UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Tõnis Pool

Generic Reloading for Languages Based on the Truffle Framework

Master's Thesis (30 ECTS)

Supervisor: Allan Raundahl Gregersen, PhD
Supervisor: Vesal Vojdani, PhD

TARTU 2016

Generic Reloading for Languages Based on the Truffle Framework

Abstract: Reloading running programs is a well-researched and increasingly popular feature of programming language implementations. There are plenty of proposed solutions for various existing programming languages, but typically the solutions target a specific language and are not reusable. In this thesis, we explored how the Truffle language implementation framework could aid language creators in adding reloading capabilities to their languages. We created a reusable reloading core that different Truffle-based languages can hook into to support dynamic updates with minimum amount of effort on their part. We demonstrate how the Truffle implementations of Python, Ruby and JavaScript can be made reloadable with the developed solution. With Truffle's just-in-time compiler enabled, our solution incurs close to zero overhead on steady-state performance. This approach significantly reduces the effort required to add dynamic update support for existing and future languages.

Keywords: Language implementation, Truffle, Graal, dynamic software updates

CERCS: P170, Computer science, numerical analysis, systems, control

Käitusaegne programmi uuendamine Truffle raamistiku keeltele

Lühikokkuvõte: Programmide käitusaegset uuendamist on põhjalikult uuritud ning selle kasutamine programmeerimiskeelte implementatsioonides kogub hoogu. Senised pakutud lahendused programmide käitusaegse uuendamise osas on rakendatavad ainult konkreetsetele keeltele ja ei ole taaskasutatavad. Käesolevas lõputöös on uuritud seda, kuidas Truffle-nimeline programmeerimiskeelte loomise raamistik suudaks aidata keelte loojatel lisada käitusaegse uuendamise tuge. Autor on loonud taaskasutatava dünaamilise uuendamise lahenduse, mida erinevad Truffle raamistikus loodud keeled saavad kasutada selleks, et vähese vaevaga toetada käitusaegseid uuendusi. Antud lahendusega on võimalik uuendatavaks teha Pythoni, Ruby ja JavasScripti Truffle implementatsioone. Väljatöötatud lahendusel on peaaegu olematu mõju keele tippvõimsusele, kui on sisse lülitatud Truffle täppisajastusega (JIT) kompilaator. See lahendus teeb käitusaegse uuendamise toe lisamise uutele ja tulevastele keeltele märkimisväärselt lihtsamaks.

Võtmesõnad: Programmeerimiskeele implementeerimine, Truffle, Graal, dünaamiline uuendamine

CERCS: P170, Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

Introduction

Overview

Truffle [1] is a novel programming language implementation framework. It is developed by Oracle Labs in collaboration with the Institute for System Software at the Johannes Kepler University Linz. The main goals are making implementing new languages easier, while achieving competitive performance compared to existing well-tuned language runtimes. Language implementers have to implement an AST interpreter, within the Truffle framework, for their new language and the framework handles the rest.

Truffle seeks to enable generic solutions to common but complex problems. Examples include the ability to add high performance debugger support to a language [2] or the Truffle Object Storage Model (OSM) for implementing types and objects [3]. Truffle has even opened up the possibility for high performance cross language interoperability, where Truffle can inline and optimize function calls across language barriers [4]. One problem they have yet to provide a solution for is dynamically updating running applications.

After a developer makes a change in the source code, they typically need to restart the running program to see those changes having an effect. This is true for many popular languages such as Java and C# without relying on some external tools. In this thesis we refer to *reloading*, also known as dynamic code evolution or dynamic system update, as seeing the effects of source code changes immediately in a running application.

Some dynamic languages such as Erlang, Ruby and Python have reloading support built in, but developers must take explicit action, for example call a specific method, to achieve the desired behavior. It would be more convenient if the running program was automatically updated whenever source code changes are detected. This way developers could just concentrate on getting the program's behavior correct and iterate faster.

Importance of Reloading

Programming is a complex cognitive endeavor that puts a strain on both the short-term (working) and long-term memory. Good working memory has been shown to predict a

better ability to learn programming [5]. However, short-term memory is, as its name suggests, fleeting. Research has shown a significant decay in working memory over time spans as short as 15 seconds [6].

This means that every second counts when working on problems that require keeping lots of details in working memory. At the same time programmers are constantly bombarded with interruptions. A Java developer productivity survey carried out in 2012 found that developers spend only about one third of their work week on actually writing code [7]. The rest of the time is spent on various related activities such as debugging, compiling, deploying, communicating, etc.

Given the fleeting nature of short-term memory and the general overhead associated with programming, it becomes increasingly important to have coding feedback cycles shorter than the short-term memory decay period. If source code changes are immediately visible in the running application it decreases the strain on working memory and should increase the likelihood of developers being able to complete the task without losing the state of their working memory.

Developer productivity is only one aspect why reloading matters. Another well known problem is updating high availability applications to include new critical bug fixes without causing any downtime. Erlang is a programming language with a managed runtime that is often used in 99.999% uptime guaranteed systems [8]. To help in meeting these high expectations, Erlang has reloading built into the runtime so that modules could evolve over time without causing any downtime [9, Chapter 13]. For such applications reloading must not affect the steady-state performance nor bring about unexpected side effects that can compromise the stability of the application.

The desire of developers to reload code changes can also be validated by the plethora of different tools and solutions for various languages and platforms. JRebel [10] is a successful enterprise Java reloading system that is actively used by tens of thousands of developers. Many popular web application frameworks have built custom solutions to avoid restarting the development server. Examples include:

1. **Django**[1] — A web application framework written in Python. The provided development server will automatically reload Python code for each request.
2. **Ruby on Rails**[2] — A web application framework written in Ruby. When running in development mode Rails also supports reloading changed code.
3. **Grails**[3] — A Groovy-based web application framework. Can be configured to reload classes and view files.

[1] https://www.djangoproject.com/
[2] http://rubyonrails.org/
[3] https://grails.org/

Degrees of Reloading

Reloading is not a binary property that languages either support or not. Instead, we have to look at the degree of code changes that the language enables us to apply at runtime. For example, Java normally only supports changing method bodies in a running application via HotSwap [11], but with external tools, many more modifications are possible.

Possible supported runtime changes can be loosely categorized as:

1. **Function level changes**
    - Changing function bodies
    - Changing function signatures (arguments, return type, visibility)
    - Adding/removing functions
2. **Changes in code organization/structure**
    - Changing visibility of classes/modules
    - Changing structure of classes/modules (fields and methods)
    - Adding/removing classes/modules

The second option depends highly on the semantics of the language. Some languages do not even have notions of classes or modules, so only changes listed in Category 1 are more universally applicable. Though, to the best of our knowledge, all programming languages exhibit a notion of grouping code into callable units that can be classified as functions. Having such groups of code, it is natural to modify one of them or to add new ones to change the program's behavior.

Research Questions

The main research question this thesis looks to answer is *if and how the Truffle framework can help language implementers in adding reloading to their languages.* More specifically the questions can be listed as:

1. Can the Truffle framework help language implementers achieve reloading in their languages?
2. How much effort, if any, is required from the language implementer to add code reloading support?
3. What language constructs make supporting generic reloading harder?
4. How does adding reloading support affect the steady-state performance of a language?

Contributions

Dynamically updating running programs is a subject that has been studied for many decades already [12]. There have been many proposed solutions for specific programming languages that do not natively provide reloading, such as [13] or [14]. Our focus was on trying to produce a generic reloading solution with maximum reuse and minimal language specific work. Of course, such a tool requires a common building block for representing different languages by a shared model, which Truffle provides.

The main contribution of this thesis is a series of prototypes that show various ways how language neutral reloading can be supported in the Truffle framework:

- The initial prototype contributes a simple approach to reloading languages by using the Truffle Instrumentation API. That approach can be used to reload languages that are already built with dynamic code changes in mind. The solution works well in that it can leverage a lot of the existing tools in the Truffle framework and requires no language specific code. Unfortunately the approach is fairly limited and cannot be used to reload languages with more complicated ways of structuring code, besides grouping it into functions.
- The next prototype contributes a more generic approach that can be used to reload languages that have not been built with dynamic updates in mind. The prototype supports languages with different code structuring semantics, such as classes and modules. Unfortunately at this point we already lose absolute generality and require some input from the language implementer. Though the solution works in setups where Truffle is running on a default Java Virtual Machine, it does not work with the GraalVM (introduced in Section 1.3).
- Finally we improve the last prototype so that it also works on the GraalVM and has minimal overhead to the steady-state performance of the application. We also describe how the needed changes to the Truffle framework can be applied automatically via a Java agent.

The main results of this thesis were also summed up as a research paper, written together with the supervisors, which has been accepted for publication as part of the 11th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS '16).[4]

Thesis Structure

The thesis continues in Chapter 1 by giving an introductory overview of the components and internals of the Truffle framework. Key aspects and techniques are explained, with

[4] http://2016.ecoop.org/track/ICOOOLPS-2016

focus on the parts relevant to achieving the goals of this thesis. The goal of the chapter is to help the reader understand the coming challenges and better follow the subsequent chapters. Readers who are already familiar with the internals of the Truffle framework, should be able to skip Chapter 1 without compromising their ability to follow subsequent discussions.

Chapter 2 describes the first attempts at reloading the Truffle framework reference language implementation, called Simple Language. Various obstacles met along the way are explained together with things that have to be taken into consideration to avoid problems down the road.

Chapter 3 builds upon the learnings from the initial prototype to work towards a more generic reloading solution for Truffle. It is followed by Chapter 4 that discusses GraalVM specific challenges, implementation details and minimizing the runtime overhead. Finally Chapter 5 evaluates the proposed techniques by applying them for four concrete language implementations: Simple Language, ZipPy, JRuby+Truffle and Graal.js.

Chapter 1

Truffle Framework

There are many ways to implement a programming language, but loosely they fit into the following categories [15, Chapter 1]:

1. **Interpretation** — Write an interpreter that follows the instructions in source code and carries out the corresponding actions.
2. **Compilation** — Write a compiler that translates the source code into some other language that can either execute directly on the hardware or be in turn subject to interpretation or compilation.

Compilation generally should have better performance as there are usually inherent bottlenecks to interpreters. Interpreters have to read instructions one by one and then decide what is the right action to take. This causes unnecessary overhead, where instead of just executing the programmers instructions there's always a translation step in between during runtime. Languages that compile directly to machine code do not have that overhead as the program consists of instructions the CPU can execute directly.

Whichever strategy is chosen, there's a common predecessor to both compilation and interpretation, which is parsing the source code. A common way is to parse the source code into an Abstract Syntax Tree (AST). The AST shows the structure of the instructions written in source code. This structure can then be translated into another language — compiled — or directly interpreted.

Despite the inherent problems, the Truffle framework has chosen the interpreter route with some clever tricks to overcome the shortcomings. The general idea is that a programmer wishing to create her own language implements an AST interpreter within the Truffle framework. Truffle then overcomes the overhead usually associated with pure interpretation of a language. This is a win-win situation as usually writing an AST interpreter is easier than writing a compiler or a managed runtime [15, Chapter 1]. The implementer gets the language up and running faster and Truffle ensures that the performance of the language is comparable to other compiled languages.

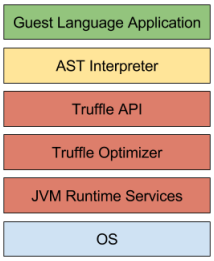

Figure 1.1: Architecture of the Truffle framework [16]

1.1. General Architecture

Truffle is implemented in Java and builds on top of the standard Java Virtual Machine (JVM). The overall architecture is shown on Figure 1.1. It reuses existing runtime services in the JVM, such as garbage collection. The Truffle Optimizer layer represents partial evaluation using the Graal compiler which will be discussed further in Section 1.3. The overall goal of the Truffle framework, of course, is to be able to execute applications written in a *guest* language. Within truffle, the *guest* language runtime is implemented in Java using the Truffle APIs that then runs on top of the JVM.

The main entry point for everyone who wants to implement a Truffle-based language is the `TruffleLanguage` class. Implementers are expected to extend it and add the `@Registration` annotation that declares the name, version and source code MIME type for the language. Truffle uses this information to generate metadata associated with the language project. The main abstraction used for running the languages is the `PolyglotEngine` class that acts as a gateway into the world of Truffle languages.

Evaluation is triggered by calling the `PolyglotEngine#eval` method, which locates the language implementation and invokes the `TruffleLanguage#parse` method. This in turn transforms the source code into an executable AST. The suitable `TruffleLanguage` implementation is chosen based on the MIME type from the `@Registration` annotation and the MIME type of the source code file.

Each evaluation of a `TruffleLanguage` is associated with a global language context that Truffle creates by calling the `TruffleLanguage#createContext` method. Typically the context is where the all the global information about the running program is kept, such as global variables and method definitions. Running evaluations are expected to access the current context by using the `#createFindContextNode` and `#findContext` methods.

1.2. Self-Optimizing AST Interpreters

In the Truffle framework the AST itself is the interpreter. It consists of a tree of Java objects that have an `#execute` method, which carries out the action the AST node represents. All AST node classes have to inherit from a single `Node` class and all callable function definitions have to inherit from a `RootNode` class.

In order to call a function, a `RootCallTarget` has to be obtained from the Truffle framework. After finishing parsing a function definition, the resulting `RootCallTarget` is stored into the correct location depending on the language semantics. It could be a global namespace or part of a class definition, etc. `RootCallTarget` extends the `CallTarget` interface, which defines the `Object call(Object... arguments)` method that can be invoked to execute the underlying function. Each `RootCallTarget` corresponds to a tree of nodes that carry out the actions of the function.

Interpreters are self-optimizing, meaning they change during execution by replacing one node with another. Optimizations are triggered by reacting to actual runtime values of variables and behavior of the program. As a simple example, imagine an abstract `AddNode` that represents a `+` operator in source code. Depending on the constraints of the language under evaluation, addition can mean several different operations. The code could be adding text, integers or floating point numbers together.

Handling all the possible cases in a single node is not optimal because of the necessary type checks and potential variable boxing. Instead, the first time the `AddNode` is executed, the generic node, which handles all possible cases, can be replaced with a specialized one. If the variables are both integers, then `AddNode` becomes an `IntegerAddNode` instead. `IntegerAddNode` only knows how to add integers and ignores other possibilities, thus its implementation is more efficient.

Of course as soon as the node is specialized, an implicit assumption is created that this specialization will always hold. The assumption might not be valid for the next invocation of `IntegerAddNode` when it encounters text for example. If an assumption fails a node will be replaced with a *more generic* one. The new node does not necessary yet have to handle all possible cases, just the ones encountered so far [17]. The exact method how the *more generic* node is chosen is explained in Section 1.4.

1.2.1. Examples of Optimizations

There are many possible optimizations the AST could perform, such as:

1. **Operation Specialization** — In dynamic languages the performed operation can vary greatly depending on the types of operands. As shown with the `AddNode`,

simple `+` operator can mean integer or floating point addition, text concatenation or a method dispatch depending on the language semantics. Thus the *AddNode* AST node can be specialized to different forms.

This specialization is not limited to dynamic languages. Say a programmer has specified an addition on 2 double variables in a statically typed language. But in runtime it turns out that they are both actually integers and the addition does not overflow. Then doing an integer operation is beneficial as on all current architectures integer operations can be performed faster than floating point computations [17].

2. **Polymorphic Inline Caches (PICs)** - Languages with dynamic dispatch have to look up the call target of the function call before executing it, which can be costly. It has been observed, however, that function call targets change seldom for a specific call site — the place where the `CallTarget` is invoked. Call sites can be divided into 3 classes [18]:

   (a) *Monomorphic* — Only one target.

   (b) *Polymorphic* — A few targets.

   (c) *Megamorphic* — Arbitrarily many targets.

   Polymorphic Inline Caches is a well known technique for caching and linking up to a small number of call targets. When a function is called at a call site the cached list of call targets is iterated for a type match. Truffle AST rewriting can easily encompass PCIs by creating a chain of cached nodes. For every new entry in the cache, a new node is added to the tree. When the chain reaches a certain predefined length, the whole chain replaces itself with one node responsible for handling the fully megamorphic case [16].

3. **Dispatch Chains** - Dispatch chains are a generalization of PICs that can be used to optimize reflective method invocation. Essentially dispatch chains build layers of caches. For reflective method invocation the first layer is caching on the method name and the second layer is a classic PIC for the resolved method. Similar technique can be used to optimize reflective field and global access operations.

   Again the AST can be modified with new nodes as new method names are encountered during runtime and new cache nodes added. Dispatch chains can be used to eliminated the overhead of reflective operations [19].

1.2.2. Method Inlining

Operation Specialization can be rendered useless, when a method is always called with different types of arguments. Listing 1.1 shows a simple Python function that is called in an endless loop with both integers and text as arguments. In such a case the `#add` function would be specialized into a form generic enough to handle both integer addition

```
def add(first, second):
  return first + second

while True:
  add(1, 1)
  add("Hello", " world")
```

Listing 1.1: Calling methods with different types of arguments

and text concatenation. It would be more efficient if the first call site would be specialized for integers and the second for text.

Truffle handles this by doing AST level inlining. When a particular function call site is executed a predefined number of times then Truffle will clone the function `CallTarget` AST subtree into the calling function. The inlined subtree will be cloned as if the method has not been called before; all of the nodes are back to the uninitialized state. On the first invocation it can then specialize itself according to the argument types present in a specific call site.

Naturally in a dynamic environment the inlined `CallTarget` might change for a given call site. In this case it is up to the language implementers to guard against it. The general approach is to create an assumption that the `CallTarget` for a given call site does not change. Before calling the target, the assumption should be checked. When the call target is changed then the assumption is invalidated causing the next check to fail. On assumption failure the node replaces itself with one that looks up the `CallTarget` once more before making the call. Truffle assumptions will be discussed in more detail in Section 1.4.

1.3. Partial Evaluation

Replacing the AST nodes with more specialized ones makes sure that the performed actions are optimal, but still leaves the overall overhead of traversing an AST. As mentioned in the Introduction, Truffle claims to achieve similar performance to compiled languages. Partial evaluation is the process of taking a computation π with $m + n$ variables $c1, \cdots, cm$, $r1, \cdots, rn$. Substituting known values of $c1, \cdots, cm$ into π resulting in a computation with n variables $(r1, \cdots, rn)$ [20]. The known variables are substituted where possible, computations carried out and unknown variables left into the remaining computation.

Truffle leverages this to just-in-time (JIT) compile parts of the AST. It counts the number of invocations of a `CallTarget` and resets the counter in the event of a node replacement. When the number of invocations on a stable tree exceeds a threshold, Truffle assume that

every node of the tree is constant and therefore many values in the tree can also be considered as constants. The tree is compiled into machine code with dynamic dispatch turned into direct calls, thus removing one of the main performance penalties of interpreters [16].

Partial evaluation and compilation is performed by the Graal compiler, which integrates with the Truffle framework, but will not be examined in much detail. Graal is a new JIT compiler for Java bytecode written in Java. It plugs into a the standard HotSpot Java Virtual Machine (JVM) through some newly developed interfaces. Ongoing effort as part of the JDK Enhancement Proposal (JEP) 243 is trying to standardize these interfaces so that Graal could be used with a standard JVM in the future [21].

The result is that at the time of writing Truffle has two runtime modes:

1. **Interpreted** - When running on a standard JVM (without Graal) the AST nodes will only be interpreted. Specialisations and optimizations via node rewriting still occur, but nothing is compiled to machine code, thus the overhead of the interpreter remains.

2. **GraalVM** - When running on GraalVM (that is a JVM with necessary patches to make the Graal compiler work) then Truffle can schedule parts of the AST to be compiled to machine code. Only when running with Graal can languages implemented with the Truffle framework achieve the promised performance. As after Graal JIT compilation the overhead of interpreters is removed altogether.

1.4. Truffle DSL

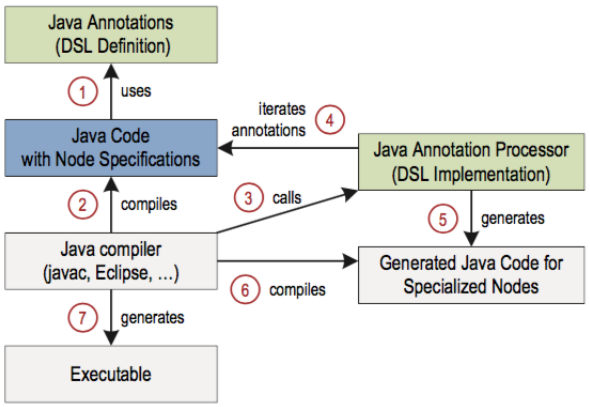

Figure 1.2: Truffle DSL annotation processing pipeline [22]

All the node specialisations have to be written by the language implementer. To make it easier, Truffle offers a Java annotations based domain specific language (DSL). Truffle DSL is declarative, meaning annotations are used to declare the intent of the optimisations. The framework takes care of generating necessary code to express those intents.

```
@NodeInfo(shortName = "+")
@NodeChildren({@NodeChild("leftNode"), @NodeChild("rightNode")})
public abstract class AddNode extends ExpressionNode {

    public AddNode(SourceSection src) {
        super(src);
    }

    @Specialization(rewriteOn = ArithmeticException.class)
    protected long add(long left, long right) {
        return ExactMath.addExact(left, right);
    }

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

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

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

Listing 1.2: Hypothetical addition node using Truffle DSL

The DSL is implemented as a standard Java annotation processor to generate additional source code during compilation [22]. Figure 1.2 shows the entire DSL processing pipeline.

As an example let's look at a hypothetical addition node that can add arbitrary precision integer numbers and text together, shown in Listing 1.2. Starting from the top, the `@NodeChildren(...)` annotation says that this AST node has 2 children called "leftNode" and "rightNode". The first `add` method takes in two Java `long` type variables and adds them using `ExactMath.addExact` which throws an `ArithmeticException` when the addition overflows. In that case the addition node will be rewritten to a more generic one that uses `BigInteger` data type that has arbitrarily large precision. This is achieved by the `@Specialization` annotation on the method. The annotation tells the Truffle system to rewrite this node to a more generic one when that exception is thrown.

Every Truffle language has to provide a class that has the Truffle `TypeSystem` annotation. The `TypeSystem` annotation contains an ordered list of types in which every type precedes its super types. This means that the types that are more concrete come first

in the list. The Truffle DSL uses this information to replace a node with a more general one when the current specialization fails. A single guest language type could be modeled with various Java level types, depending on what is actually needed. Truffle allows to define implicit casts between different Java types.

In Listing 1.2, a single guest language type `Number` is simulated with either a Java `long` or a `BigInteger`. Generally using a `long` is more performant, but in case precision larger than 64 bits is needed, it automatically falls back on using `BigInteger`s.

Specialisations are guarded by various *guards* provided by the DSL in the form of attributes on the `@Specialization` annotation [22]:

1. *Type guards* - In order for the specialization to hold the type of the arguments must match those of the method parameters.

2. *Method guards* - The `guards` attribute defines the set of methods that need to return `true` in order for the specialization to hold. An example of this is the `isString(left, right)` value in Listing 1.2.

3. *Event guards* - The `rewriteOn` attribute says to trigger a re-specialization when a specific exception occurs.

4. *Assumption guards* - The `assumptions` attribute defines a set of expressions whose return types must be `com.oracle.truffle.api.Assumption`. When any of the returned `Assumption` types are invalidated during runtime the node is triggered for re-specialization.

An instance of the `Assumption` class can be obtained from the Truffle framework by invoking the `TruffleRuntime#createAssumption` function. When running without the Graal compiler (as described in 1.3) assumptions are essentially just wrapped boolean flags that throw exceptions when invalidated. On GraalVM, when an assumption does not hold anymore, it also takes care of *invalidating* all machine code generated based on it. Invalidating machine code, also known as deoptimization, causes the execution to switch back to interpreting the AST [23]. Thus execution never continues under faulty assumptions, regardless whether the method is interpreted or running in machine code.

1.5. Discussion

For the purposes of this thesis Truffle already provides some necessary tools. We could leverage the `TruffleLanguage#parse` method to re-parse the source code if needed. The global context associated with the executing `TruffleLanguage` should be reused when `#parse` is invoked again at a later stage, thanks to the way it is accessed via the `#findContext` method. That should mean that all the state (global objects and stored values) remain the same even after re-parsing.

On GraalVM the existing `Assumption` mechanism can be reused to invalidate any code executing the previous version of the source code. This theoretically should point towards the possibility of having a very low overhead to the steady state performance of the application, as everything that is related to reloading could be guarded by `Assumption`s.

As every method call in Truffle can result in cloning and inlining the `RootCallTarget` nodes, then language implementations already have to be prepared that the execution of a particular method restarts from an uninitialized AST state at any moment and build their interpreters accordingly. This works to our advantage, as after reloading all the method ASTs would be in an uninitialized state.

To sum up, Truffle is already built with a highly dynamic execution environment in mind, where the interpreted AST is expected to change at any moment due to a change in program behavior. Reloading can be seen as a node specialization to its newest version. When source code changes then the whole AST, or parts of it if possible, should be triggered to update. As an additional requirement, reloading should work correctly for both the interpreted AST execution and compiled mode.

The next chapter will cover some of the tools Truffle itself provides to extend the framework with new capabilities and evaluate if and how these could be reused to help language implementers add reloading capabilities.

Chapter 2

Reloading Simple Language

It is a good engineering practice to evaluate existing solutions before coming up with one's own. Following this guideline we started our quest of reloading by first investigating how existing tools in the Truffle framework could be reused to help language implementers add reloading to their languages. In this chapter we'll outline the initial prototype for reloading the Truffle Simple Language using the Truffle Instrumentation API. Key requirements and techniques that were used to achieve reloading are outlined and described, together with many shortcomings.

2.1. Simple Language Overview

Simple Language (SL) is a language created to demonstrate and showcase features of the Truffle framework. Many new additions to the framework will first get implemented there, so that other language implementers would have a place to learn. As the name hints, it is a relatively simple programming language. SL is a dynamically strongly typed language with C like syntax. Functions are first class citizens. SL supports arbitrary precision integer numbers, booleans, Unicode characters, function types and the null type [24].

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

function apply(f, sum, i) {
  return f(sum, i);
}

function main() {
  i = 0;
  sum = 0;
  while (i <= 10000) {
    sum = apply(add, sum, i);
    i = i + 1;
  }
}
```

Listing 2.1: SL has first class functions

There are only a handful of built-in library functions such as `println`, `readln` and `nanoTime`. Objects resemble JavaScript objects and just contain name/value pairs. Listing 2.1 showcases the simple C like syntax and how it handles functions as first class citizens.

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

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

Listing 2.2: SL function redefinition

One interesting and relevant language aspect for this thesis is that SL also supports function redefinition via the built-in `#defineFunction`. Listing 2.2 shows how the function `#dynamic` does not even exist when the program starts and is defined on line 4. Later on it is redefined to subtract instead of adding. This shows that SL is already built with the consideration that function definitions might change at any point during execution.

2.2. Requirements for Reloading

To proceed with reloading SL, the required high level steps can be grouped as:

1. **Detecting source code changes** - In order to reload anything the first step is detecting source code changes to trigger a reload. Luckily Truffle associates a `SourceSection` with every `RootNode`. A `SourceSection` is simply a contiguous section of text within program code. Every `SourceSection` has a reference to a `Source` that represents the whole original source code file under evaluation. A `RootNode` is not required to have a `SourceSection` attached to it, but they are needed also by other tools, such as the debugger support. Thus language implementers are likely to add correct `SourceSections` to their `RootNode` implementations and indeed this is the case for all Truffle based languages that were investigated.

   There are many implementations of the abstract `Source` class in Truffle, but this thesis focuses on the `FileSource` implementation, as it is the one used when evaluation of a source code file is triggered. From the `FileSource` one can obtain access to the underlying `java.io.File` object. Thus detecting source changes can be naively implemented by checking the `lastModified` value of the `java.io.File` as it will be updated whenever a developer saves the file after making changes to it.

2. **Source code parsing** - After detecting a source code change the next step is to parse the code into an updated AST. The only official API for invoking parsing is

the `TruffleLanguage#parse` method that Truffle invokes when it starts evaluating a `Source`. Parsing should not execute any user code, just create the tree of nodes to represent the source.

3. **Updating the execution tree** - With the updated AST in place, the next step is making sure that the running program actually starts to use it. As every method in Truffle lives behind a `CallTarget` one approach would be to make sure that `CallTarget`s are invoked on their newest version.

   This is easy in SL as all functions are kept in a global function registry. During parsing all `CallTarget`s are registered with their function names. If the function has been registered before then the existing `SLFunction` node is modified to use the new `CallTarget`. Function redefinition in SL relies on this behavior and it can be reused for general reloading.

   However this is a very implementation specific approach and not applicable in general. A more general approach will be discussed in the next chapter.

2.3. Truffle Instrumentation API

To fulfill the requirements listed in Section 2.2, one of the first challenges is coming up with a way of checking the current `Source` for changes during AST evaluation. In order to achieve reloading one needs to inject some code into the running AST. This is where the *Truffle Instrumentation API* was used to reload SL. It is specifically designed to offer external tools access to the execution state of Truffle AST interpreters. Truffle Instrumentation API can be used to implement various tools such as debuggers, code coverage trackers and even profilers [25, 26].

The API can be divided into two logical parts; one that the language implementer has to implement in order to make the language instrumentable and the second part that the tool developer uses to gain access to the executing AST state. The general approach of the instrumentation API is to inject synthetic wrapper nodes into the AST that normally just delegate to the nodes they wrap, but if needed also perform additional actions [25].

2.3.1. Probing

In practice the language implementer marks some AST nodes as instrumentable by making `TruffleLanguage#isInstrumentable(Node node)` return `true`. If a node is marked as instrumentable then Truffle creates a wrapper node for it by invoking the `TruffleLanguage#createWrapperNode(Node node)` method. That method has to return an implementation of a `WrapperNode` interface for the given node. The language implementer has to provide the correct implementation of `WrapperNode` for each instru-

```
public class ReloadingFunctionInvokeASTProber implements ASTProber {
  @Override
  public void probeAST(Instrumenter instrumenter, RootNode startNode) {
    startNode.accept(new NodeVisitor() {
      @Override
      public boolean visit(Node node) {
        if (node instanceof SLInvokeNode) {
          final Probe probe = instrumenter.probe(node);
          probe.tagAs(StandardSyntaxTag.CALL, null);
        }
        return true; // visit all nodes
      }
    });
  }
}
```

Listing 2.3: Simplified custom ASTProber for SL

mentable node.

The wrapper nodes are created by walking the AST of a function when a `CallTarget` is created for a `RootNode`. The language implementer has to provide an implementation of an `ASTProber` to achieve this. The job of an `ASTProber` is to walk the tree of nodes and create `Probe`s for any nodes that could be of interest to tool developers.

A `Probe` is a binding between a location in the source code in an executing Truffle AST and a dynamic collection of listeners that receive event notifications. Probes can be also tagged with a `SyntaxTag` to mark that this AST location represents a specific abstract action such as an assignment or a function call. This enables tool developers to operate without specific knowledge about the AST nodes; they can simply state that they are interested in being notified when a node representing an assignment statement is executed. If the probes are properly tagged by the language implementer the tool developer will get exactly what is requested.

Listing 2.3 shows a simplified `ASTProber` that was used to probe the function invocation nodes of SL and tag them as function calls. A custom `ASTProber` was needed as the standard one provided by SL itself only tagged generic statements and assignments, but no function calls.

2.3.2. Receiving Events

After the AST is populated with `Probe`s, tool developers can attach two types of listeners to them to receive events — `SimpleInstrumentListener` and `StandardInstrumentListener`. Both of them contain methods that will be called before the wrapped node executes or

after it returns (exceptionally or with a value).

The difference between them is that the `SimpleInstrumentListener` only provides access to the currently active probe, but `StandardInstrumentListener` additionally provides access to the wrapped node and the current execution frame. Tool developers can implement these interfaces and carry out the necessary actions when the callbacks are invoked.

2.4. Initial Prototype

The initial prototype used the Truffle Instrumentation API to register a custom `ASTProber` that tagged all function call nodes in the tree. For each probe tagged with the `CALL` syntax tag an implementation of `SimpleInstrumentListener` is registered that enables us to have a callback before every function call.

That callback obtains a reference to the underlying `Source` as described in Section 2.2. As the `java.io.File` object is not directly accessible due to visibility limits, Java reflection is used to reflectively obtain the underlying file. Then in order to achieve reloading before every SL function call:

1. Check the `java.io.File` time stamp. If it has changed goto 2. Otherwise proceed with the function call as normal.
2. Clear any necessary caches using Java reflection.
3. Obtain a reference to the current `TruffleLanguage` object and invoke the `#parse` method.
4. During parsing all function definitions are (re)registered to the function registry. New functions are added and existing functions get a new `CallTarget`, thus on next invocation the new method definition will be used.

As the Truffle framework was not designed with the reloading purpose in mind, many of the needed parts of the system are not accessible by default. Luckily Java reflection enables us to bypass these restrictions and allows carrying out the needed operations at runtime. Similarly as mentioned in step 2 there are several caches along the way that need to be cleared in order to correctly load and re-parse the `Source`. Otherwise the new AST would be re-parsed based on cached source code or not at all.

2.5. Discussion

The approach outlined in this chapter achieves the desired behavior for the Truffle Simple Language. Whenever source code is changed the running evaluation of that code is updated to reflect those changes. Unfortunately there are several steps in this prototype

that are Simple Language specific and therefore do not necessarily apply to other Truffle framework based languages.

The first problem is that this solution required a correct `ASTProber` to tag function calls. As was seen then already SL did not provide such an `ASTProber`; a language implementation specific one had to be created. This could increase the amount of work that needs to be done to add reloading to other languages.

Even if the language provides an `ASTProber` that tags function calls, there is no guarantee about the location of the tag; it could be placed before or after the `CallTarget` invocation. The exact location of the tagged AST node can usually be seen as an implementation detail, but for reloading it's an important aspect. If the tag is set before execution starts the `CallTarget` can be swapped with a newer one, but when the `CallTarget` is already executing, changing it requires knowledge about the specific language implementation and its AST structure.

A bigger problem is the fact that thanks to the way SL was built we got reloading requirement #3 (updating the execution tree) essentially for free. We did not need to do anything after re-parsing as the function registry already contained updated and added `CallTarget`s. But that is a SL implementation detail that does not hold for other languages. In case other Truffle based languages follow the implementation pattern of SL, with regards to a global function registry that supports redefinition, then the approach outlined in this chapter can be used to modify and add functions. But in general the initial prototype is not a generic solution that language implementers can use to add reloading capabilities. Despite this, it did highlight many of the upcoming problems and give better insight into the Truffle framework.

The above problems left us no other choice, but to continue with looking for an alternative reloading strategy; one that would require as little language specific knowledge as possible and would universally address the problem of updating the execution tree.

Chapter 3

TruffleReloader

The previous chapter left us looking for a better reloading strategy. The approach using the Truffle Instrumentation API was good in that it provided a clear injection point into the framework. The downside was that it did not provide a general way of updating the running AST. Having been inspired by the Instrumentation API we formulated a collection of tricks that together form the tool called TruffleReloader. The main components/approaches of TruffleReloader are: *Proxy CallTarget*, *CallTarget Identity* and *Partial AST Replay*.

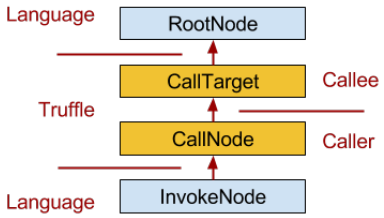

Figure 3.1: Default method invocation in Truffle [24]

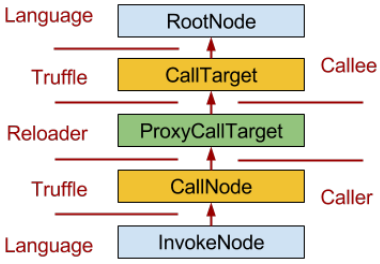

Figure 3.2: Truffle method invocation with Proxy CallTarget

3.1. Proxy CallTarget

Figure 3.1 shows the default control flow of a method invocation in Truffle. A call site, in the implemented programming language, invokes a call node that was obtained from the Truffle runtime. The call node then in turn invokes the corresponding `CallTarget`, which will start executing the `RootNode` of that `CallTarget`. This indirection through the Truffle runtime for every function call is needed to detect and trigger optimisations such as inlining or scheduling for compilation. The intermediate Truffle runtime nodes keep the necessary metadata to decide whether a method is hot enough or not for compilation.

Every call node needs a `CallTarget` that it is meant to invoke. The `CallTarget` in turn is obtained from the Truffle runtime by calling the `TruffleRuntime#createCallTarget`, which expects the `RootNode` of the `CallTarget` as an argument. `CallTarget`s are usually created during the parsing phase of the evaluation, before any source code is executed. Whenever the parser finishes parsing a function definition it creates a `CallTarget` for the method and stores it.

TruffleReloader leverages the fact that every invocation in Truffle goes through the above mentioned control flow by injecting a proxy node into it. Figure 3.2 shows the same flow when a proxy `CallTarget` is added. When Truffle thinks it is invoking the actual `CallTarget` it is instead calling a dummy proxy, quite similarly to the Truffle Instrumentation API. Unlike the Instrumentation API, the proxy nodes are created automatically by modifying the needed Truffle runtime method to wrap the returned `CallTarget`s with a proxy. This approach ensures that TruffleReloader receives a callback before any method invocation.

Before every invocation the proxy `CallTarget` checks whether the `Source` of the actual `CallTarget` has changed. In case it has changed TruffleReloader invokes the language parsing method. During parsing new `CallTarget`s are created, which are also wrapped. At this point TruffleReloader has created two ASTs, parallel universes, both containing our proxies.

The next task is to map the old `CallTarget` to its counterpart in the newly constructed AST. In the simplest option this can be done by parsing the function name from the underlying `SourceSection` of the `RootNode`. As mentioned in the previous chapter, the `SourceSection` points to the continuous section of source code that represents this AST node. For `RootNode`s it is the method declaration in the source code. Thus TruffleReloader can simply parse the method name from the method declaration. Using this simple strategy one can match the old and the new `CallTarget` by the underlying method name and redirect the proxy to the newest known `CallTarget`.

In practice this means that TruffleReloader has to keep a reference to every `CallTarget` as they are recreated all at once during parsing, but matching happens later, when the proxy is actually invoked. In TruffleReloader, whenever a proxy `CallTarget` is created, a `java.lang.ref.WeakReference` to the wrapped `CallTarget` is stored in a mapping with the parsed method names as keys.

Weak reference are kept to avoid creating additional memory overhead. The assumption is that if a `CallTarget` is needed, the AST itself or the global context holds a strong reference to it. Whenever a `CallTarget` is not needed anymore, the AST removes the reference and it can be reclaimed.

Now TruffleReloader has all the needed pieces to achieve basic reloading:

1. Proxy `CallTarget` detects a source code change and triggers parsing.
2. Parsing creates new proxy `CallTarget`s. TruffleReloader determines their corresponding method names and updates the reference mapping with the newer values.
3. After parsing, control flow is redirected to the (potentially) new `CallTarget`.

However as stated, this strategy is limited by at least two problems:

1. Using method names, which are parsed directly from source code, as keys in the mapping is not a viable solution for many real programming languages due to existence of different scopes.
2. In practice this strategy will also not allow us to rename methods or add new ones. Thus its functionality would be similar to the standard JVM HotSwap [11], which is limited to updating statements inside methods.

These problems are addressed by the remaining two components of TruffleReloader.

3.2. CallTarget Identity

Besides using named functions to organize code, many programming languages also have notions of classes or modules that make up larger chunks of code; usually with the goal of helping the programmer navigate on higher abstraction levels. Say a developer is working on two modules. One for calculating trigonometry values and another for modeling human social behavior. It is not impossible for both of these modules to contain a method called `#sin`. The first calculates the sinus function and the other that makes a person commit a bad deed.

The reloading strategy outlined above has no way of distinguishing between the two and after a reload will start redirecting invocations for both of these methods to only one of them. The underlying problem is that programming languages can have arbitrary ways to group functions together. Truffle does not impose any concrete structure here, and rightly so, as one of the goals of Truffle is to be able to support a wide range of programming languages.

The problem boils down to understanding the *context* of the function definition. The goal is to uniquely identify a `CallTarget` so that it can be correctly matched with its newer counterpart on reload. Unfortunately there is no way of achieving this for *any* programming language. Thus TruffleReloader leaves this burden to the language implementer as only they can tell how to uniquely identify a `CallTarget` from others for their language.

CallTarget Identity simply means that every Truffle based language has to provide an unique stable identity for every `CallTarget`. The identity has to be globally unique, so that functions that have the same name, but belong to different context (classes or modules) are not mixed up. The identity also has to be stable in that the same function in the same context will always yield the same identity. As an example for Java a good identity would be the fully qualified class name where the method is defined together with the method signature.

A correct identity will ensure that TruffleReloader does not mix methods that have the same name, but are defined in different modules. One could safely modify and invoke the trigonometry `#sin` implementation without worrying about a potential transgression.

3.3. Partial AST Replay

After the problem of methods with the same name is solved the next challenge involves renaming and adding methods. While the solution described so far can be useful in and of itself, it is limited to supporting small fixes within method bodies. Ability to refactor existing methods into smaller, more concise, ones is considered a good programming practice and can make the program easier to read and understand [27, Chapter 3].

The reason the above solution does not work for renaming and adding methods is again dependent on the language implementation details. For Simple Language the solution already supports also renaming and adding methods. As SL keeps all functions in a global registry, which is populated fully during (re-)parsing (as mentioned in Section 2.5), then both renamed and new methods will be registered. During execution SL will successfully find and be able to invoke those functions.

In languages that have notions of different scopes the implementation cannot be as simple. For example in both Python and Ruby there are defined rules for method lookup, where they usually start searching from the most concrete scope, moving up the hierarchy to broader scopes. The method is returned from the first scope that contains it.

We observed that when such languages are implemented in Truffle they will do the population of scopes as first steps of the AST. As the tree is parsed from source code, the first nodes in the AST take care of inserting function references to the correct scopes, depending on the language semantics. With the current strategy when TruffleReloader redirects execution to a reloaded `CallTarget` that tries to invoke a new method it would fail to look up the new method as its reference has not been written to the appropriate scope.

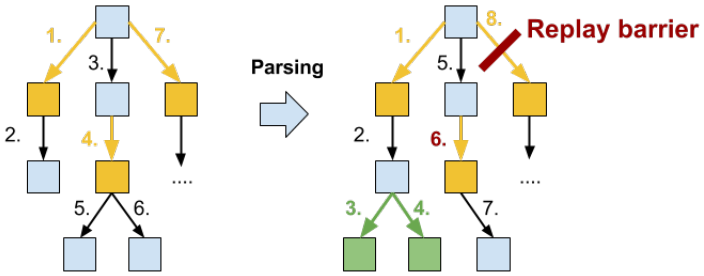

Figure 3.3: Illustration of Partial AST Replay

To overcome this obstacle TruffleReloader employs the technique called *Partial AST Replay*. Partial AST Replay simply means that after re-parsing, the new AST will be partially executed up to a certain point. Similarly to replaying only the first part of a music track. This will allow us to execute all the nodes that write function references to the various scopes again, which will ensure that all new methods are registered and later found.

Figure 3.3 illustrates the process. The left side of the image shows the initial AST, with first seven executed nodes drawn out. An important note here is that, not all executed nodes go through the method invocation flow; most of them just directly call the `#execute` method on some other node. On Figure 3.3 yellow boxes illustrate the nodes that invoke a method, which means those are the nodes that also go through our proxy `CallTarget`s.

Say a programmer added a couple of Python methods to the running program. After proxy `CallTarget` has triggered a reload, the new AST contains additional nodes (colored green). Before redirecting execution to the new `CallTarget` from our proxy node, TruffleReloader also starts executing the new AST.

At a certain point, called the *replay barrier*, it stops the execution and returns control flow to the old AST. This is achieved by throwing an exception when the replay barrier is hit, which will be caught in the proxy node. When the proxy node that triggered the reload returns, TruffleReloader has re-parsed the source code and also partially executed the new AST. Assuming that the replay barrier is at the correct place, then all necessary scopes should be repopulated with (potentially) new function references.

Of course the knowledge about which nodes to re-execute and where to stop the replay is once more language implementation specific. There can be no common replay barrier, as every programming language can define scopes with arbitrary structure and implementation details. It is left up to the language implementer to provide the correct location for stopping the replay.

```
public interface LanguageReloader {

  String mimeType();

  Supplier<CallTargetIdentity> getIdentityFor(RootCallTarget
      callTarget);

  ReplayController getReplayController();

  default Predicate<String> acceptCodePath() {
    return (path) -> true;
  }
}
```

Listing 3.1: TruffleReloader SPI main interface

3.4. TruffleReloader SPI

Both *CallTarget Identity* and *Partial AST Replay* require some input from the language implementer. To achieve this, TruffleReloader provides a Service Provider Interface (SPI) that the language implementers, wishing to add reloading to their language, are expected to implement. Listing 3.1 shows the main interface that is the touch point between TruffleReloader and a given programming language.

The first method `#getDescriptorFor` has to return a unique `CallTargetIdentity` for a given `RootCallTarget`. `CallTargetIdentity` is a concrete class that consists of just two `java.lang.String` fields for the context and method names. `#getDescriptorFor` returns a `Supplier` to communicate the fact that `CallTargetIdentity`s are retrieved, when they are really needed for the first time, which is only after the Partial AST Replay has finished.

Determining the identity of a `CallTarget` after the AST replay is needed to give the language implementation a chance to correctly initialize the required metadata to uniquely identify the `CallTarget`. The suppliers of `CallTargetIdentity`s are stored when new `CallTarget`s are created during (re-)parsing, but at that time it might not yet be possible to know the unique identity due to missing metadata. The idea is that when the AST replay finishes all the necessary metadata is present, so that all `RootCallTarget`s can be uniquely identified.

Listing 3.2 shows the entire `ReplayController` interface that language implementers also have to implement. It is used to define when to stop the Partial AST Replay. Additionally it provides two life cycle methods that language implementers can use to carry out additional actions. The `#beforeStart` method is called before the source code is re-parsed, so it can be used to clear some internal caches or carry out other necessary

```
public interface ReplayController<T extends ExecutionContext> {

  default void beforeStart(T context, Source source, Object[] currentArgs) {}

  default void afterStop(T context, Source source, Object[] currentArgs) {}

  boolean shouldStopAt(RootCallTarget executableCallTarget);
}
```

Listing 3.2: ReplayController controls the Partial AST Replay

preparations. Similarly, the `#afterStop` method is called when AST replay has stopped, but before forwarding execution to the new AST. It can be used to run any migration steps needed to transfer items between the old and the new ASTs.

TruffleReloader normally proxies all functions declared in a file based `Source`, but often the language's standard library can also be implemented in the language itself. It is not usually desirable to reload the standard library, so the TruffleReloader SPI provides a way to exclude creating proxies for certain source code based on its canonical file path. The `#acceptCodePath` is an optional method that can be used to control whether to create proxies for the functions declared in the given source code file or not.

Finally, TruffleReloader is left with the problem of finding and initiating the `LanguageReloader` instances implemented by different languages. We rely on the standard JDK utility `java.util.ServiceLoader`, which was designed for just such use cases; to find and load service providers that implement a predefined service interface.

To register a service provider, the language implementer has to create a provider configuration file. It has to be stored in the `META-INF/services` directory of the Truffle language's JAR file. The name of the configuration file has to be the fully qualified class name of the service provider, in which each component of the name is separated by a period [28]. For TruffleReloader, the file name has to be:
com.poolik.truffle.reload.spi.LanguageReloader.

In that provider configuration file the language implementer has to write the fully qualified class name of the service provider class. On startup TruffleReloader calls the `ServiceLoader#load` method to find all registered implementations of the `LanguageReloader` interface. The correct `LanguageReloader` for a language is chosen by matching the MIME type returned by the `#mimeType` method to the MIME type of the source code file.

Truffle language implementers should already be familiar with the way `ServiceLoader`s work as Truffle itself also uses the same mechanism for detecting the MIME type of different source code files. Based on this file MIME type Truffle knows which language implementation from all the available ones it should use for evaluation, as each Truffle

language also declares its expected MIME type.

3.5. Discussion

We introduced TruffleReloader, a tool consisting of three main approaches to achieve reloading behavior that can be applied to various languages implemented using the Truffle framework. *Proxy CallTarget* makes sure that TruffleReloader can execute reloading checks before every method call and gives it a suitable place for forwarding the execution flow to the new AST.

CallTarget Identity helps by making sure that TruffleReloader always forwards to the correct new methods and *Partial AST Replay* tries to make sure that TruffleReloader can support adding and removing methods as well. Parts of the solution that depend on the language implementation details were pushed down to the language implementer level by requiring them to implement the TruffleReloader SPI.

While these approaches form the basis of TruffleReloader, there are many implementation specific challenges that were not yet discussed. As outlined so far, TruffleReloader works when Truffle is running in the interpreted mode (as described in Section 1.3). Details about how to support TruffleReloader on the GraalVM, with minimum overhead, are the topic of the next chapter.

Chapter 4

Implementation Challenges

GraalVM is where the Truffle framework really shines and is able to provide great performance for AST interpreters. The two modes of running Truffle are backed by two different implementations of the Truffle APIs. This presents a challenge for TruffleReloader, because the implementation of Truffle that runs with Graal differs substantially from the default one.

This chapter explains how TruffleReloader was modified to account for the different implementation of Truffle. The main approaches remained the same, just the injection points had to be changed. Besides that, the implementation was reworked to get the best possible performance from Graal when running with TruffleReloader. Finally, many implementation details about overcoming most important technical challenges will be explained to provide a thorough overview.

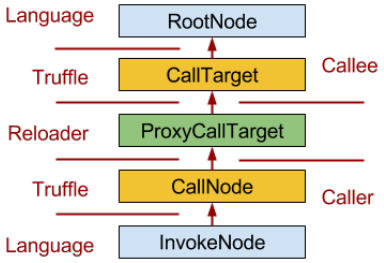

Figure 4.1: TruffleReloader injection point before

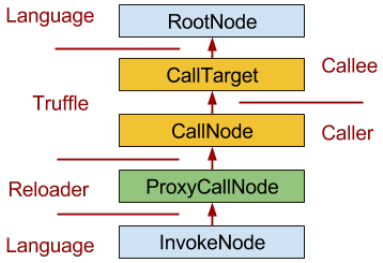

Figure 4.2: Updated TruffleReloader injection point

4.1. Reworking Truffle Hooks

Figure 4.1 shows how TruffleReloader initially injected itself between the caller and callee. Every `CallTarget` created in the Truffle framework was automatically wrapped in a proxy. It provided a convenient place to both redirect execution and to get a reference to every (re)created `CallTarget`.

Unfortunately the same strategy will not work on GraalVM, simply because it makes certain assumptions about the caller and callee classes. Graal assumes the `CallTarget` implementation will be of a concrete class and will effectively break the `CallTarget` contract by skipping the `Object call(Object... arguments)` method invocation; instead it will call a method on the anticipated class directly.

More modifications to the Graal Truffle implementation are required to make the previous strategy work. Seeing that Graal has a much closer relationship between the caller and callee we thought it better to change the integration point. The goal was to decrease the amount of changes required to Truffle and hopefully reduce future incompatibilities, when Graal developers decide to do further optimisations/modifications between the caller and callee nodes.

The only remaining option was for TruffleReloader to intercept the call on the language invoke node and Truffle call node boundary. Figure 4.2 illustrates the new injection point. However, TruffleReloader still needs to be aware of each created `CallTarget` so a second hook was inserted before Truffle returns the `CallTarget` to enable us to keep a weak reference to it.

The two hooks help us achieve the same behavior as before. The downside of the new approach is that it requires more integration code. There are two alternative call nodes that language implementers can use — direct and indirect; both of them need to be proxied to correctly support reloading. When previously we had to implement and proxy a single method call from the `CallTarget` interface then now we need to delegate all of the method calls of the call node to the wrapped nodes.

4.2. Low Overhead Reloading

Ideally TruffleReloader would not introduce any overhead to the peak steady state performance of the Truffle-based language. This is impossible when Truffle runs in the interpreted mode, but on GraalVM leveraging the `Assumption`s used in node rewriting help us minimize the overhead when nothing has actually changed.

The proxy call nodes have to do two things for each invocation:

1. Check whether we are currently running a Partial AST Replay. If yes, then also check whether we have to stop the replay at the current invocation, otherwise proceed.
2. Check whether the underlying `Source` has changed. If yes, then trigger a reload and replace the current wrapped call node with a new one that has the latest `CallTarget`, otherwise proceed.

One can see, however, that if nothing has changed, there is no need to do anything. TruffleReloader assumes that the `Source` has not changed and that it is not running a Partial AST Replay. Until the assumptions are invalidated Graal should be able to optimize away all of the actions performed when the assumptions do not hold. TruffleReloader hooks and their associated overhead are effectively removed.

To invalidate the assumptions, TruffleReloader creates a background thread that periodically monitors all the source files for changes. Whenever a change is detected the background thread invalidates the `sourceFileNotChanged` assumption associated with the changed source file. This will cause Graal to deoptimize the compiled code for that source and revert back to interpreter. The interpreter then triggers the correct reloading behavior.

4.3. Partial Re-Parsing

TruffleReloader does not parse the entire program again, but only the changed files. The reason behind this decision is simple — parsing only changed files is faster than starting to re-evaluate the entire program. The risk of using incremental re-parsing is that the partially produced AST may potentially violate assumptions made by language implementers, but it works well in practice for the present set of Truffle languages.

One of the problems is that the partial AST will only have nodes representing the method and module definitions declared within the changed files. However, the global context associated with the `TruffleLanguage`, which remains the same after parsing, still has references to all of the method and module definitions. Similarly, TruffleReloader will have a reference to the newest versions of all `CallTarget`s.

Another challenge concerns multi-file programs, where the language implementation might cache the imported module definitions. This could result in old versions being used for importing or that re-parsing will not happen due to cache hits of the module definitions. As a solution, `LanguageReloader`s can use the `#beforeStart` and `#afterStop` life cycle methods of the `ReplayController` to remove the needed items from the global cache of imports.

There might be other potential problems, when the language implementation assumes a special role for the source file where evaluation starts. Implementations might implicitly tag it as a *main* module in which case that knowledge cannot be used as part of the `CallTarget` stable identity, as during partial re-parsing new modules will be marked as *main*. Instead, the file name could be used as prefix to the identity.

4.4. Handling Multiple Threads

Truffle language implementations are free to build any concurrency primitives they prefer, with the obvious constraints of the underlying JVM. The consequence of this is that Truffle languages can execute methods in several threads and therein can trigger a reload in any of those threads. This complicates reloading because Truffle remembers the thread where source evaluation was started and keeps a reference to the created metadata objects, such as the global language context, in a thread local variable. If the reloading thread is not the same as the one where evaluation was started, then Truffle does not have access to the thread local metadata and re-parsing results in errors.

To support reloading from any thread TruffleReloader keeps its own reference to the thread local metadata. The reference is obtained when TruffleReloader is first initiated for a given `Source`. When a reload is triggered, TruffleReloader simply sets the current thread local to the initial value; making sure that Truffle can access the right metadata context during reload.

4.5. Shape Matching

As mentioned in the Introduction, Truffle provides an Object Storage Model (OSM) for implementing types and objects. The implementation of Truffle objects uses two separate parts; a *Shape* defines the overall structure of the object, similar to a Java class, and *Object storage* that contains the per-instance data, similar to a Java object in the heap [3]. The object storage has a predefined number of slots (fields on a class) for primitive types and reference types and a reference to its current shape that knows what properties are stored in which slot.

A common problem `LanguageReloader`s have to solve for languages using the OSM is reflecting the changes to class definitions. This might happen automatically, when the language implementation is already built with support for class redefinition. For example all classes in Ruby are always open for extension; a single class definition can be spread out over several files, all of which extend the same class definition. This means that for the Ruby implementation all of the class definitions will automatically be up to date after the reload, because the language modifies the current shapes of objects instead of creating new ones. For languages that normally do not support class definition changes at runtime, new shapes are created on reload for the updated classes, which means that existing objects keep using the old shapes.

Recall that in Truffle-based languages all state is typically kept in the context of the evaluation. `LanguageReloader`s can use the life cycle methods of the `ReplayRecorder` to recursively iterate over all of the objects in the context and update object storage

instances to point to their newest shape. This requires that it is possible to match the old and the new shape of a class definition, which can usually be done based on the class name. The details are highly dependent on the language implementation, but the general approach remains the same.

By iterating and updating all objects to point to their newest shapes TruffleReloader can successfully reload class definitions for existing objects as well. All new objects in the updated AST are created using the newest shape; no additional work needed.

4.6. TruffleReloader Agent

To enhance the usability of TruffleReloader the required hooks are automatically injected into the Truffle framework classes by means of a Java agent [29]. A Java agent consists of a JAR file that specifies the agent class which will be loaded at the start of the agent in a manifest attribute. The JVM will invoke the `premain` method of the agent class handing it a reference to an `Instrumentation` API object.

The `Instrumentation` interface provides services needed to instrument Java code in the running JVM. Java agents are used extensively by many existing tools in the market to enhance applications with various capabilities, such as profiling[1], aspects[2] or reloading itself[3]. TruffleReloader leverages the `Instrumentation` API to automatically transform Truffle runtime classes to insert the required hooks. All the user has to do to enable TruffleReloader is specify an additional JVM startup argument:
`-javaagent:/path/to/truffle-reloader.jar`.

The agent is built by using an existing Java code generation library called Byte Buddy [30]. Byte Buddy simplifies the creation of Java agents and the registration of class file transformers to modify the needed Truffle classes. It provides an Java API for modifying classes that works on a higher abstraction than manual Java bytecode generation and is thus easier to use.

```
public interface Patcher {

  String className();

  AgentBuilder.Transformer transformer();
}
```

Listing 4.1: TruffleReloader Patcher interface

If the support for TruffleReloader is added by someone other than the original language author, it might be necessary to tweak the implementation in some places in order to

[1] http://zeroturnaround.com/software/xrebel/
[2] http://www.eclipse.org/aspectj/
[3] http://zeroturnaround.com/software/jrebel/

properly implement the SPI. For these purposes, TruffleReloader provides an experimental API to hook into the TruffleReloader Agent to register additional bytecode modifications. To transform additional classes, a class has to implement the Patcher interface (shown on Listing 4.1) and made discoverable via the same `ServiceLoader` mechanism as the `LanguageReloader`. Every patcher can currently register a Byte Buddy bytecode `Transformer` for a single class. All patchers are applied when the TruffleReloader Agent starts up.

4.7. Discussion

As was discussed, there are many important implementation details to making the proposed approaches work in practice. The devil is in the details and language implementations in Truffle have quite a bit of leeway in their decisions, which makes building a generic reloading solution a bit harder.

With the above described modifications in place TruffleReloader can run on GraalVM and should have minimal, if any, overhead for the language runtime. The next chapter follows with some benchmarking results to analyze the actual overhead and describe reloading effects on performance. Additionally we detail the covered reloading use cases and the testing harness created to validate them.

Chapter 5

Evaluation

Let us now turn our gaze towards investigating the generality and soundness of the developed approaches. We present an overview of the supported reloading scenarios at the time of writing and an initial investigation into the performance effects of TruffleReloader hooks. We evaluate TruffleReloader with regards to following Truffle-based languages:

1. **ZipPy**[1] — A prototype Python 3 interpreter, written largely by Wei Zhang as part of his PhD thesis [31].
2. **JRuby+Truffle**[2] — A high performance Ruby 2.2 implementation, written largely by Chris Seaton as part of his PhD thesis [32].
3. **Graal.js**[3] — An ECMAScript 262 version 2015 compliant JavaScript engine developed by Oracle Labs.
4. **Simple Language** — Language to showcase the Truffle framework, introduced in Chapter 2.

For each of the listed languages, we implemented the TruffleReloader SPI and packaged it into the TruffleReloader Agent. Reloading will work out of the box when TruffleReloader is enabled for those languages.

5.1. Functionality

As language implementations can vary greatly in their details, we opted to writing tests that cover certain reloading scenarios in different languages to prove that the described approaches are viable and work in practice. Table 5 lists all of the currently tested and supported use cases. Many of them are not applicable for SL as it does not have the notion of classes. Similarly the version 0.10 of Graal.js does not support the ECMAScript

[1] https://bitbucket.org/ssllab/zippy

[2] https://github.com/jruby/jruby/tree/truffle-head

[3] http://www.oracle.com/technetwork/oracle-labs/program-languages/downloads/index.html

Table 5.1: Tested and supported reloading use cases across different languages

Use case	**SL**	**ZipPy**	**JRuby+Truffle**	**Graal.js**
Changing a function	y	y	y	y
Changing multiple functions	y	y	y	y
Adding a new function	y	y	y	y
Multiple consecutive reloads	y	y	y	y
Changing global variable definition	n/a	y	y	y
Adding a new global variable definition	n/a	y	y	y
Reloading functions with same name	n/a	y	y	y
Adding a new class	n/a	y	y	y
Changing a static method	n/a	y	y	y
Changing an instance method	n/a	y	y	y
Adding a new instance method used from variable	n/a	y	y	y
Adding a new instance method used from new object	n/a	y	y	y
Adding a new instance method used from field	n/a	y	y	y
Adding a new instance method used from global variable	n/a	y	y	y
Multiple file reload, changing a method in dependency	n/a	y	y	n/a
Multiple file reload, adding a new method in dependency	n/a	y	y	n/a
Multiple file reload, changing an instance method in dependency on existing object	n/a	y	y	n/a
Multiple file reload, changing an instance method in dependency on new object	n/a	y	y	n/a

6 modules, so multiple file tests can not be run.

To test these use cases a simple reloading testing harness was developed that controls the evaluation of the program via standard I/O. Tests are written as simple programs with multiple versions of the program in different folders named `versionN`, where `N` stands for the version number; however, some changes to ZipPy and JRuby+Truffle were needed to make the testing harness work.

For JRuby+Truffle we re-implemented the `puts` keyword to use the Java `System.out` output stream, as it defaulted to a native process stream that could not be controlled from

tests. For ZipPy we implemented a `FileTypeDetector` to make the implementation work using the `PolyglotEngine` abstraction. We also changed the `PythonLanguage#parse` method to use the previously created context instead of creating a duplicate on each invocation. Changing the `#parse` method was needed to make sure the assumptions of TruffleReloader hold, but it can be seen also as a bug in the language implementation since there is no need to create a duplicate context in `#parse`.

When the test program signals a version switch by writing a special token to the standard output, the harness copies over all the source code files from the next version to the working directory. The program waits for this to finish by trying to read from the standard input after triggering a reload, which will block until the switch is complete. This simulates the behavior of a programmer making edits in source code, saving the file and triggering the new code to execute (by a HTTP request for example).

The tests verify that the correct behavior was achieved by validating the expected output of the program. All tests loop `n` times and output something at each iteration. Reloads are triggered at specific iterations, after which the output changes in the new version of the code. As reloads happen at known times, tests simply validate whether the real program output matches the expected when the program finishes.

5.2. Case Study

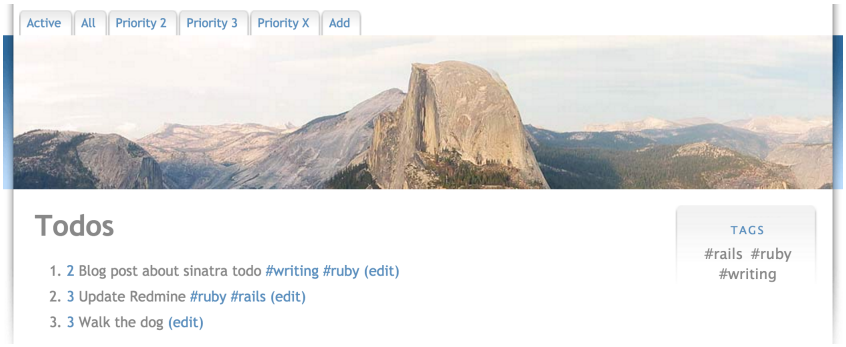

Figure 5.1: Overview of the sample application

To verify TruffleReloader's usefulness in a real world scenario, we tried running and reloading a simple open source Ruby web application[4]. At the time of writing JRuby+Truffle can not run Ruby applications that leverage native extensions, so we had to find a simple enough application to test with. The test application is a web UI for a file based task management system. It parses a file containing tasks on each line and presents them in an easily navigable form. Figure 5.1 shows the main screen of the application.

[4] https://github.com/edavis10/sinatra_todo

Within the application we did a series of fixes/improvements that a developer might be required to do without having to restart the application once. The initial application was forked and our fixes pushed there[5].

5.2.1. Fixing Task Updating

Listing 5.1 shows the change needed to fix updating task descriptions. The bug was caused by directly writing the array of tasks back to the file, which resulted in the file containing a single array of tasks, instead of the tasks themselves on each line. Without the fix the application corrupts the tasks file on a single task update as subsequent parsing of the file will fail.

```
diff --git a/lib/todo.rb b/lib/todo.rb
@@ -120,7 +120,7 @@ class Todo

     begin
       tmp = Tempfile.new("todo")
-      tmp.write(file_data)
+      tmp.write(file_data.join)
```

Listing 5.1: Code diff to fix task updating

5.2.2. Hiding Done Priority

The menu at the top of the application has links for seeing all tasks, active tasks, adding a new task and dynamically generated links for all of the defined priorities. Done tasks are marked with the `X` priority, but the application generated a link for looking at the done priority as well. Listing 5.2 shows the changes we did to stop generating the link for done items. We added a new utility method to distinguish done priorities and refactored an existing method to use the new utility to reduce code duplication.

```
diff --git a/lib/todo.rb b/lib/todo.rb
@@ -14,3 +14,3 @@ class Todo
   def active?
-    self.priority && !self.priority.match(CompleteProirityRegex)
+    self.priority && !Todo.is_done(self.priority)
   end
@@ -25,3 +25,7 @@ class Todo
   end
-
+
+  def self.is_done(priority)
+    priority.match(CompleteProirityRegex)
+  end
+
   def self.all
```

[5] https://github.com/poolik/sinatra_todo

```
@@ -72,3 +76,3 @@ class Todo
     end
-    priorities.uniq.sort
+    priorities.uniq.sort.select { |priority| !Todo.is_done(priority) }
   end
```

Listing 5.2: Code diff to hide done priority link

5.2.3. Marking Tasks as Done

The initial application had no means for marking a task as done, so it was implemented. To support this use case we needed to change the template to include a new link for each task and add a new request handler to mark the task as done. Listing 5.3 shows all of the code changes needed to do mark tasks as complete.

```
diff --git a/lib/todo.rb b/lib/todo.rb
@@ -26,2 +26,7 @@ class Todo

+  def done
+    self.raw_line[0] = 'X'
+    update(self.raw_line.chomp)
+  end
+
   def self.is_done(priority)
diff --git a/sinatra_todo.rb b/sinatra_todo.rb
@@ -35,2 +35,6 @@ helpers do
   end
+
+  def done_link(todo)
+    "<a class='small' href='/done/#{todo.line_number}'>(mark as done)</a>"
+  end
 end
@@ -75,2 +79,8 @@ end

+get '/done/:line' do
+  @todo = Todo.find(params[:line])
+  @todo.done
+  redirect '/', 302
+end
+
 put '/update' do
diff --git a/views/index.erb b/views/index.erb
@@ -14,2 +14,3 @@
   <%= edit_link(todo) %>
+  <%= done_link(todo) %>
 </li>
```

Listing 5.3: Code diff to add 'mark as done' button

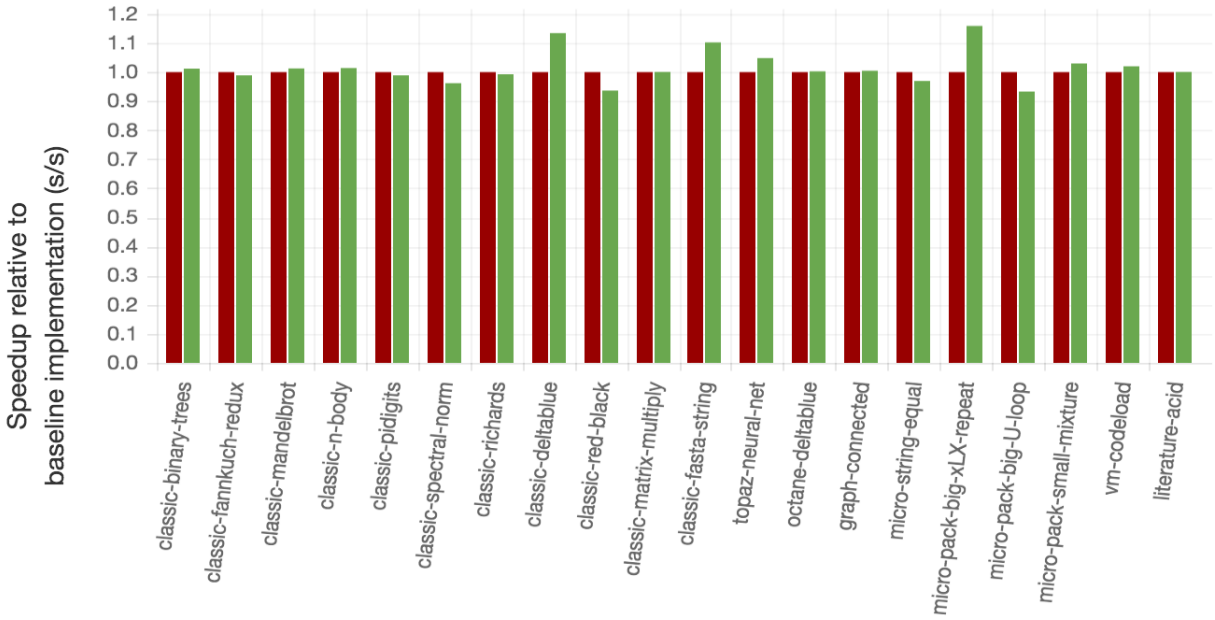

Figure 5.2: Effects of TruffleReloader on steady state performance

5.3. Benchmarks

To measure the overhead of TruffleReloader hooks we reused the benchmarking harness created by Chris Seaton to evaluate the JRuby+Truffle implementation [33]. The benchmarks focus on measuring *peak temporal performance*, otherwise also known as *steady state*. The term loosely means the peak performance of an application after it has had time to optimize and stabilize the AST.

The benchmarks include a combination of synthetic benchmarks from the Computer Language Benchmarks Game as well as parts from various Ruby libraries, stressing different aspects of the language [32].

Thanks to the use of `Assumption`s TruffleReloader has a very low overhead on the steady state performance on Graal. Figure 5.2 shows the results of running a set of synthetic benchmarks on GraalVM with TruffleReloader enabled (colored green) compared to the default GraalVM runtime (colored red). Some benchmarks show a small overhead whereas some show an increase in performance. We attribute this to the non-deterministic nature of Graal, where the hooks of TruffleReloader might have caused it to better optimize some code paths.

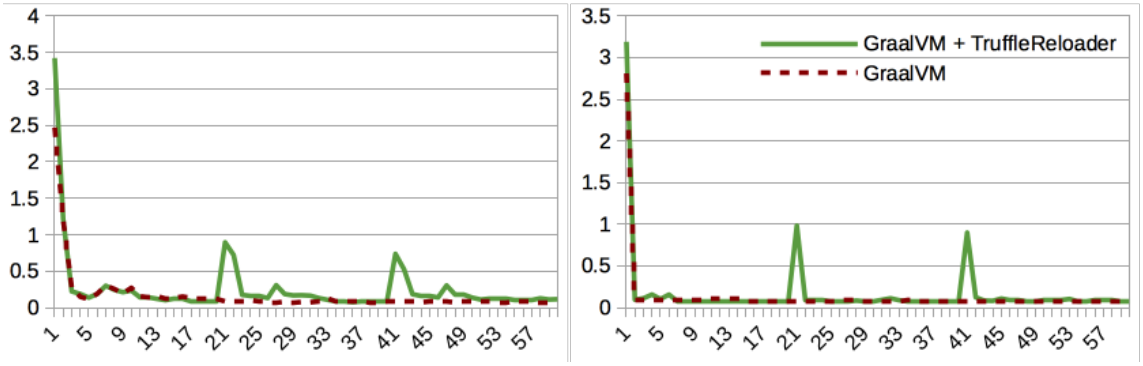

Figure 5.3: Overhead of reloading running code. The left chart shows the matrix multiplication and the right one the mandelbrot benchmark. The x-axis shows the iteration number and the y-axis the iteration time in seconds.

Additionally we plotted the overhead of triggering a reload mid benchmark. Figure 5.3 illustrates the overhead of doing a reload. We executed the `matrix-multiply` and `mandelbrot` benchmarks 60 times and triggered a reload after the 20th and 40th invocation. As expected the first invocation of a reloaded code is slow, as the AST is in an uninitialized state, but after a few invocations the tree is compiled again and achieves the same steady state performance as before.

Conclusions

This thesis started with the goal of figuring out whether the Truffle framework can help language implementers in adding reloading to their languages. As we've seen there are several ways how Truffle can be adapted to meet this goal. After several iterations and refinements, we ended up with a reusable reloading core, called TruffleReloader, that different languages can hook into to add the capability for dynamic updates.

Plugging new languages into TruffleReloader requires some language-specific implementation, but these efforts are negligible compared to the work required to implement a language, with built in reloading support, that performs on par with the Truffle framework. All that language developers have to do to support reloading is implement the TruffleReloader SPI and make their implementation discoverable. Our system even allows external developers to add reloading support for a language, if needed, as `LanguageReloader`s and language implementations are not tightly coupled.

Initial benchmarking showed that thanks to the use of the `Assumption` mechanism provided by Truffle, our solution has close to zero overhead on GraalVM. Each reload incurs an initial performance penalty, but in time the system should be able to achieve the same peak performance as before the reload. Having a low impact on performance is the first step towards being able to keep TruffleReloader running on production systems to achieve high availability, with the option to apply fixes when needed. Reloading production systems would, of course, require additional and thorough testing, but low impact on performance is important also during development; to achieve a fast feedback cycle.

TruffleReloader works thanks to the fact that all programming languages have units of executable code, what we call functions, where we can redirect the execution flow to newer versions of said functions. Reloading all other language features depend more on the implementation details of that language, thus the difficulties in supporting reloading lie in different areas. For some languages, updating class definitions requires no additional work (JRuby+Truffle for example), but others require more work to be done after the AST replay.

Having successfully integrated TruffleReloader with four different language implementations, we have a strong belief that the system is in fact portable and reusable and

that the underlying reloading techniques are not geared towards reloading a specific language. We hope TruffleReloader will become a widely used productivity tool, for many languages, once implementations mature and gain widespread adoption.

Future Work

Truffle itself and the languages built on top of it are in active development. We intend to keep up with their developments and, if need be, revise some design decisions. As existing implementations mature and developers start using them, we hope to get feedback from more people trying to reload changes in their real world applications, so that we can expand the list of covered use cases where needed.

Some concrete improvement ideas include adding a native file system watcher to get notifications when source code is changed, instead of polling in a background thread. Should the slightly risky incremental re-parsing strategy prove to incur incorrectness for new languages, an option can be added to parse the entire program instead. While this will have a negative effect on the reloading speed, the correctness is regained.

The experimental `Patcher` API might see radical changes when we encounter more complicated bytecode transformation requirements. Currently it directly exposes the underlying Byte Buddy transformation API, which tightly couples TruffleReloader to it. A better option is to provide an abstract interface for transforming bytecode, so implementations would be able to choose concrete technologies themselves. Besides the above listed ideas for improvements, we'll tackle any unforeseen obstacles with enthusiasm and eagerness to improve TruffleReloader.

Bibliography

[1] C. Wimmer and T. Würthinger, "Truffle: A self-optimizing runtime system," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, (New York, NY, USA), pp. 13–14, ACM, 2012.

[2] C. Seaton, M. L. Van De Vanter, and M. Haupt, "Debugging at full speed," in *Proceedings of the Workshop on Dynamic Languages and Applications*, Dyla'14, (New York, NY, USA), pp. 2:1–2:13, ACM, 2014.

[3] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck, "An object storage model for the truffle language implementation framework," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, (New York, NY, USA), pp. 133–144, ACM, 2014.

[4] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck, "High-performance cross-language interoperability in a multi-language runtime," in *DLS 2015*, pp. 78–90, ACM, 2015.

[5] V. J. Shute, "Who is likely to acquire programming skills?," *Journal of educational Computing research*, vol. 7, no. 1, pp. 1–24, 1991.

[6] J. S. Reitman, "Without surreptitious rehearsal, information in short-term memory decay," *Journal of Verbal Learning and Verbal Behavior*, vol. 13, no. 4, pp. 365–377, 1974.

[7] "Developer productivity report 2012: Java tools, tech, devs & data." http://zeroturnaround.com/rebellabs/developer-productivity-report-2012-java-tools-tech-devs-and-data/. [Online; accessed 13-03-2016].

[8] E. Johansson, K. Sagonas, and J. Wilhelmsson, "Heap architectures for concurrent languages using message passing," in *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, (New York, NY, USA), pp. 88–99, ACM, 2002.

[9] F. Hébert, *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, 2013.

[10] J. Kabanov and V. Vene, "A thousand years of productivity: the JRebel story," *Software: Practice and Experience*, vol. 44, no. 1, pp. 105–127, 2014.

[11] "Java platform debugger architecture - java se 1.4 enhancements." https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/enhancements1.4.html#hotswap. [Online; accessed 25-02-2016].

[12] R. S. Fabry, "How to design a system in which modules can be changed on the fly," in *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, (Los Alamitos, CA, USA), pp. 470–476, IEEE Computer Society Press, 1976.

[13] T. Würthinger, C. Wimmer, and L. Stadler, "Dynamic code evolution for java," in *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, (New York, NY, USA), pp. 10–19, ACM, 2010.

[14] G. Hjálmtýsson and R. Gray, "Dynamic c++ classes: A lightweight mechanism to update code in a running program," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, (Berkeley, CA, USA), pp. 6–6, USENIX Association, 1998.

[15] A. Ranta, *Implementing Programming Languages: An Introduction to Compilers and Interpreters*. Texts in computing, College Publications, 2012.

[16] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. 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! 2013, (New York, NY, USA), pp. 187–204, ACM, 2013.

[17] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer, "Self-optimizing ast interpreters," in *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, (New York, NY, USA), pp. 73–82, ACM, 2012.

[18] U. Hölzle, C. Chambers, and D. Ungar, "Optimizing dynamically-typed object-oriented languages with polymorphic inline caches," in *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, (London, UK, UK), pp. 21–38, Springer-Verlag, 1991.

[19] S. Marr, C. Seaton, and S. Ducasse, "Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises," *SIGPLAN Not.*, vol. 50, pp. 545–554, June 2015.

[20] Y. Futamura, "Partial evaluation of computation process—anapproach to a compiler-compiler," *Higher Order Symbol. Comput.*, vol. 12, pp. 381–391, Dec. 1999.

[21] J. Rose, "JEP 243: Java-Level JVM Compiler Interface." http://openjdk.java.net/jeps/243, 2014. [Online; accessed 05-02-2016].

[22] C. Humer, C. Wimmer, C. Wirth, A. Wöß, and T. Würthinger, "A domain-specific language for building self-optimizing ast interpreters," *SIGPLAN Not.*, vol. 50, pp. 123–132, Sept. 2014.

[23] G. Duboscq, T. Würthinger, and H. Mössenböck, "Speculation without regret: Reducing deoptimization meta-data in the graal compiler," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, (New York, NY, USA), pp. 187–193, ACM, 2014.

[24] C. Wimmer, "Truffle tutorial." https://www.youtube.com/watch?v=N_sOxGkZfTg, February 2014. [Online; accessed 05-02-2016].

[25] M. V. D. Vanter, "Instrumentation api documentation." https://wiki.openjdk.java.net/display/Graal/Instrumentation+API. [Online; accessed 12-02-2016].

[26] G. Savrun-Yeniçeri, M. L. Van de Vanter, P. Larsen, S. Brunthaler, and M. Franz, "An efficient and generic event-based profiler framework for dynamic languages," in *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, (New York, NY, USA), pp. 102–112, ACM, 2015.

[27] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin series, Prentice Hall, 2009.

[28] "The java™ tutorials - creating extensible applications." https://docs.oracle.com/javase/tutorial/ext/basics/spi.html. [Online; accessed 25-02-2016].

[29] "Package java.lang.instrument." https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html. [Online; accessed 19-03-2016].

[30] R. Winterhalter, "Byte Buddy." http://bytebuddy.net/, 2014. [Online; accessed 19-03-2016].

[31] W. Zhang, *Efficient Hosted Interpreter for Dynamic Languages*. PhD thesis, University of California, Irvine, 2015.

[32] C. Seaton, *Specialising Dynamic Techniques for Implementing the Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

[33] JRuby Team, "bench9000." https://github.com/jruby/bench9000. [Online; accessed 02-04-2016].

Non-exclusive licence to reproduce thesis and make thesis public

I, Tõnis Pool (date of birth: 7th of February 1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Generic Reloading for Languages Based on the Truffle Framework
supervised by Allan Raundahl Gregersen and Vesal Vojdani

2. I am aware of the fact that the author retains these rights.

3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, 19.05.2016