LMS-Truffle:

combining staging with self-optimizing AST interpretation

Roman Tsegelskyi
Purdue University
rtsegels@purdue.edu

Abstract

Abstract syntax tree (AST) interpretation is an integral part of a wide variety of different tasks in computing, for example, programming languages interpretation, SQL interpretation, etc. A lot of significant research has been conducted to improve performance and decrease overhead of AST interpretation in comparison to executing compiled code. One of the recent interesting developments in that area is Truffle framework/Graal VM introduced by Oracle Labs, which has an ambitious goal to is to improve the performance of Java virtual machine-based languages to match native languages performance. This project presents and attempt to combine Truffle Framework/Graal VM with another interesting optimization concept - program staging. As a particular example, Lightweight Modular Staging is used to stage program code. The main goal of this project is to proved that needed machinery for using Truffle Nodes being a target for some typical uses cases of Lightweight Modular Staging and evaluate the performance.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Code Generation, Multi-stage programming, language implementation; optimization

1. Introduction

Since the advance in hardware have allowed implementations of interpreted languages to be more suitable for mainstream use, there have been a lot of attemepts to improve performance of interpretation with different techniques, while keeping the benefits that the high level of abstraction constructs provide. Just in scope of this class, we reviewed major techniques like multi-staged programming, partial evaluation, language embedding, self-modifying interpreters, etc.

In this report, I show some details of integration of Lightweight Modular Staging with Truffle framework/Graal VM using Scala as a host language. Abstract syntax tree nodes are implemented by Truffle Framework convention and are used as a target for code generation for lightweight modular staging. The rest of this report is orginized as follows:

- Rest of this section gives short overview of Truffle Framework/-Graal VM and Ligthweight Modular Staging.
- Section 2 describes idea behind integration of LMS and Truffle and provides some implementational details.
- Section 3 outlines experiments and benchmarks.
- Section 4 describes future work and possible improvements. And Section 5 concludes.

1.1 Graal VM and Truffle framework

Graal is a new experimental just-in-time compiler for Java developed by Oracle Labs. It is integrated with the HotSpot virtual machine and it's main objective is to improve the performance of Java virtual machine-based languages to match native languages performance. It tries is to provide excellent peak performance via new techniques in the area of method inlining, removing object allocations, and speculative execution. The term GraalVM is used to denote a HotSpot virtual machine configured with Graal.

Truffle is a multi-language framework for executing dynamic languages that achieves high performance when combined with Graal [4]. Truffle Framework/Graal VM is work in progress with a lot of ongoing work and active core community. There are several current open source projects building Truffle-based runtimes for other languages, e.g., Ruby (see "TruffleRuby"), R (see "FastR"), or Python (see "ZipPy").

1.2 Lightweight Modular Starging

Lightweight modular staging (LMS) is a novel dynamic code generation approach developed in EPFL and Oracle Labs. It exploints multi-stage programming paradigm, that is that many computations can naturally be separated into stages distinguished by frequency of execution or availability of information. Staging transformations aim at executing certain pieces of code less often or at a time where performance is less critical. Also aim of staging is to allow typesafe run-time code generation. LMS has certains distinctive features that differenciate it from other staging solutions like MetaOCaml, variation of Lisp, etc.

Some of the main characteristics of LMS, that are important for this project, are the following [2]:

- LMS does not require any special syntactic annotations, rather then distingushing binding types by types
- different code generation targets can be supported (heterogeneous staging); their implementations can share common code
- program generators have full control over when compilation happens and how compiled code is re-used

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Implementation details

LMS is centered around distinguishing binding times by changing declared types.

For example, take a look at the poster child of multi-staged programming.

```
def power(b: Double, x: Int): Double =
  if (x == 0) 1.0 else b * power(b, x - 1)
```

Annotating *Double* type of first argument wiht *Rep[Double]* means that computation will yield Double in the next stage.

```
def power(b: Rep[Double], x: Int): Rep[Double] =
  if (x == 0) 1.0 else b * power(b, x - 1)
```

From code with annotated types, LMS generated internal intermediate representation (IR) which can be simply viewed *(symbol, value)* pairs. Such intermediate reprensentation can be potentially translated to any target by providing an implementation for those nodes in IR as shown in Figure 1.

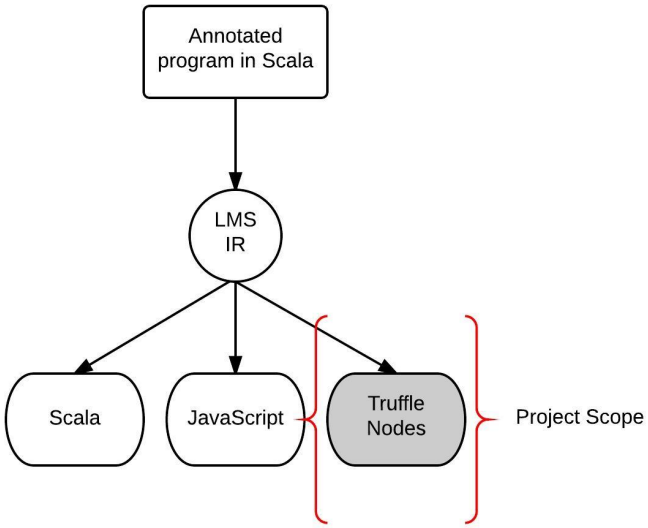

Figure 1. LMS code generation pipeline

As was stated before, the main goal of this project was to proved needed machinery for having Truffle Nodes as a code generation target. Particularly that involved that adding Truffle nodes for needed operations and taking care of proper vatiable assignment.

2.1 Truffle Nodes

Implementation of needed Truffle nodes start with defining nodes that represent Rep[T]. Essentially those are nodes that when called will return the appropriate bound value.

```
type Rep[+T] = Exp[T]

abstract class BaseNode extends Node with Product

trait Exp[@specialized +T] extends BaseNode {
  def execute(frame: VirtualFrame): T
}
```

Figure 2. Rep[T] and Exp Node

Main LMS procedure is creating a Truffle RootNode from Scala code with annotated types. *Reify* processes the IR and creates Assignment Nodes.

```
def lms[T:Typ,U:Typ](f: Rep[T] => Rep[U]) = new
     (T=>U) {
  val rootNode = {
    try {
      frameDescriptor = new FrameDescriptor();
      val t = reify(f(getArg[T](0)));
      new LMSRootNode(frameDescriptor,t);
    } finally {
    ...
    }
  }
  val target = runtime.createCallTarget(rootNode)

  override def apply(x: T) = {
    val result =
        target.call(Array(x.asInstanceOf[AnyRef]));
    result.asInstanceOf[U]
  }
}
```

Figure 3. LMS procedure

In the end sequence of Assignments are generated, for example in Figure 4.

```
Assign([0,x0,Int],GetArg(0))
Assign([1,x1,Int],IntTimes(Sym([0,x0,Int]),Sym([0,x0,Int])))
Assign([2,x2,Int],IntTimes(Sym([0,x0,Int]),Sym([1,x1,Int])))
Assign([3,x3,Int],IntTimes(Sym([0,x0,Int]),Sym([2,x2,Int])))
Assign([4,x4,Int],IntTimes(Sym([0,x0,Int]),Sym([3,x3,Int])))
Assign([5,x5,Int],IntTimes(Sym([0,x0,Int]),Sym([4,x4,Int])))
```

Figure 4. Truffle Nodes for Power(6) function

3. Evaluation and benchmarks

Other than implementing all needed machinery for Truffle Nodes genration, my other main goal was to compare different dimensions of performace and what kind of overhead comes from using Truffle framework/Graal VM as target for LMS. I picked 3 common use-cases where LMS brings significant advantages - Power function, Fast Fourier Transform and SQL interpretation[1].

The simple goal of my benchmarks was to compare how Truffle genereted AST compares to code in Scala and Staged Scala code. Where possible I was aiming to trigger AST compilation.

3.1 Power function

The classical example of staging application is to specialize the power function for a given exponent. Usual Scala implementation

To extend Truffle to support this not a lot of nodes were required.

Evaluation was done by calling power(2,2048) for 1000 times and power(2,8) - 10000 times. Also for last case I make sure that generated Truffle AST is compiled by Graal. In this case it is interesting to see how compiled Scala code differes from compiled Truffle nodes in terms of performance.

	Power(2, 2048)	Power(2, 8)
Pure Scala	3ms	3ms
Staged Scala	1ms	2ms
Staged Truffle	87ms	38ms
Compiled Staged Truffle	8ms	3ms

Table 1: Power function benchmarks

As can be seen from Table 1, Truffle nodes interpetation is significatly slower (20-40x), but once compilation is triggered performance gap almost disappers.

3.2 Fast Fourier Transform

Next comes far more interesting and practical example of the fast fourier transform (FFT). LMS in this case is used to specialize code for fixed size of input data. The staged code is free of branches and redundant computations.

For this task I needed to implement a concept of Arrays. I choose just a variant of Object slot as can be seen in the next figure. This might be one of the things that needs improvement, because each time before reading an element from array, and array has to be read from a slot.

```
case class ArrayRead[T](@(Child @field) arr:
     Exp[Array[T]], @(Child @field) x: Exp[Int])
     extends Def[T] {
  def execute(frame: VirtualFrame) = {
    val index = x.execute(frame)
    val res = arr.execute(frame)(index)
    res
  }
}
```

Figure 5. Array Read operation

Moreover, for this case I implemented different optimizations specific for FFT, for example trigonometric simplifications in Figure 5.

```
trait TrigOptFFT extends TrigOpt {
  val cos_values = Map(-2*Pi -> 1.0, -3.0/2*Pi -> 0.0,
      -Pi -> -1.0, -1.0/2*Pi -> 0.0, 0.0 -> 1.0,
      1.0/2*Pi -> 0.0, Pi -> -1.0, 3.0/2*Pi -> 0.0,
      2*Pi -> 1.0)

  override def sin(x: Rep[Double]) = x match {
    case Const(f) => if (sin_values.contains(f))
         Const(sin_values(f).asInstanceOf[Double]) else
         lift(math.sin(f))
    case _ => super.sin(x)
  }
   ...
}
```

Figure 6. Trigonometric optimizations for FFT

As can be seen from Table 2, for FFT Truffle performes better than for power function. It is still slower, but now the gap is rather smaller than for power function (3-7x). I think that the reason for better perfomance of Truffle Nodes for FFT than power function is percentage of time associated with Truffle overhead is smaller, because more operations are performed. It is also interesting to note that optimization have significant (improvement is 20-50%).

	FFT(4)	FFT(32)
Pure Scala	212ms	286ms
Staged Scala	106ms	159ms
Staged Truffle	710ms	991ms
Staged Truffle (with optimization)	438ms	728ms

Table 2: FFT benchmarks

3.3 SQL Interpretation

Another interesting application of LMS is and SQL query compiler. In such case LMS is used to generate a specific code for a table and query in contrast to traditional systems that interpret query operator by operator [1].

First of all, I implemented a comparable version of SQL interpretation using Truffle framework. Interesting part about this task was staging a scanner. Again, as in previous example, it might be better to move the implementation to slot assignment.

```
case class ScannerNew(@(Child @field) filename:
     Rep[String]) extends Def[Scanner] {
  def execute(frame: VirtualFrame) = {
    val descr = frame.getFrameDescriptor();
    val s = new Scanner(filename.execute(frame))
    s
  }
}
```

Figure 7. Staged Scanner

Intrestingly, as can be seen from the benchmarks, that version is quite close to same implementation in Scala (about 15-20% slower). That is the smallest gap among all experiments.

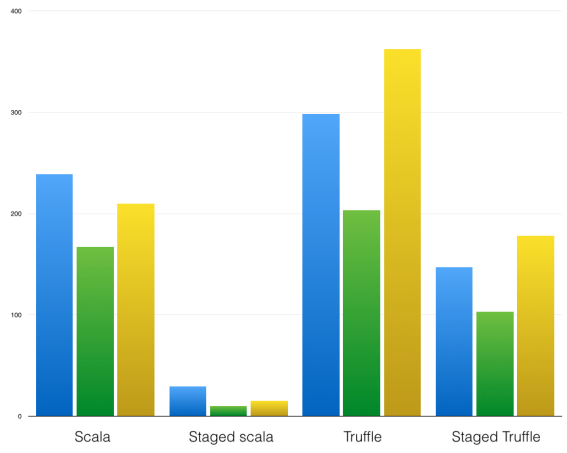

Figure 8. SQL interpretation pipeline

However, staging bring significant improvements in Scala (10-15x), but not that much in Truffle (2x). I am not exactly sure why that happens. That is an interesting question, that I plan to digg into later.

	Query 1	Query 2	Query 3
Pure Scala	239ms	167ms	210ms
Staged Scala	29ms	10ms	15ms
Truffle	298ms	203ms	362ms
Staged Truffle	147ms	103ms	178ms

Table 3: SQL interpretation benchmarks

4. Future Work

There are multiple ways in which the work done in this project can be improved and extended:

- Explore more optimizations from Truffle framowork/Graal VM. For example, Assumptions explained in [3], more agressive compilation and Truffle annotations [4].
- Right now all this is implemented with basic . Integration of this project as a fully compatible generation target for LMS would allow to run more examples from tutorials and perform better evaluations.
- During the course of this project only basic benchmarking was done. More extensive evaluation is needed on different examples and configurations.
- More explarations are needed to determine where exactly overhead comes from when using Truffle Nodes as a target for LMS generation. For now my assumption is that it comes from Truffle node dispatch, but better benchmarking should be helpful to understand that.

5. Conclusions

During the course fo this project I implemented Truffle node generation for Fast Fourier Transform, and for handwritten SQL interpretation. The implementation consists mostly of creating Truffle Nodes for different operations. Interesting part was to understand how LMS IR is generated, and the evalution step between stages. Also for me that was the hardest part, because it took me quite a while to understand how things work internally in LMS and why certain code is generated. Also, I have done different evaluations and measurments for Truffle vs Scala, JVM vs Graal, etc. While staging still results in better performance with Truffle Nodes, benefits of it are not as clear as with Scala for example. Still, Truffle Framework/Graal VM combines with LMS rather naturally and I think that with better optimizations more interesting results can be achieved.

References

[1] I. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. In *Proceedings of the VLDB Endowment*, volume 7, 2014.

[2] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *SIGPLAN Not.*, 46(2):127–136, Oct. 2010. . URL `http://doi.acm.org/10.1145/1942788.1868314`.

[3] 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, pages 2:1–2:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2916-3. . URL `http://doi.acm.org/10.1145/2617548.2617550`.

[4] 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, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. . URL `http://doi.acm.org/10.1145/2509578.2509581`.