TastyTruffle: A Subtitle

by

James You

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This is the abstract.

Acknowledgements

I would like to thank all the little people who made this thesis possible.

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Abbreviations

TASTy Typed Abstract Syntax Tree 5

Chapter 1

Introduction

Chapter 2

Background

This section should mainly explore type erasure and how it relates to the various sections below.

2.1 Intermediate Representations

2.1.1 Java Bytecode

2.1.2 Scala Typed Abstract Syntax Trees

2.1.3 GraalVM Intermediate Representation

2.2 Managed Runtimes

2.2.1 Type Erasure

2.2.2 Just-in-time Compilation

Chapter 3

Implementation

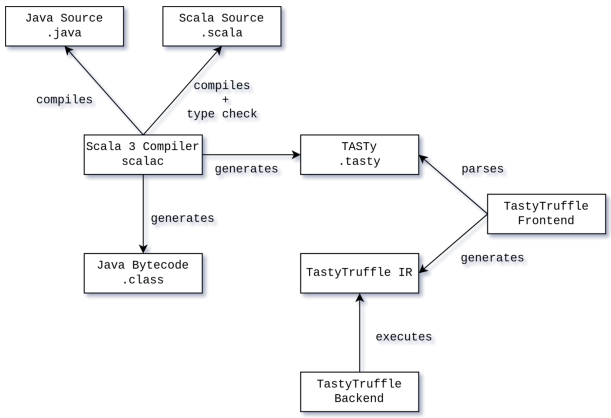

Figure 3.1: TastyTruffle in the context of the Scala compilation pipeline.

3.1 Case Study: A List in TastyTruffle

```
1   abstract class List[+T] {
2       def head: T
3       def tail: List[T]
4       def length: Int
5       def isEmpty: Boolean = length == 0
6       def contains[T1 >: T](elem: T1): Boolean
7   }
```

Figure 3.2: Definition of an abstract `List` class

```
1   case class ::[+T](head: T, tail: List[T]) extends List[T] {
2       override def length: Int = 1 + tail.length
3
4       override def contains[T1 >: T](elem: T1): Boolean = {
5           var these: List[T] = this
6           while (!these.isEmpty) {
7               if (these.head == elem) return true
8               these = these.tail
9           }
10          false
11      }
12
13      override def hashCode(): Int = {
14          var these: List[T] = this
15          var hashCode: Int = 0
16          while (!these.isEmpty) {
17              val headHash = these.head.hashCode()
18              if (these.tail.isEmpty) hashCode = hashCode | headHash
19              else hashCode = hashCode | headHash >> 8
20              these = these.tail
21          }
22          hashCode
23      }
24  }
25
26  case object Nil extends List[Nothing] {
27      override def head: Nothing = throw new NoSuchElementException("head of empty list")
28      override def tail: Nothing = throw new UnsupportedOperationException("tail of empty list")
29      override def length: Int = 0
30      override def contains[T1 >: Nothing](elem: T1): Boolean = false
31      override def hashCode(): Int = 0
32  }
```

Figure 3.3: Implementations of `List` class

3.2 TastyTruffle Intermediate Representation

Scala programs in TASTy format are unsuitable for execution in a Truffle interpreter. Programs in must be parsed and transformed into an executable representation in TastyTruffle. As TASTy represents a Scala program close to its equivalent source representation, canonicalization compiler passes that would otherwise normalize the IR are not present. Instead, we implement TastyTruffle IR to represent a canonicalized executable intermediate representation which can be specialized on demand.

The following sections will introduce the nodes in TastyTruffle IR and how they are derived from Scala source and TASTy.

Local Variables and Values

`(x: T)`

Control Flow

Object Allocation

Calls

```
class ApplyNode(sig: Signature, receiver: TermNode, args: Array[TermNode]) extends TermNode {

    final val INLINE_CACHE_SIZE: Int = 5;

    @Specialization(guards = "inst.type == tpe", limit = "INLINE_CACHE_SIZE")
    def cached(
        frame: VirtualFrame,
        inst: ClassInstance,
        @Cached("inst.type") tpe: Type,
        @Cached("create(resolveCall(instance, sig)") callNode: DirectCallNode
    ): Object = callNode.call(evalArgs(frame, inst));

    @Specialization(replaces = "cached")
    def virtual(
        frame: VirtualFrame,
        inst: ClassInstance,
        @Cached callNode: IndirectCallNode
    ): Object = {
        val callTarget = resolveCall(instance, sig);
        callNode.call(callTarget, evalArgs(frame, inst))
    }
}
```

Figure 3.4: Simplified implementation of the call node used in TastyTruffle.

Member Access

Types

```
new Foo
```

```
new Array[Int]
```

```
new Array[T]
```

3.3 Specialization

3.3.1 Specializing Terms

The basic polymorphic unit of code in Scala are terms whose types are derived directly from a type parameter `T` or indirectly from an applied type such as `Array[T]`.

3.3.2 Specializing Methods

Generic methods in Scala can be polymorphic under class type parameters, method type parameters, or both. In the latter two cases, polymorphic methods contain additional reified type parameters. In addition to the polymorphic terms present in the method body discussed in the previous section, the type of method term parameters may be polymorphic. The following components of a generic method must specialized:

- Polymorphic method parameters.
- Polymorphic terms inside the method body.

Code cannot be specialized any further after method specialization.

3.3.3 Method Parameters

3.3.4 Typed Dispatch

Polymorphic methods which contains type parameters have their underlying specialized implementations

```
class TypeDispatchNode(parent: RootNode) extends TermNode {

    type TypeArguments: Array[Type]
    @CompilerDirectives.CompilationFinal
    var cache: Map[TypeArguments, DirectCallNode]

    override def execute(frame: VirtualFrame): Object = {
        val types: TypeArguments = resolveTypeParameters(frame)
        dispatch(frame, args);
    }

    def dispatch(frame: VirtualFrame, types: TypeArguments): Object = cache.get(types) match {
        case Some(callNode) => callNode.call(frame.getArguments)
        case None => createAndDispatch(frame, types)
    }

    def createAndDispatch(frame: VirtualFrame, types: TypeArguments): Object = {
        CompilerDirectives.transferToInterpreterAndInvalidate()
        val specialization = parent.specialize(types)
        val callNode = DirectCallNode.create(specialization)
        cache = cache.updated(types, callNode)
        callNode.call(frame.getArguments)
    }
}
```

Figure 3.5: Simplified implementation of generic dispatch node based on reified type arguments.

3.3.5 Code Duplication

3.3.6 Partial Evaluation

3.4 Specializing Classes

3.4.1 Rewiring

Chapter 4

Evaluation

Chapter 5

Related Work

Chapter 6

Future Work

Chapter 7

Conclusions

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[2] Donald Knuth. *The TeXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

APPENDICES