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

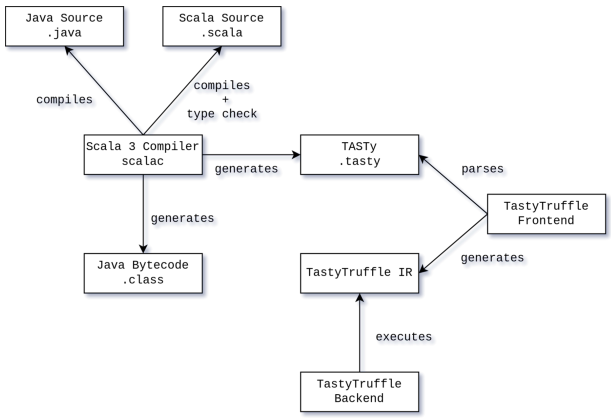

Figure 3.1: TastyTruffle in the context of the Scala compilation pipeline.

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

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 (see appendix A) 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

There are two distinct methods for declaring local variables in Scala:

```
1    // Constant reference, cannot be reassigned.
2    val constant: Int = 0
3    var variable: Int = 1
4    ...
5    variable = 2
```

Figure 3.4: `val` and `var` variable declarations

As the `val` abstraction is syntactic sugar[2] for the Scala compiler to validate that a `val` definition is never reassigned, TastyTruffle does not distinguish between `val` and `var` variable declarations. Each unique variable declaration has a corresponding frame slot in the frame descriptor of its root node. While Scala has lexical scoping[1], scope resolution occurs after pickling at a later stage in Scala compilation. As a result, symbols do not contain sufficient information to disambiguate between scopes. We consider variable declarations which share the same symbol but not the same `Block` trees to be unique in order to address this.

Truffle permits each frame slot in a frame descriptor be described by a *frame slot kind*. At the time of writing, a frame slot kind is given as:

```
    object FrameSlotKind extends Enumeration {
        type FrameSlotKind = Value
        val Object, Long, Int, Double, Float, Boolean, Byte = Value
    }
```

Figure 3.5: Simplified implementation of `FrameSlotKind`

We determine the frame slot kind of a type using the following method:

```
def getFrameSlotKind(tpe: Type): Option[FrameSlotKind] = {
    if (tpe.isMonomorphic && tpe.isPrimitive)
        Some(primitiveSlotKindOf(tpe))
    else if (tpe.isParameter)
        None
    else
        Some(FrameSlotKind.Object)
}
```

Figure 3.6: Pseudocode for determining the frame slot kind of a type.

Truffle specializes local variable access based on the variable's type during partial evaluation[3]. To simplify and eliminate the need to specialize read and writes of variables where types are monomorphic and statically refer to a primitive type, the primitive frame slot kind is matched in the frame descriptor. In all other cases, including when the type is not resolvable through a single type parameter, e.g. `val x: T`, we assign the frame slot the `Object` frame slot kind. We will defer discussion of variable declarations which have polymorphic types that cannot be resolved statically until section 3.3.

Terms

Binary Expressions

Member Access

Control Structures

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 3.7: Simplified implementation of the call node used in TastyTruffle.

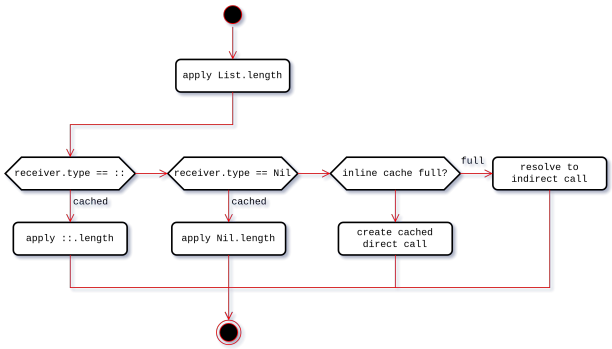

Figure 3.8: A flow diagram for the virtual dispatch of `List.length`

Object Allocation

Types

3.3 Specialization

```
1    trait PolymorphicTermNode extends TermNode {
2        def resolveType: ClassType
3        override def execute(frame: VirtualFrame): Object =
4            throw new UnsupportOperationException("generic code cannot be executed!")
5    }
```

Figure 3.9: A placeholder node for polymorphic code in TastyTruffle

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 a type constructor such as `Array[T]`. Polymorphic terms can be divided into the following categories:

Polymorphic local access

Polymorphic field access

Polymorphic method call

Polymorphic instantiation

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.

Method Parameters

Typed Dispatch

```
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 3.10: Simplified implementation of generic dispatch node based on reified type arguments.

Code Duplication

Partial Evaluation

3.4 Specializing Classes

Chapter 4

Evaluation

Chapter 5

Related Work

Chapter 6

Future Work

Chapter 7

Conclusions

References

[1] Scala Language Specification | Scala 2.13.

[2] Peter J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[3] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 662–676, New York, NY, USA, 2017. Association for Computing Machinery.

APPENDICES

Appendix A

Scala 3 Compiler Phases

```
/** Phases dealing with the frontend up to trees ready for TASTY pickling */
protected def frontendPhases: List[List[Phase]] =
    List(new Parser) ::                       // scanner, parser
    List(new TyperPhase) ::                   // namer, typer
    List(new YCheckPositions) ::              // YCheck positions
    List(new sbt.ExtractDependencies) ::      // Sends information on classes' dependencies to sbt via callbacks
    List(new semanticdb.ExtractSemanticDB) :: // Extract info into .semanticdb files
    List(new PostTyper) ::                    // Additional checks and cleanups after type checking
    List(new sjs.PrepJSInterop) ::            // Additional checks and transformations for Scala.js (Scala.js only)
    List(new Staging) ::                      // Check PCP, heal quoted types and expand macros
    List(new sbt.ExtractAPI) ::               // Sends a representation of the API of classes to sbt via callbacks
    List(new SetRootTree) ::                  // Set the `rootTreeOrProvider` on class symbols
    Nil
```

```
/** Phases dealing with TASTY tree pickling and unpickling */
protected def picklerPhases: List[List[Phase]] =
    List(new Pickler) ::            // Generate TASTY info
    List(new PickleQuotes) ::       // Turn quoted trees into explicit run-time data structures
    Nil
```

```
/** Phases dealing with the transformation from pickled trees to backend trees */
protected def transformPhases: List[List[Phase]] =
    List(new FirstTransform,          // Some transformations to put trees into a canonical form
    new CheckReentrant,         // Internal use only: Check that compiled program has no data races involving global vars
    new ElimPackagePrefixes,    // Eliminate references to package prefixes in Select nodes
    new CookComments,           // Cook the comments: expand variables, doc, etc.
    new CheckStatic,            // Check restrictions that apply to @static members
```

```
    new BetaReduce,             // Reduce closure applications
    new init.Checker) ::        // Check initialization of objects
    List(new ElimRepeated,           // Rewrite vararg parameters and arguments
    new ExpandSAMs,             // Expand single abstract method closures to anonymous classes
    new ProtectedAccessors,     // Add accessors for protected members
    new ExtensionMethods,       // Expand methods of value classes with extension methods
    new UncacheGivenAliases,    // Avoid caching RHS of simple parameterless given aliases
    new ByNameClosures,         // Expand arguments to by-name parameters to closures
    new HoistSuperArgs,         // Hoist complex arguments of supercalls to enclosing scope
    new SpecializeApplyMethods, // Adds specialized methods to FunctionN
    new RefChecks) ::           // Various checks mostly related to abstract members and overriding
    List(new ElimOpaque,             // Turn opaque into normal aliases
    new TryCatchPatterns,       // Compile cases in try/catch
    new PatternMatcher,         // Compile pattern matches
    new sjs.ExplicitJSClasses,  // Make all JS classes explicit (Scala.js only)
    new ExplicitOuter,          // Add accessors to outer classes from nested ones.
    new ExplicitSelf,           // Make references to non-trivial self types explicit as casts
    new ElimByName,             // Expand by-name parameter references
    new StringInterpolatorOpt) :: // Optimizes raw and s string interpolators by rewriting them to string concatentations
    List(new PruneErasedDefs,        // Drop erased definitions from scopes and simplify erased expressions
    new InlinePatterns,         // Remove placeholders of inlined patterns
    new VCInlineMethods,        // Inlines calls to value class methods
    new SeqLiterals,            // Express vararg arguments as arrays
    new InterceptedMethods,     // Special handling of `==`, `|=`, `getClass` methods
    new Getters,                // Replace non-private vals and vars with getter defs (fields are added later)
    new SpecializeFunctions,    // Specialized Function{0,1,2} by replacing super with specialized super
    new LiftTry,                // Put try expressions that might execute on non-empty stacks into their own methods
    new CollectNullableFields,  // Collect fields that can be nulled out after use in lazy initialization
    new ElimOuterSelect,        // Expand outer selections
    new ResolveSuper,           // Implement super accessors
    new FunctionXXLForwarders,  // Add forwarders for FunctionXXL apply method
    new ParamForwarding,        // Add forwarders for aliases of superclass parameters
    new TupleOptimizations,     // Optimize generic operations on tuples
    new LetOverApply,           // Lift blocks from receivers of applications
    new ArrayConstructors) ::   // Intercept creation of (non-generic) arrays and intrinsify.
    List(new Erasure) ::             // Rewrite types to JVM model, erasing all type parameters, abstract types and refine
    List(new ElimErasedValueType,    // Expand erased value types to their underlying implmementation types
    new PureStats,              // Remove pure stats from blocks
    new VCElideAllocations,     // Peep-hole optimization to eliminate unnecessary value class allocations
    new ArrayApply,             // Optimize `scala.Array.apply([....])` and `scala.Array.apply(..., [....])` into `[...]`
    new sjs.AddLocalJSFakeNews, // Adds fake new invocations to local JS classes in calls to `createLocalJSClass`
    new ElimPolyFunction,       // Rewrite PolyFunction subclasses to FunctionN subclasses
    new TailRec,                // Rewrite tail recursion to loops
    new CompleteJavaEnums,      // Fill in constructors for Java enums
    new Mixin,                  // Expand trait fields and trait initializers
    new LazyVals,               // Expand lazy vals
    new Memoize,                // Add private fields to getters and setters
    new NonLocalReturns,        // Expand non-local returns
    new CapturedVars) ::        // Represent vars captured by closures as heap objects
    List(new Constructors,           // Collect initialization code in primary constructors
    // Note: constructors changes decls in transformTemplate, no InfoTransformers should be added after it
    new Instrumentation) ::     // Count calls and allocations under -Yinstrument
    List(new LambdaLift,             // Lifts out nested functions to class scope, storing free variables in environments
    // Note: in this mini-phase block scopes are incorrect. No phases that rely on scopes should be here
    new ElimStaticThis,         // Replace `this` references to static objects by global identifiers
    new CountOuterAccesses) ::  // Identify outer accessors that can be dropped
    List(new DropOuterAccessors,     // Drop unused outer accessors
```

```
65          new Flatten,                // Lift all inner classes to package scope
66          new RenameLifted,           // Renames lifted classes to local numbering scheme
67          new TransformWildcards,     // Replace wildcards with default values
68          new MoveStatics,            // Move static methods from companion to the class itself
69          new ExpandPrivate,          // Widen private definitions accessed from nested classes
70          new RestoreScopes,          // Repair scopes rendered invalid by moving definitions in prior phases of the group
71          new SelectStatic,           // get rid of selects that would be compiled into GetStatic
72          new sjs.JUnitBootstrappers, // Generate JUnit-specific bootstrapper classes for Scala.js (not enabled by default)
73          new CollectSuperCalls) ::   // Find classes that are called with super
74      Nil
```

```
1  /** Generate the output of the compilation */
2  protected def backendPhases: List[List[Phase]] =
3      List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not enabled by default)
4      List(new GenBCode) ::             // Generate JVM bytecode
5      Nil
```
