically known most of the time. Exceptions are function pointers that can have multiple, compile time unknown targets. However, this was not a concern yet, since our target benchmarks do not exercise this feature.

The only case where Truffle/C uses rewrites is for control structures and when calling functions: A function call node first starts with an uninitialized state and rewrites itself once the function is actually executed. The file format of Truffle/C supports this by allowing an easy implementation of lazy resolving. If a function is never called in Truffle/C, the interpreter does not have to construct the AST for this function. Also, only at this point in time, the execution decides whether the function is a Truffle/C or a native function.

C cannot refer to outer stack frames, since standard C does not support nested functions (except in the GNU C extensions [6, GCC 6.4]) and the compiler can determine references to variables statically. This facts keeps references to the Truffle API small and the nodes very simple. Use of the Truffle DSL [42] allow the nodes to be minimal in terms of implementation code.

Similar to other Truffle implementations, Truffle/C uses the API to communicate the branch probabilities of loops and branches to Graal [42]. Also, control flow exceptions like `break`, `continue` and `goto` are implemented with Java Exceptions, same as in other Truffle implementations. Truffle/C also facilitates the API to guide function calls and inlining. Truffle/C uses the `Frame` to store local variables wherever possible. Since C has to interact with native libraries and refer to addresses, this is not always possible.

Overall, using the Truffle API also proved to be useful for the static language C. Truffle/C could facilitate code generation and basic API functionality from Truffle. In comparison to dynamic languages, Truffle/C did not use the Truffle facilities for type specialization and other more sophisticated optimizations. But even without that, the format of the interpreter allows Graal to perform optimizations to gain a peak performance that is not far behind long-standing industrial compilers.

10.3.2 Platform Dependence

A major advantage of using Java as a host language is that it is platform independent. Any platform which has Java installed can execute the Truffle/C interpreter. However, there are some parts of the project which restrict the platform independence or the correct execution on different platforms.

The first part of platform dependency is Clang. We modified the Clang parser, in order to write a representation of its internal AST to a file. Clang is a cross-platform project. However, the project has to be compiled for different platforms before its execution. To support several platforms, Truffle/C has to offer the modified Clang binaries for different platforms.

The Truffle/C interpreter is self sufficient and thus not platform dependent. However, for real world

applications, it is important to be able to include precompiled libraries. Thus, Truffle/C relies on a native function interface in this part [22]. Additionally, a preferable property is to obtain equal results between executed code compiled by different compilers and Truffle/C.

To address the issue of platform dependency Truffle/C has to determine the following properties in a platform specific way:

- The bit width of primitives and composites
- The alignment for composites
- Function calling convention

C only specifies minimum sizes and relationships between data types and not the exact size [23, C99 5.2.4.2]. However, the most useful behavior for Truffle/C is to use the same configuration as native compilers such as GCC. Since we used Linux AMD 64 as underlying platform, we followed the bit width and alignment of GCC for this platform. To guarantee interoperability between Truffle/C and native functions for other platforms, an additional configuration for the platform specific as mentioned above have to be provided.

The bit width of primitives is only a minor problem. C exposes these platform dependent bit width for integer types in *limits.h* and for float types in $float.h$ [23, C99 5.2.4.2]. Truffle/C could either parse these files and accordingly generate files during compile time of the Java project or read them during construction of the Java nodes during run-time. Using the same ranks guarantees the same precision of the results as other execution environments on the platform. Currently, the mapping between C data type and Java data type is hard-coded.

The alignment of composites and function calling convention is mainly related to the native function interface presented in the thesis of my colleague. While the alignment of composites have to be considered for every platform, the function calling convention is a concern of Graal, since the function interface is a capability of Graal [21, 22].

The Clang modification consists of 1,500 lines of code (LOC). Out of the 24,000 LOC of the Java side of Truffle/C, only 2,1% are platform dependent, namely 200 for the composite alignments, 100 for the bit sizes and decision which Java data type to use to represent C data types, and 200 LOC for varargs in function. While the interoperability to the native side is not yet complete, this number is especially small. In comparison, the Portable C Compiler contains 25% platform dependent code, although the compiler targets to keep the amount as small as possible [24]. In contrast to traditional compilers, Truffle/C does not have to produce machine code by itself but can use Graal to do so.

Chapter 11

Future Work

Completeness: The Truffle/C implementation already supports the basic language features of C, many "new" features of C99, as well as some of the GNU extensions. However, the implementation is still incapable of executing large real world programs. To be able to execute these programs, Truffle/C requires a higher degree of completeness. Hence not only ANSI C can be considered, but an implementation of the GNU extensions would be useful.

Security: Buffer overflows are one of the most serious security problems [13]. While static analyzes have to place pessimistic assumptions of whether a buffer overflow can occur or not, Truffle/C could (partly) check this during run-time depending on the input values. Truffle/C could put as much data as possible in the `Frame` where, e.g., a an invalid access to an array would cause a `NullPointerException` which can be handled gracefully. Also, Truffle/C could save additional run-time information through which checks could ensure data validity.

Portability: While Truffle/C is already portable to a high degree, there is still room for improvement. It would be preferable that Truffle/C could operate entirely without having to implement platform dependent logic. This could be achieved by extracting information from the header files as mentioned before, but also by extending Graal's native function interface by the necessary capabilities or bundling according libraries.

Performance Optimization: Truffle/C does not yet target runtime performance, but a clean design and no conceptional performance overhead. Truffle/C allows to experiment with optimizations that use profiling feedback that have not yet been explored for C because they are hard to implement in traditional compilers. Function inlining [10, 11] is an important optimization because it enhances the effect of other optimizations [8]. Truffle/C allows to implement heuristics that not only depend on static metrics but, e.g., depend on the call count of a function. Also, Truffle/C could assume that

compound objects or primitives are never referenced or passed to native functions, and put them into the `Frame` so Graal can better optimize such usages. Only if the assumption is invalidated, Truffle/C would have to write them into the `Memory`. To make these assumptions more reliable it would be useful to combine traditional static analyzes with the dynamic assumptions.

Chapter 12

Related Work

12.1 Optimizations Methods Similar to Truffle

Efficient interpreter implementations usually produce bytecodes. The cost of interpreting these bytecodes depends on on the branch prediction cost and accuracy [18].

Bytecode is usually hard and inflexible to replace once a compiler generates it. However, one technique is to use superoperators [31], to combine frequently executed groups of bytecode in one bytecode. Combining bytecodes reduces the number of bytecode reads, as well as pushes and pops in a stack-operating interpreter. In this approach, the interpreter can be generated on a per program basis, to adopt to the bytecodes present in the program. These replacements of bytecode are comparable to rewriting in Truffle. But in contrast to Truffle, this optimization is performed not at the language, but at the level of the bytecodes used to represent a program. Also, this method only increases the performance in interpreter mode, which the paper demonstrates for an ANSI C interpreter, while the Truffle nodes are subject to further optimization by Graal.

12.2 Platform Independence and Portability

One approach to reach the same degree of portability as Truffle/C is to use LLVM[1] with a front end such as Clang[2]. LLVM is a program analysis and transformation framework that retains its internal SSA form representation, enriched by high level information, throughout the program phases. It allows optimization and analysis at link-time, install time, runtime and between program runs. LLVM

[1] The LLVM Compiler Infrastructure: `http://llvm.org/`
[2] clang: a C language family frontend for LLVM: `http://clang.llvm.org/`

needs a front end such as Clang for C [26], which has to generate a representation of the program for LLVM. While the output of the front end is platform independent (but already contains the composite alignments), the platform dependent code is produced only before execution. In this respect, it is similar to Java and JVM or CLI, although its goals are different. It also allows run-time profiling, and to optimize and re-compile the native code. LLVM can also run in interpreter mode, when no native code generator is available [27]. The main difference in eyes of language developers of this approach compared to Truffle and Graal is that they can express the semantics of a language for Truffle in high level Java code, while for LLVM they have to express the semantics with LLVM bytecode. Since this language is close to RISC assembly, it takes more effort to map high level constructs for LLVM. Additionally, language developers for Graal do not need any knowledge about Graal itself, but only have to follow the Truffle API.

To the best of my knowledge, no C CLI language has been implemented so far. However, there exist efforts to implement a CLI C language [35]. For the JVM, some indirect approaches that base on translation to bytecode exist. One such approach is LLJVM[3], which uses a LLVM front end to compile a C program to LLVM IR, which it then translates to Jasmin assembly code and finally uses the assembly code to construct the JVM bytecode. NestedVM [9] either directly translates C source code or MIPS binaries to Java source code including a MIPS interpreter with the MIPS machine code or Java bytecode. For facilitating precompiled libraries, it assumes that they already have MIPS format to translate them to Java. When NestedVM directly generates JVM bytecodes it can use the goto specified by the Java Virtual Machine Specification [29, JVM 6.5]. Otherwise, a jump is the change of the pc variable which results in that the interpreter will fetch the instruction from another position in the array that NestedVM uses to store the instructions. A similar effort is Cibyl [25], that uses GCC to again translate the C program to MIPS. The presented tool then translates this MIPS code to Java bytecode. To obtain an executable class file, the project uses the Jasmin assembler. From the MIPS code, Cibyl also generates Java wrappers for system calls. In contrast to NestedVM, Cibyl targets Java J2ME devices.

There are several efforts to keep large portions of C compilers platform independent. One such example is the Portable C Compiler, which keeps 75% [24] of its code machine independent.

12.3 C Interpreter

One widespread C interpreter is CINT, which uses an incremental bytecode compiler [2]. It was developed for the object oriented data analysis framework ROOT. CINT[4] supports about 95% of

[3] LLJVM: `http://da.vidr.cc/projects/lljvm/`
[4] CINT: `http://root.cern.ch/drupal/content/cint`

ANSI C and 85% of C++ and is written in ANSI C itself [32]. The philosophy behind the language is to implement a useful script language combination of both C and C++.

Another prevalent C interpreter is Ch[5] which targets cross-plattform scripting. It supports the language features and libraries from the ISO C90 standard and most features of the ISO 99 standard. Its design goal is to provide a learning and testing environment. Ch provides useful features like array bounds checking but recommends to use a C compiler for final production. The website offers a free standard version, while a version with more features is commercial [12]. Both CINT and Ch can be used in a read-eval-print-loop.

There are several special purpose or hobby-project C interpreters. PicoC[6] is a C interpreter originally written for the on-board flight system of UAV's. CINLA[7] is a C interpreter specialized on linear algebra. There are also wrappers for compilers, that allow a read-eval-print-loop execution such as igcc[8] for GCC and ccons[9] for clang and LLVM. EiC[10] is another small C interpreter.

All these interpreters do not focus on performance but fill gaps on niches such as embedding C into applications, for learning purposes or hobby projects. Since the language is close to the machine level and compilers can easily map the concepts of C to machine code, it is hard to implement a competitive interpreter. Industrial quality interpreters often target dynamic languages, where compilers face the challenge of having to map the dynamic type system and operations to machine code. For dynamic languages, interpreter can thus easier gain a performance advantage. Another reason why up to today no industry-strength C interpreters exist is that when C came up interpreters were not fast enough to execute C. Nowadays, C compilers exist for almost every platform, diminishing the value of platform independent interpreters.

[5] Ch: `http://www.softintegration.com/`
[6] picoc. A very small C interpreter: `https://code.google.com/p/picoc/`
[7] CINLA: `http://www.pliatech.com/CInterpreter.aspx`
[8] Interactive GCC: `http://www.artificialworlds.net/wiki/IGCC/IGCC`
[9] ccons. Interactive Console for the C Programming Language: `https://code.google.com/p/ccons/`
[10] EiC: `http://sourceforge.net/p/eic/wiki/Home/`

Chapter 13

Conclusions

This thesis describes the Truffle/C project, which implements an interpreter based on Truffle for the execution of C programs. It shows that an implementation of C in Truffle is not only feasible, but also has a small and simple implementation compared to the traditional approaches such as with a compiler like GCC. While Truffle/C contains many nodes, all these nodes contain simple implementations for their execute methods, that are often not longer than a single line. Truffle/C can exploit the Truffle DSL, to automatically generate a major portion of the code. Truffle/C also shows that concepts not present in Java can be feasibly implemented, such as unsigned data types.

The evaluation shows that Truffle/C already has a high degree of completeness. The performance of the interpreter on benchmarks is not slower than 5.26 times compared to the execution of highly optimized GCC code. In contrast to dynamically implemented languages, Truffle/C has less specialization potential. However, there is still significant potential for performance improvements.

Acknowledgement

First and foremost, I would like to thank Hanspeter Mössenböck and Lukas Stadler for their constant and valuable feedback on my work. I particularly value the written feedback to my thesis by Lukas, which not only often made me smile but also contributed a lot to my ability of writing in a scientific style. I would also like to thank my colleagues in the Oracle project, especially Thomas Würthinger and Roland Schatz, for countless discussions and for all the feedback on my work.

I also want to express my special gratitude to my friend and colleague Matthias Grimmer for help and inspiration throughout my studies, and on the work on Truffle/C. We have always been a great team and after discussions on problems we always arrived at solutions that neither of us could come up alone. I wish you all the best for your PhD!

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

List of Figures

Listings

Bibliography

[1] Elf-64 object file format. version 1.5 draft 2. http://downloads.openwatcom.org/ftp/devel/docs/elf-64-gen.pdf.

[2] Cint incremental bytecode compiler. http://root.cern.ch/viewvc/branches/v5-34-00-patches/cint/doc/bytecode.txt, November 2013.

[3] Clang - features and goals. http://clang.llvm.org/features.html#gcccompat, November 2013.

[4] Clang 3.4 documentation. http://clang.llvm.org/docs/UsersManual.html, November 2013.

[5] Extensions to the C language family. http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html, November 2013.

[6] Gcc 4.9.0 online documentation. http://gcc.gnu.org/onlinedocs/gcc/, November 2013.

[7] Linux kernel coding style. https://www.kernel.org/doc/Documentation/CodingStyle, 2014.

[8] R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, pages 241–249, New York, NY, USA, 1988. ACM. URL: http://doi.acm.org/10.1145/53990.54014, doi:10.1145/53990.54014.

[9] Brian Alliet and Adam Megacz. Complete translation of unsafe native code to safe bytecode. In Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators, IVME '04, pages 32–41, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1059579.1059589, doi:10.1145/1059579.1059589.

[10] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, PLDI '89, pages 246–257, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/73141.74840, doi:10.1145/73141.74840.

[11] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-mei W. Hwu. Profile-guided automatic inline expansion for c programs. Softw. Pract. Exper., 22(5):349–369, May 1992. URL: http://dx.doi.org/10.1002/spe.4380220502, doi:10.1002/spe.4380220502.

[12] Harry H Cheng. Ch: Ac/c++ interpreter for script computing. CC PLUS PLUS USERS JOURNAL, 24(1):6, 2006.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, volume 2, pages 119–129 vol.2, 2000. doi:10.1109/DISCEX.2000.821514.

[14] Edsger W. Dijkstra. Letters to the editor: Go to statement considered harmful. Commun. ACM, 11(3):147–148, March 1968. URL: http://doi.acm.org/10.1145/362929.362947, doi:10.1145/362929.362947.

[15] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop, 2013.

[16] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages, VMIL '13, pages 1–10, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2542142.2542143, doi:10.1145/2542142.2542143.

[17] A.M. Erosa and L.J. Hendren. Taming control flow: a structured approach to eliminating goto statements. In Computer Languages, 1994., Proceedings of the 1994 International Conference on, pages 229–240, May 1994. doi:10.1109/ICCL.1994.288377.

[18] M Anton Ertl and David Gregg. The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism, 5:1–25, 2003.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional, 2005.

[20] James Gosling and Henry McGilton. The Java language environment, volume 2550. Sun Microsystems Computer Company, 1995.

[21] Matthias Grimmer. A runtime environment for the truffle/c vm. Master's thesis, Johannes Kepler University, November 2013.

[22] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An efficient native function interface for java. In Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2500828.2500832, doi:10.1145/2500828.2500832.

[23] ISO/IEC. Iso/iec 9899:tc3. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, November 2013.

[24] S. C. Johnson. A tour through the portable c compiler. In Unix Programmer's Manual, 7th Edition, 2B, Section 33, 1981.

[25] Simon Kågström, Håkan Grahn, and Lars Lundberg. Cibyl: An environment for language diversity on mobile devices. In Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE '07, pages 75–82, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1254810.1254821, doi:10.1145/1254810.1254821.

[26] Chris Lattner. Llvm and clang: Next generation compiler technology. In The BSD Conference, 2008.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[28] John Levine. Dynamic linking and loading. revision 2.3. http://www.iecc.com/linker/linker10.html, November 2013.

[29] Tim Lindholm and Frank Yellin. Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[30] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. The gnu c library reference manual for version 2.18. http://www.gnu.org/software/libc/manual/pdf/libc.pdf, November 2013.

[31] Todd A. Proebsting. Optimizing an ansi c interpreter with superoperators. In Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95, pages 322–332, New York, NY, USA, 1995. ACM. URL: http://doi.acm.org/10.1145/199448.199526, doi:10.1145/199448.199526.

[32] Fons Rademakers and Rene Brun. Root: An object-oriented data analysis framework. Linux J., 1998(51es), July 1998. URL: http://dl.acm.org/citation.cfm?id=327422.362112.

[33] Manuel Rigger. Python ast interpreter in java. bachelor thesis, Johannes Kepler University, 3 2012.

[34] Dennis M. Ritchie. The development of the c language. SIGPLAN Not., 28(3):201–208, March 1993. URL: http://doi.acm.org/10.1145/155360.155580, doi:10.1145/155360.155580.

[35] E. Rohou, A.C. Ornstein, and M. Cornero. Cli-based compilation flows for the c language. In Embedded Computer Systems (SAMOS), 2010 International Conference on, pages 162–169, July 2010. doi:10.1109/ICSAMOS.2010.5642069.

[36] Suman Saha, Julia Lawall, and Gilles Muller. An approach to improving the structure of error-handling code in the linux kernel. In Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, LCTES '11, pages 41–50, New York, NY, USA, 2011. ACM. URL: http://doi.acm.org/10.1145/1967677.1967684, doi:10.1145/1967677.1967684.

[37] Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas Reps. Coping with type casts in c. In Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7, pages 180–198, London, UK, UK, 1999. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=318773.318942.

[38] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and

scalar replacement for java. In Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14, pages 165:165–165:174, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2544137.2544157, doi:10.1145/2544137.2544157.

[39] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs, volume 30. ACM, 1995.

[40] Christian Wimmer and Stefan Brunthaler. Zippy on truffle: A fast and simple implementation of python. In Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, SPLASH '13, pages 17–18, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2508075.2514572, doi:10.1145/2508075.2514572.

[41] Andreas Wöß. Self-optimizing ast interpreter for javascript. Master's thesis, Johannes Kepler University, 2013.

[42] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! '13, pages 187–204, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2509578.2509581, doi:10.1145/2509578.2509581.

[43] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In Proceedings of the 8th Symposium on Dynamic Languages, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2384577.2384587, doi:10.1145/2384577.2384587.

Eidesstattliche Erklärung

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

Xiamen, am 2. April 2014

Manuel Rigger, BSc.

Curriculum Vitae

Personal Information

Name	**Rigger, Manuel**
Address	Husslstraße 27b, 6130 Schwaz, Austria
Telephone	+43 664 3478783
Email	manuel.rigger@chello.at

Professional Experience

2013–present	Remote Research Assistant, Institute for System Software, Linz, Austria.
2012–2013	Research Assistant, Institute for System Software, Linz, Austria.
2011	Internship as Eclipse RCP Plugin Developer at elexis-austria, Wels, Austria.
2010	Internship as Database Developer at Linz AG/Gas Wärme, Linz, Austria.

Education

2013–present	MSc in Chinese Philosophy, Xiamen University, China.
2012–present	MSc in Software Engineering, Johannes Kepler University, Austria.
2012	Exchange Semester in Computer Science, National Taiwan University, Taiwan.
2009–2012	BSc in Computer Science, Johannes Kepler University, Austria.

Others

Research Interests	Compilers, Interpreters, Virtual Machines, Compiler Optimization
Languages	German (native), English (CEFR: C1), Chinese (CEFR: B2)
Hobbies	Chinese Culture, International Relations, Traveling in China and Southeast Asia