

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

© James You 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	viii
List of Figures	ix
Abbreviations	xi
1 Introduction	1
2 Background	2
2.1 Scala	2
2.2 Case Study: A List in Scala	3
2.3 Typed Abstract Syntax Trees	5
2.3.1 Definitions	6
2.3.2 Terms	9
2.3.3 Types and Type Trees	10
2.4 Java Bytecode	11
2.5 Type Erasure	13
2.6 GraalVM	15
2.6.1 Graal	16
2.6.2 Truffle	16

3	Implementation	19
3.1	Execution	19
3.1.1	Converting the <code>DefDef</code> tree into a Truffle Root Node	20
3.1.2	Deriving Shapes from <code>ClassDef</code> trees	24
3.1.3	Creating Instances with the <code>New</code> Tree	26
3.1.4	Resolving <code>Apply</code> trees	27
3.1.5	Field Access with the <code>Select</code> Tree	31
3.1.6	Storing Values with the <code>Assign</code> Tree	31
3.1.7	Loop Nodes with the <code>While</code> Trees	32
3.1.8	The <code>Block</code> tree	32
3.1.9	Accessing Locals and Globals with <code>Ident</code> Tree	33
3.2	Specialization	34
3.2.1	Specializing Object Layout with <code>Applied</code> type trees	34
3.2.2	Specializing Call Sites with <code>TypeApply</code> trees	34
3.2.3	Specializing Terms	37
4	Evaluation	39
5	Related Work	41
5.1	Truffle Interpreters	41
5.2	Specializing Scala	41
5.3	Specializing Other Languages	41
6	Future Work	42
7	Conclusions	43
	References	44
	APPENDICES	50
A	Scala 3 Compiler Phases	51

List of Tables

List of Figures

2.1	Definition of <code>List</code> class	3
2.2	Implementation of <code>Cons</code> class	4
2.3	Implementation of <code>Nil</code> class	5
2.4	TASTy in the context of the Scala compilation pipeline.	6
2.5	Pseudocode class definitions for a subset of TASTy trees.	7
2.6	Simplified implementation of the <code>object Nil</code>	7
2.7	Tree structure for the definition of <code>List</code> . For brevity, we use <code>_</code> to represent inferred[21] type trees by the compiler.	8
2.8	Pseudocode class definitions for a subset of TASTy trees.	9
2.9	Pseudocode class definitions for a subset of TASTy type trees.	10
2.10	Pseudocode class definitions for a subset of TASTy type trees.	11
2.11	Java bytecode of <code>Cons.contains</code>	12
2.12	<code>Cons</code> class after type erasure	13
2.13	Example of autoboxing introduced for a list	14
2.14	GraalVM overview[23].	15
2.15	Adaptive optimization loop of GraalVM	17
2.16	Pseudocode for a Truffle node implementation of an equality which supports node rewriting.	18
3.1	Pseudocode to convert a <code>ClassDef</code> into a <code>ClassShape</code>	20
3.2	Pseudocode of a root node.	20

3.3	Defintion of a <code>DefDef</code> tree with names of less important members replaced with <code>_</code>	21
3.4	Simplified implementation of <code>FrameSlotKind</code>	22
3.5	Pseudocode for <code>DefDefNode</code> and <code>Parameter</code>	23
3.6	Pseudocode for parsing <code>DefDef</code> into <code>DefDefNode</code>	23
3.8	Pseudocodeto convert a <code>ClassDef</code> into a <code>ClassShape</code>	25
3.9	Pseudocode of the field property.	25
3.10	Pseudocode of a method signature.	26
3.11	Pseudocode of a <code>NewNode</code>	27
3.12	Pseudocode of parsing an <code>Apply</code> tree.	27
3.13	Simplified implementation of the call node with a polymorphic inline cache used in <code>TastyTruffle</code>	29
3.14	A possible polymorphic inline cache for a <code>List.contains</code> callsite.	30
3.15	Pseudocode of field read node with a polymorphic inline cache.	31
3.16	Pseudocode to parse an <code>Assign</code> tree.	32
3.17	Pseudocode for parsing <code>Block</code> into <code>BlockNode</code>	33
3.18	Pseudocode of a <code>BlockNode</code>	34
3.19	A placeholder node for polymorphic code in <code>TASTYTRUFFLE</code>	34
3.21	Simplified implementation of generic dispatch node based on reified type arguments.	36
3.22	The typed dispatch chain for a <code>List.contains</code> call site	37
3.23	Graal IR of <code>List.head</code> after field read of <code>head0</code> is specialized.	38
4.1	Code of the <code>ArrayBuffer</code> benchmark.	40

Abbreviations

AST Abstract Syntax Tree [5](#)

DSL Domain Specific Language [17](#), [29](#)

IR Intermediate Representation [5](#)

JIT Just-in-time [2](#), [15](#), [29](#)

JVM Java Virtual Machine [2](#), [22](#)

TASTy Typed Abstract Syntax Tree [2](#), [5](#), [19](#)

Chapter 1

Introduction

Chapter 2

Background

In this chapter, we will provide an introduction to the Scala programming language. We will showcase a running example that we will use for the remainder of this thesis which exhibits features commonly present in Scala programs. We will describe [Typed Abstract Syntax Tree \(TASTy\)](#), an intermediate storage format used for separate compilation[?] of Scala programs. We will introduce a critical transformation, type erasure, which alters Scala programs so that they may be executable on their default platform the [Java Virtual Machine \(JVM\)](#). We will detail GraalVM [Just-in-time \(JIT\)](#) compiler infrastructure, an alternative JVM implementation which we use to implement a runtime for Scala in this thesis.

2.1 Scala

Scala[43] is an object-oriented, generic and statically typed programming language. Scala uses a *pure* object-oriented programming model[29] and addresses many of the shortcomings[28] in other object-oriented programming languages. Scala can be still considered *Java-like* because of the interoperability between Java and Scala programs. Programs in Scala may contain generic definitions, allowing Scala programs to be composable and reusable[45]. While these features offer abstractions which facilitate the design of increasingly complex programs, there are significant challenges with their implementation. In the subsequent sections of this chapter, we will describe the challenges implementing these paradigms when manifested in the various intermediate representations of Scala. We first begin with an explanation of the relevant programming paradigms present in Scala:

Object-oriented Every value in Scala is an object and every operation is method invocation on an object. Every object in Scala is an instance of a *class* and their type is determined by its class. Classes[19] are a mechanism for defining state and behaviour for a group of objects.

Generic Classes in Scala may contain *type parameters* and such classes can be considered *polymorphic*[51]. Polymorphic classes may define behavior independent of their data, allowing them to be reused extensively for multiple types of data.

Statically typed Static typing is a discipline where the type information about a program is known *before* it is executed. In order for a Scala program to compile successfully, it must be *well-typed*. For our purposes, computation should always produce a value which has a type matching the type declared by the programmer to be considered well-typed. Classes are the primary syntactical mechanism for declaring types in Scala. The properties of classes such as state, in the form of fields, and behaviour, in the form of methods, must be well-typed. Similarly, the uses of these properties in other classes must also be well-typed.

2.2 Case Study: A List in Scala

In this section, we will introduce the running example that will be used for the remainder of this thesis and our motivations for its selection. Figures 2.1, 2.2 and 2.3 contain an abstract singly-linked list class and its two concrete subclass implementations. This set of `List` implementations represent probable real-world use cases as they are a scaled down and simplified version of the list implementation present in the Scala collections library. The `List` definition from the collections library is available by default to all Scala programs.

```
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 2.1: Definition of `List` class

Figure 2.1 is an example which showcases the paradigms discussed in the previous section that are also commonly present real-world Scala programs. Implementations which extend the abstract `List` class exhibit the object-oriented property of *inheritance*. The `List` class contains a mixture of polymorphic and non-polymorphic methods to showcase type specialization. The `head` method is class-polymorphic in that its type is derived from a class parameter and becomes specialized when the class is specialized. The `contains` method is method-polymorphic and must be specialized after the class is specialized.

```

1 case class Cons[+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.## // Compute hashcode
17      if (these.tail.isEmpty) hashCode = hashCode | headHash
18      else hashCode = hashCode | headHash >> 8
19      these = these.tail
20    }
21    hashCode
22  }
23 }

```

Figure 2.2: Implementation of `Cons` class

Figure 2.2 contains the implementation of a list node. The `Cons` implementation contains two polymorphic fields, `head` and `tail`. For specialization, how the `head` field fits into the storage layout of a `Cons` instance may differ between a `Cons[Int]` and a `Cons[String]`. On the other hand, the `tail` field does not have to differ between instances of `Cons[Int]` and `Cons[String]`.

Figure 2.3 contains the implementation of the empty list. We provide the implementation of this class for completeness.

```
1 case object Nil extends List[Nothing] {
2   override def head: Nothing = throw new NoSuchElementException("head of empty list")
3   override def tail: Nothing = throw new UnsupportedOperationException("tail of empty list")
4   override def length: Int = 0
5   override def contains[T1 >: Nothing](elem: T1): Boolean = false
6   override def hashCode(): Int = 0
7 }
```

Figure 2.3: Implementation of Nil class

2.3 Typed Abstract Syntax Trees

An [Intermediate Representation \(IR\)](#) is a structural abstraction representing a program during compilation or execution. Intermediate representations are more suitable for reasoning about a program than program source code. [IR](#) can be used for compilation[38], optimization[38][18], or execution[39][40].

[Typed Abstract Syntax Tree \(TASTy\)](#) is a high-level [Intermediate Representation \(IR\)](#) which is produced and emitted after the type checking phase (also called the typer) of the Scala compiler (see appendix A). Figure 2.4 gives an overview of TASTy generation in the context of the Scala compilation pipeline, note that TASTy is only generated for Scala program sources. TASTy is a well-typed variation of an [Abstract Syntax Tree \(AST\)](#). Abstract syntax trees are a commonly used intermediate representation which resemble the program source representation. TASTy can be considered a *complete* IR of a Scala program before compilation, unlike the other intermediate representations we will examine throughout this thesis. A complete IR is able to capture all information of the original Scala source program. We will expand on why complete intermediate representations are significant in section 2.5.

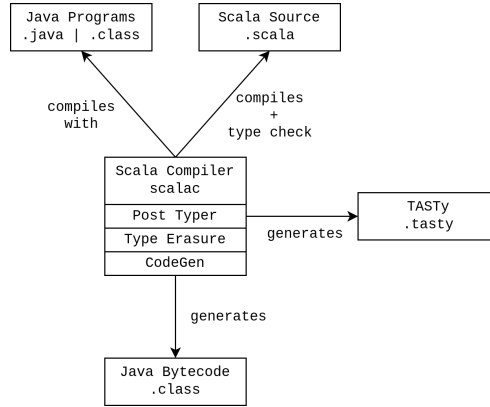


Figure 2.4: TASTy in the context of the Scala compilation pipeline.

The full TASTy IR can represent all Scala programs. The Truffle interpreter in this thesis supports a subset sufficient to express the the programs given in figures 2.1 and ???. The TASTy trees used in this thesis can be divided into categories, definitions, terms, and types. We give the pseudo implementations of these trees in figures: 2.5, 2.8, and 2.10.

2.3.1 Definitions

A Scala program consists of top level class definition which themselves contain statements. Statements either represent a declaration inside a class, such as method definitions, or executable code (or terms), which we discuss in section 2.3.2. Figure 2.5 provides the pseudo implementations of all definitions in our subset of TASTy. Every tree has a symbol, which is a unique reference to a definition. For the use cases in this thesis, most definitions can be translated and be represented by a corresponding implementation in Truffle. A **ClassDef** represents a top level class definition. A **DefDef** tree is the definition of a method inside a class definition.

A **ValDef** tree is a context-dependent definition which represents different value definition semantics depending on the tree it is defined in. A top level **ValDef**, that is a **ValDef** with no parent represents the **object** abstraction in Scala. The **object** abstraction is commonly used to represent the *Singleton* pattern[28] or as a class-like interface to define static methods. Consider the **Nil** class given in figure 2.3, a simplified TASTy-like equivalent would resemble the following:

A **ValDef** tree defined in the **body** of a **ClassDef** tree represent field definitions. A **ValDef** tree defined in the **TermParam** section of a **DefDef** tree represent parameter defi-

```

1  // Tree representing code written in the source
2  trait Tree {
3      def symbol: Symbol
4  }
5  trait Statement extends Tree      // Tree representing a statement in the source code
6  trait Definition extends Statement // Tree representing a definition in the source code.
7
8  // Tree representing a class definition.
9  case class ClassDef(
10     name:      String,
11     constructor: DefDef,
12     parents:    List[Tree],
13     self:       Option[ValDef],
14     body:       List[Statement]
15 ) extends Definition
16 // Tree representing a method definition in the source code
17 case class DefDef(
18     name:      String,
19     params:    List[ParamClause],
20     returnTpt: TypeTree,
21     rhs:       Option[Term]
22 ) extends Definition
23 // Tree representing a value definition in the source code.
24 case class ValDef(name: String, tpt: TypeTree, rhs: Option[Term]) extends Definition
25 // Tree representing a type (parameter or member) definition in the source code
26 case class TypeDef(name: String, rhs: Tree) extends Definition

```

Figure 2.5: Pseudocode class definitions for a subset of TASTy trees.

```

1  val Nil = new Nil$
2  class Nil$ extends List[Nothing] { ... }

```

Figure 2.6: Simplified implementation of the `object Nil`

nitions of the method. A `ValDef` tree defined among the statements in a `Block` tree is a local variable definition limited to the scope of the block.

Similarly, `TypeDef` trees refer to different kinds of definitions depending on their definition site. A `TypeDef` in the body of a `ClassDef` refers to a polymorphic class type parameter in our subset of TASTy. When a `TypeDef` is located in the `TypeParam` section a `DefDef` tree, it refers to a polymorphic method type parameter. The trees defined here can be used to represent more complex object-oriented and functional abstractions such as nested classes or closures, but they are beyond the scope of this thesis.

Figure 2.7 is the TASTy structure of the `List` class given in figure 2.1. Recall that `ClassDef` trees have four structural components, the constructor, the list of parent class definitions, the self type, and the body of the definition. In this thesis, we will not discuss the self type as it is an abstraction for composition[12][17] and is not relevant for execution. The list of parents in a class definition in our subset of TASTy is always a singleton. Note that while the abstract `List` class did not explicitly declare a constructor, the compiler autogenerates and inserts the appropriate constructor implementation before emitting TASTy. Since `List` is polymorphic, it contains an inner type definition of its sole type parameter. This distinction is what makes TASTy a complete IR when compared to the other intermediate representations we will describe later in this chapter.

```

1  ClassDef(
2      // name
3      "List",
4      // constructor
5      DefDef("<init>", List(TypeParams(TypeDef("T", TypeBoundsTree(_, _)), TermParams(Nil)), _, None)),
6      // parents
7      List(Apply(Select(New(_, "<init>"), Nil))),
8      // self
9      None,
10     // body
11     List(
12         TypeDef("T", TypeBoundsTree(_, _)),
13         DefDef("head", Nil, TypeIdent("T"), None),
14         DefDef("tail", Nil, Applied(TypeIdent("List"), List(TypeIdent("T"))), None),
15         DefDef("length", Nil, TypeIdent("Int"), None),
16         DefDef("isEmpty", Nil, TypeIdent("Boolean"), None),
17         DefDef(
18             "contains",
19             List(
20                 TypeParams(TypeDef("T1", TypeBoundsTree(TypeIdent("T"), _))),
21                 TermParams(ValDef("elem", TypeIdent("T1"), None))
22             ),
23             TypeIdent("Boolean"),
24             None
25         )
26     )
27 )

```

Figure 2.7: Tree structure for the definition of `List` . For brevity, we use `_` to represent inferred[21] type trees by the compiler.

Similarly, `DefDef` trees also retain their polymorphic properties. The parameters section of a `DefDef` tree is split into two halves. The type parameter section preserves any polymorphic type parameters in the method definition. The term parameter section con-

tains the normal value parameters found in a method. Term parameters may have types which are derived from the type parameter section.

2.3.2 Terms

```

1 // Tree representing an expression in the source code
2 trait Term extends Statement {
3   def tpe: Type
4 }
5 // Tree representing a reference to definition
6 trait Ref extends Term
7
8 // Tree representing an assignment lhs = rhs in the source code
9 case class Assign(lhs: Term, rhs: Term) extends Term
10 // Tree representing new in the source code
11 case class New(tpt: TypeTree) extends Term
12 // Tree representing a block `{ ... }` in the source code
13 case class Block(statements: List[Statement], expr: Term) extends Term
14 // Tree representing a while loop
15 case class While(cond: Term, body: Term) extends Term
16 // Tree representing a selection of definition with a given name on a given prefix
17 case class Select(qualifier: Term, selector: String) extends Term
18 // Tree representing an application of arguments.
19 case class Apply(applicator: Term, arguments: List[Term]) extends Term
20 // Tree representing an application of type arguments
21 case class TypeApply(fun: Term, args: List[TypeTree]) extends Term
22 // Tree representing a reference to definition with a given name
23 case class Ident(name: String) extends Ref

```

Figure 2.8: Pseudocode class definitions for a subset of TASTy trees.

Figure 2.8 gives the implementation for terms in our subset of TASTy. Terms represent executable atoms of code which return values. Terms can be considered analogous to expressions from the abstract syntax trees commonly used for other imperative programming languages. Our term tree subset of TASTy represents a basic language with support for simple imperative programming with control flow constructs such as branching and loops. A basic set of object-oriented features are also encapsulated in the tree definitions given above. The set of object-oriented features include object creation, instance method invocation, and instance field access. This subset of TASTY is sufficient to represent the creation of polymorphic classes as well as the invocation of polymorphic methods to showcase the examples described in this thesis.

Terms in TASTy also retain their types after type checking by the Scala compiler. A type for a term describes the type of the value produced by the term. Terms with

no children, such as `Ident` trees, are *explicit* typed. Childrenless terms have their type information encoded in a TASTy file. For terms with children, their types are derived from those of their children trees. Type information for non-leaf term trees is regenerated from term leaves when a TASTy file is read. In essence, types ‘flow’ upwards from leaf nodes in TASTy to their parent terms until the root term. The interpreter described in this thesis interprets a tree where the types for all trees have been regenerated. We will describe types in detail in the following section.

2.3.3 Types and Type Trees

TASTy encodes Scala programs with two kinds of type information, type trees and types. Type trees are a subset of trees which represent types as they are declared in Scala source code. On the other hand, types are the canonical representation of type trees after type checking in the Scala compiler. Multiple type trees may denote the same underlying type.

```

1 // Type tree representing a type written in the source
2 trait TypeTree extends Tree {
3   def tpe: Type
4 }
5
6 // Type tree representing a reference to definition with a given name
7 case class TypeIdent(name: String) extends TypeTree
8 // Type tree representing a type application
9 case class Applied(tpt: TypeTree, args: List[TypeTree | TypeBoundsTree]) extends TypeTree
10 // Type tree representing a type bound written in the source
11 case class TypeBoundsTree(lo: TypeTree, hi: TypeTree) extends TypeTree

```

Figure 2.9: Pseudocode class definitions for a subset of TASTy type trees.

Figure 2.9 gives the subset of type trees which we will use in this thesis. For our purposes, there are only three ways to refer to types. A `TypeIdent` type tree is a reference to a type which is a `ClassDef`. An `Applied` type tree represents a type constructor, which accepts type arguments and produces a new type. For example, the type `Cons[T]` would be represented as an applied type tree, where `Cons` would be the constructor and `T` would be the type argument. A `TypeBounds` tree represents the type expression `Lo <: T <: Hi`, a constraint where `T` must be a subtype of type `Hi` and supertype of type `Lo`. Type bounds are typically used to represent declared type parameter constraints, otherwise known as *bounded quantification*[\[13\]](#), in polymorphic classes or polymorphic methods. However, type bounds are also inserted by the `Typer` because type parameters in TASTy are universally

constraints. A type parameter `T` is expanded to `Nothing <: T <: Any`, that is the type parameter `T` must be a subtype of `Any` and a supertype of `Nothing`. In the context of this thesis, we can use subtype to mean *subclass of* and supertype to mean *superclass of*. Practically, this means the type parameter `T` has no constraints since `Any` is the super type of all types and `Nothing` is the subtype of all types.

```

1 trait Type                                // A type, type constructors, type bounds
2 trait NamedType extends Type              // Type of a reference to a type or term symbol
3 case class TypeRef extends NamedType     // Type of a reference to a type symbol
4 case class AppliedType extends Type      // A higher kinded type applied to some types T[U]
5 case class TypeBounds extends Type       // Type bounds

```

Figure 2.10: Pseudocode class definitions for a subset of TASTy type trees.

Figure 2.10 is set of types used in our subset of TASTy. In most cases in our subset of TASTy, the type trees have a corresponding type of the same name. However, the `NamedType` does not appear in type trees as they are predominantly used to type terms. The `TypeRef` type is a reference to a `ClassDef` tree or a type parameter `TypeDef`.

In the Scala compilation pipeline, TASTy is eventually simplified and transformed by the Scala compiler to produce Java bytecode. In chapter 3, We will go over each tree before such transformations and their relevance for execution in our interpreter .

2.4 Java Bytecode

Java bytecode is a portable and compact intermediate language and instruction set used by the Java Virtual Machine to execute programs. Java bytecode can be considered similar to an assembly language, where programs are represented as sequences of atomic instructions which manipulate a stack or registers. The type system in Java bytecode can describe primitive values such as `int` and references to objects such as `String`. As bytecode is intended to be simple for execution, it is not possible to represent polymorphic programs fully in Java bytecode.

Types in TASTy are not immediately compatible with types available in Java bytecode. Scala's type semantics must be eliminated from programs by the compiler before Java bytecode of the program can be emitted. The resulting Java bytecode is considered an *incomplete* IR of Scala source programs, as the type information found in the program source or inferred from compilation is no longer present. This becomes a particular

drawback for executing Scala programs on the JVM because speculative optimizations are unable to incorporate source level semantics.

```
1  aload_0
2  astore_2
3  aload_2
4  invokevirtual #44 // List.isEmpty:()Z
5  ifne        30
6  aload_2
7  invokevirtual #46 // List.head:()Ljava/lang/Object;
8  aload_1
9  invokestatic #52 // Method scala/runtime/BoxesRunTime.equals:(Ljava/lang/Object;Ljava/lang/Object;)Z
10 ifeq        22
11 iconst_1
12 ireturn
13 aload_2
14 invokevirtual #53 // List.tail:()LList;
15 astore_2
16 goto        2
17 iconst_0
18 ireturn
```

Figure 2.11: Java bytecode of `Cons.contains`

Figure 2.11 is the Java bytecode of the `contains` defined at line 4 in figure 2.2. Typical control flow elements of Scala programs such as `if` terms and `while` terms have been converted into branch and jump instructions. Notice that there are no polymorphic type parameters in the description of classes nor in the invocation of polymorphic methods present in the bytecode. In particular, notice the equality comparison in line 7 of figure 2.2 is actually a method invocation (instruction 14 in figure 2.11). As the Scala compiler is unable to determine the type of a polymorphic type parameter during compilation time, it is unable to select a Java bytecode instruction which implements polymorphic comparison. Instead, a bridge method part of the Scala standard library is responsible for handling polymorphic operations which operate on both reference and primitive types during runtime. In the next section, we describe the process which transforms Scala programs to a representation amenable for Java bytecode generation and the necessary additional runtime overhead associated with this transformation.

2.5 Type Erasure

Type erasure^[42] is a transformation which converts polymorphic classes and methods in Scala to monomorphic classes and methods. This conversion is necessary because the JVM does not support polymorphic classes during runtime. Erasure ensures that any given polymorphic class and method has a single representation in practice. Type erasure is a crucial part of the Scala compilation which renders TASTy incomplete. Figure 2.12 shows the `Cons` class after type erasure.

```
1 case class Cons(head: Any, tail: List) extends List {
2   override def length: Int = 1 + tail.length
3
4   override def contains(elem: Any): Boolean = {
5     var these: List = 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 = this
14    var hashCode: Int = 0
15    while (!these.isEmpty) {
16      val headHash = these.head.##
17      if (these.tail.isEmpty) hashCode |= headHash
18      else hashCode |= headHash >> 8
19      these = these.tail
20    }
21    hashCode
22  }
23 }
```

Figure 2.12: `Cons` class after type erasure

The polymorphic `Cons` class has all type parameters in its class definition *erased* and replaced by the `Any` type. The `Any` type is a Scala platform-independent^[43] abstract type representing the supertype of primitive and reference types. In Java bytecode, the `Any` type is compiled to the `Object` type, the supertype of all reference types on the JVM.

While type erasure simplifies classes for runtime, the Scala compiler must resolve the incompatibility of operations between primitive types and reference types on the JVM^[39]. In order for primitive types to have a uniform representation compatible with reference types, primitive types are encapsulated into corresponding boxed classes whose objects are

passed by reference. For example, `java.lang.Integer` is a class with an `Int` field. In a polymorphic context in which a type variable has been replaced by the reference type `Object`, an `Int` value is not passed directly, but by reference to an object of class `Integer` that contains the primitive value. The set of operations introduced by the compiler whenever a primitive value is accessed under a polymorphic context is known as *autoboxing*[1]. Autoboxing can be divided into two operations. *Boxing* occurs when a primitive value must be used where a polymorphic value is expected. *Unboxing* occurs when a polymorphic value must be used where a primitive value is expected. Figure 2.13 shows a simple example of inserted autoboxing operations when using the polymorphic `Cons` class after type erasure.

```

1 // Before type erasure
2 val lst: List[Int] = Cons(1, Nil)
3 val head: Int = lst.head
4 // After type erasure
5 val lst: List = Cons(box(1), Nil)
6 val head: Int = unbox(lst.head)

```

Figure 2.13: Example of autoboxing introduced for a list

The `head0` field inside the `Cons` class after erasure is no longer polymorphic and instead has the type `Any`. The integer value of `1` which is passed into the class constructor for the list is boxed and the primitive value is wrapped as an instance of its boxed class. Similarly, when the `head0` field of the instance is read and stored into a local variable, an unboxing operation occurs which extracts the primitive value out of its wrapper instance. In the Scala collections library, a set of commonly used polymorphic data structures, autoboxing operations are frequent and necessary. The computational overheads of autoboxing operations on programs which make substantial use of polymorphic collections, especially the Scala standard library, is significant[47]. The elimination of this overhead through optimizing autoboxing operations is one of the central goals of this thesis. In addition to this direct overhead, autoboxing is a significant indirect source of overhead which makes the analysis of programs using primitive values in a polymorphic context and thus inhibits many significant compiler optimizations.

2.6 GraalVM

GraalVM[57] is an implementation of a JVM. Traditionally, the JVM is responsible for the majority of the performance optimizations in Java programs[46] through **Just-in-time (JIT)** compilation. JIT compilation is an adaptive optimization which occurs during program execution. JIT compilation is concerned with optimizing and eliminating *hotspots* or portions of the program which are executed most frequently. JIT compilers[27][3] employ a range of *speculative* techniques to transform the program under optimization. Speculative optimizations use information collected during program execution, otherwise known as *profiling*. Assumptions are then made about gathered profiling data in order to generate high-performance native machine code. A key aspect of speculative optimizations using assumptions is that optimizations may be undone when their underlying assumptions are violated. This enables the JIT compiler to optimize programs without the need to statically prove assumptions hold in every execution path.

While other implementations of Java virtual machines were designed specifically for Java, GraalVM was designed from the onset to be *language-independent*. GraalVM can be divided into two major components of interest. The first is *Graal*, a language-agnostic JIT compilation infrastructure which handles speculative optimizations and generation of high-performance machine code. The second is *Truffle*, a framework for translating the semantics of a source language, also called a *guest language*, to take advantage of the Graal infrastructure.

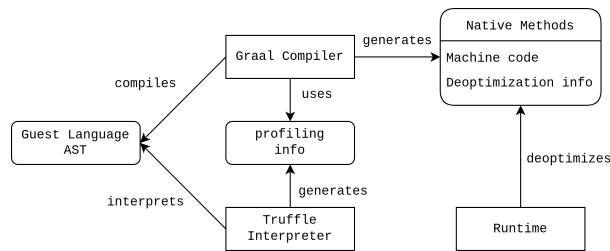


Figure 2.14: GraalVM overview[23].

This thesis makes substantial of both components of GraalVM to create a runtime for Scala programs using TASTy. The runtime is able to incorporate source level information for speculative optimizations.

2.6.1 Graal

GraalVM incorporates an existing implementation of a JVM[46] for the actual execution of programs. Graal is *only* the general-purpose just-in-time compilation infrastructure which optimizes the programs to be executed. Graal is general-purpose in that it conducts analysis and optimization on the same intermediate representation, *Graal IR*, regardless of the original source language. Notably, most implementations of a source language utilizing GraalVM have an implementation in Truffle, Java is an exceptional case. In addition to a Truffle interpreter for Java bytecode[?], there is a direct translator for Java programs in GraalVM which parses Java bytecode into Graal IR.

Graal IR[23]¹ is an IR which is suitable for speculative optimizations while still retaining information from the Truffle guest language AST. Graal IR is based on the *sea of nodes* concept[16] and satisfies the *static single-assignment*[18] property. A sea of nodes is an abstraction based on a directed graph structure which relates the control flow graph[7] of a program to its data flow graph[6]. An intermediate representation is in single-static assignment form when each variable is declared once and every use of a variable occurs immediately after its declaration[34].

GraalIR enables Graal to speculatively compile only the *hot* branches[24], or branches that are most frequently taken, in the control flow portion of the IR and their transitive data dependencies. When a compiled program violates any of its underlying assumptions, execution is *deoptimized*[32] and the program resumes execution in its uncompiled format. Deoptimization occurs when the compiled program is no longer considered stable and therefore is invalid. Graal automatically inserts *guard nodes* into the IR, which are conditional checks which validate that speculative assumptions used to compile the program still hold. Deoptimization is part of an execution loop between Graal and Truffle which allows GraalVM to aggressively adapt and speculate to find the best optimization in a dynamic execution environment.

2.6.2 Truffle

Truffle is a framework for implementing an interpreter embedded into GraalVM. Truffle differs significantly from other implementations of interpreters. Interpreters can usually be divided into two subsets: tree interpreters and bytecode interpreters. Tree interpreters transform program source into an abstract syntax tree which is then executed in post-order, children nodes are executed before their parents. Abstract syntax tree interpretation has

¹Given the number of intermediate representations introduced thus far, we promise this is the last one

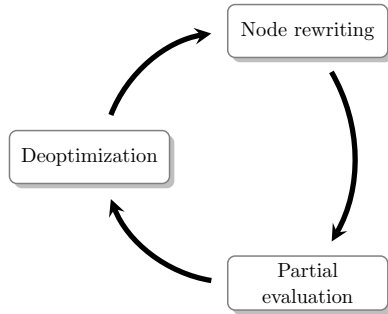


Figure 2.15: Adaptive optimization loop of GraalVM

the added benefit of executing an intermediate representation which is quite close to the program source representation and is therefore more amenable to program optimization. In contrast, bytecode interpreters such as the JVM, execute a vastly simplified representation of programs. While interpreters of bytecode programs tend to be faster than their tree counterparts, the absence of detailed source information such as types often makes program optimization difficult. The challenge of efficiently executing bytecode while retaining the ability to optimize them effectively using source program information is particularly difficult for Scala on the JVM.

Truffle is an atypical tree interpreter in that it combines the definition, execution, and optimization of an abstract syntax tree structure into a single abstraction. While the structure of input programs in other interpreters is independent of the implementation of the interpreter, a Truffle interpreter is integrated into the structure of its input. By defining execution semantics inside the abstract syntax tree to be executed, an interpreter is essentially derived from the implementation of its input tree structure. The execution semantics of the AST are additionally augmented with the Truffle [Domain Specific Language \(DSL\)](#), which allows such trees to be *self-optimizing*. The Truffle DSL is a mechanism to allow a *guest language* to embed semantics into a Truffle AST for optimization. A guest language is a set of semantics, most commonly a programming language, which is described by a Truffle AST. In this thesis, the guest language which our Truffle AST encodes and executes is TASTy (which represents Scala).

During execution of the AST, profiling information collected from the interpreter is used to drive *node rewriting*. While Graal is language-agnostic, Truffle is able to exploit guest language semantics for dynamic optimizations. This process of replacing nodes in the AST with better, specialized guest language counterparts in Truffle is called node rewriting. Node rewriting makes Truffle abstract syntax trees self-optimizing and serves two purposes. The first is to dynamically incorporate guest language semantics into the

executing program. The second is to augment the AST for more efficient JIT compilation. The nature of compiler optimizations require that programs are incrementally simplified in order to be optimized. While such types of optimizations are widely applicable to many languages using the JVM, node rewriting is a high-level language-specific optimization which occurs *before* such simplifications.

```

1 abstract class EqualsNode extends BinaryOpNode {
2     @Specialization
3     def equalsInt(lhs: Int, rhs: Int): Boolean = lhs == rhs
4
5     @Specialization(replaces="equalsInt")
6     def equals(lhs: Any, rhs: Any): Boolean = if (lhs == null) rhs == null else lhs.equals(rhs)
7 }

```

Figure 2.16: Pseudocode for a Truffle node implementation of an equality which supports node rewriting.

Figure 2.16 demonstrates an example of a node which can be rewritten. The node declares semantics of the equality operation between integers and values of type **Any**. This equality node has semantics for every type because the **Any** type is the super type of all types in Scala. A Truffle node which can be rewritten starts off in the uninitialized state. When both the left and right hand side operands are integers, the node is rewritten to `equalsInt` state. When arguments of any other combination of types are detected, either in the uninitialized state or the `equalsInt` state, the node is rewritten to the `equals` state.

After node rewriting, Graal JIT compiles Truffle ASTs into native machine code using *partial evaluation*. Partial evaluation is a program optimization technique for specializing a program (code) for a given input (data)[26]. In the context of Truffle, this means specializing an AST node (code) based on the values (or types of values) produced by their children nodes (data)[55]. We can say that the partial evaluation of an AST will produce an AST which is *specialized* for a particular set of values, or more commonly in our case, a particular set of types. If a frequently executed Truffle AST cannot be rewritten further, it is considered *stable* for JIT compilation into native machine code. The sequence of optimizations given in figure 2.15, node rewriting, partial evaluation into machine code, and deoptimization is the advantage that a TASTy Truffle interpreter has over the traditional JVM bytecode interpreter for Scala. Truffle allows for the incorporation of source-level type information into the just-in-time compilation loop. In this thesis, we will focus heavily on using node rewrites in the execution of TASTy with type information to augment JIT compilation.

Chapter 3

Implementation

This chapter is divided into two halves. The first half of this chapter will describe the methods used to execute TASTy in a Truffle interpreter *without* polymorphism. Section [3.1](#) will cover how to transform the semantics of TASTy to Truffle AST nodes. In particular, the first section focuses how to translate the organization of data and code in the `DefDef`, `ClassDef`, and `Term` trees into an Truffle implementation which is executable. The first half will omit the discussion of translating `TypeDef`, `TypeTree` and `Type` constructs in general. The second half of this chapter will discuss the implementation of parametric polymorphism in our interpreter and covers the techniques we use specialize nodes to eliminate autoboxing.

3.1 Execution

Scala programs in [TASTy](#) format are unsuitable for execution in a Truffle interpreter. Programs must be parsed and transformed into an executable representation in Truffle. This involves translating the TASTy tree structure into a simpler, but semantically equivalent Truffle AST. For the rest of this thesis, we refer to the Truffle AST of TASTy as *TastyTruffle IR*. 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.

In the following sections, we will describe the individual types of TASTy nodes and why some are directly unsuitable for execution and how to simplify their semantics for execution. We will begin with an explanation of how data is encoded and defined in TASTy.

```

1 def evaluate(tree: Tree): Object = tree match {
2   case vdef: ValDef =>
3     lazy val obj = initializeObject(vdef)
4     registerObject(vdef.symbol, obj)
5   case cdef: ClassDef =>
6     registerShape(cdef.tpe, parseClassDef(cdef))
7   case _ => ()
8 }
9
10 def newInstanceOf(tpe: Type): TruffleObject

```

Figure 3.1: Pseudocode to convert a `ClassDef` into a `ClassShape`.

3.1.1 Converting the `DefDef` tree into a Truffle Root Node

In this section, we describe the conversion of `DefDef` trees to *root nodes*. `DefDef` trees are the primary structure which organizes code (terms) in TASTY. Root nodes represent the root of an executable Truffle AST, the primary structure which organizes code in Truffle. In our case, root nodes are the Truffle analog of a `DefDef`. Each root node has a corresponding *call target*, which is used for invocation of the root node. Call targets are the primary compilation unit for Graal. A compilation unit is an organization of code which can be independently compiled. A root node is automatically instrumented^[49] to profile its number of invocations. When a root node has reached a limit on invocations inside the interpreter, it is JIT compiled into machine code by Graal. Subsequent invocations of the call target will then use the more efficient compiled root node.

```

1 abstract class RootNode(desc: FrameDescriptor) {
2   def execute(frame: VirtualFrame): Object
3   def getCallTarget: CallTarget
4 }

```

Figure 3.2: Pseudocode of a root node.

Figure 3.2 gives a simplified implementation of a root node. Each root node in Truffle has a *frame descriptor* and execution semantics. A guest language must subclass and implement its own root node in order to enable function invocation semantics.

A frame descriptor describes guest language variables which are in scope during execution. The abstract `execute` method describes the invocation behaviour of a root node.

When a root node is executed, it always supplied with a *frame*. A frame contains the arguments supplied during invocation and storage slots for local variable definitions in the body of the method. By default, all frames in execution are *virtual*. Virtual frames are an Truffle abstraction which provides guest languages an opportunity to exploit escape analysis. Escape analysis[35] reasons about the dynamic scope of object allocations. Truffle and Graal both exploit the observations of *Partial Escape Analysis*[50], a path-sensitive variant of escape analysis, to enable the following optimizations for guest languages:

Region Allocation[11][53] The substitution of heap allocations with stack allocations to eliminate unnecessary garbage collection.

Scalar Replacement[36] The complete elimination of an object allocation, where the fields of the replaced object are substituted by local variables.

The virtual frame abstraction allow guest languages to read and write normally without having to optimize their object allocations. Escape analysis occurs automatically during partial evaluation with no guest language intervention necessary.

```
1 class DefDef(_: String, params: List[ParamClause], _: TypeTree, rhs: Option[Term]) extends Definition
```

Figure 3.3: Defintion of a DefDef tree with names of less important members replaced with _

A further simplified definition of a DefDef tree is provided in figure 3.3. In this section, we focus on two members of a DefDef trees. The parameters of a DefDef tree are given by the `params` field. In practice, the type of a `ParamClause` is an alias for the union type `TypeParams | TermParams`, so we omit the `ParamClause` definition. A DefDef tree will have a parameter section for type parameters when they are polymorphic and will always have term parameters section. DefDef trees may optionally have a body defined in the `rhs` field. When trees do not have a body defined, they are abstract method definitions and do not have corresponding root node in Truffle. We will only consider non-abstract method definitions which have a body (a term) defined to be executable. We will cover the parsing of terms into nodes for execution in detail after section ??

Each value definition in the parameters of a DefDef will have a corresponding frame slot in its parent frame descriptor. A frame slot references a unique frame value in the context of a root node. Truffle permits each frame slot in a frame descriptor be described

```

1 object FrameSlotKind extends Enumeration {
2     type FrameSlotKind = Value
3     val Object, Long, Int, Double, Float, Boolean, Byte = Value
4 }
5
6 def getFrameSlotKind(tpe: Type): FrameSlotKind =
7     if (tpe.isPrimitive)
8         getPrimitiveSlotKind(tpe) // Int => FrameSlotKind.Int, ..., Double => FrameSlotKind.Double
9     else
10        FrameSlotKind.Object

```

Figure 3.4: Simplified implementation of `FrameSlotKind`

by a *frame slot kind*. In Truffle, there is a corresponding frame slot kind for reference types and each JVM primitive type. Pseudocode of a frame slot kind and a method to convert a type into a slot kind is given in 3.4.

Truffle profiles frame accesses in order to minimize the amount of autoboxing which occurs when reading from frame slot with an `Object` kind. To eliminate unnecessary specialization of frame accesses where types are monomorphic and statically refer to a primitive type, a parameter is assigned the matching primitive frame slot kind in the frame descriptor. In cases where the type is not a primitive type or a polymorphic applied type, e.g. `List[T]` but not `T`, we assign its frame slot the `Object` kind. Otherwise, the type is a polymorphic parameter which *could* resolve to a primitive type and the frame slot kind cannot be resolved statically. We will defer discussion on how to handle parameters of such polymorphic types that cannot be resolved statically until section 3.2.

Figure 3.5 provides the implementation of the `DefDefNode` and its parameters, the root node equivalent of a `DefDef`. The execution of a `DefDefNode` is divided into two stages, argument preparation and execution. First, the arguments of the frame constructed during invocation (see 3.1.4), are copied into their respective parameter frame slots. Frames contains separate regions for values of each frame slot kind. Based on the frame slot kind prescribed to a parameter, we copy each argument into the appropriate frame slot region. Storing parameters in this manner eliminates any unnecessary unboxing which would otherwise occur during a frame access. After arguments are copied into the frame, their values become available for access during the execution of the body. The body of a `DefDefNode` is then executed and its computed value returned .

Figure 3.6 provides a summary on parsing a `DefDef` tree into its Truffle equivalent `DefDefNode`. Frame slot and a frame slot kinds provide an abstraction for parameters and arguments to be resolved before the execution of the main body in a `DefDefNode`.

```

1 case class LocalVal(slot: FrameSlot, kind: FrameSlotKind)
2
3 class DefDefNode(desc: FrameDescriptor, params: Array[LocalVal], body: TermNode) extends RootNode(desc) {
4     override def execute(frame: VirtualFrame): Object = {
5         copyArgumentsToFrame(frame)
6         body.execute()
7     }
8
9     def copyArgumentsToFrame(frame: VirtualFrame): Unit =
10         for ((param, arg) <- params zip frame.getArguments)
11             param.kind match {
12                 case FrameSlotKind.Int =>
13                     frame.setInt(param.slot, arg.asInstanceOf[Int])
14                 ...
15                 case FrameSlotKind.Double =>
16                     frame.setDouble(param.slot, arg.asInstanceOf[Double])
17                 case _ =>
18                     frame.setObject(param.slot, arg)
19             }
20 }

```

Figure 3.5: Pseudocode for DefDefNode and Parameter

```

1 def parseDefDef(ddef: DefDef): DefDefNode = {
2     val desc = new FrameDescriptor
3     val parameters =
4         self :: ddef.params.map {
5             case vdef: ValDef => createParameter(valDef, desc)
6         }
7
8     val body = parse(definition.rhs)
9     new DefDefNode(desc, parameters, body)
10 }
11
12 def createParam(vdef: ValDef, desc: FrameDescriptor): LocalVal = {
13     val kind = getFrameSlotKind(vdef.tpt.tpe)
14     val slot = desc.addSlot(kind)
15     Parameter(slot, kind)
16 }

```

Figure 3.6: Pseudocode for parsing DefDef into DefDefNode

In addition to the parameters which are explicitly present in TASTY, the root node will have additional parameter which represents the receiver of the method. The receiver is an object instance whose class definition owns the method being invoked. In Scala, every method invocation has a receiver. In TASTy, this translates to every `DefDef` is owned by

a `ClassDef`. In the next section, we detail how to organize call targets in Truffle by using `ClassDef` trees.

3.1.2 Deriving Shapes from `ClassDef` trees

```

1 class ClassDef(
2     name:      String,
3     constructor: DefDef,
4     parents:    List[Tree],
5     _:         Option[ValDef],
6     body:       List[Statement]
7 ) extends Definition

```

(a) Pseudocode of a `ClassDef`.

```

1 class ClassShape(
2     symbol:     Symbol,
3     parents:    Array[Symbol],
4     fields:     Array[Field]
5     methods:    Map[MethodSignature, CallTarget]
6     vtable:     Map[MethodSignature, Symbol]
7 )

```

(b) Pseudocode of a shape for a `ClassDef`.

`ClassDef` tree define the layout of an object in TASTy. The layout of a object dictate the values which an object instance stores as well the methods which can be invoked on an object instance. The data layout of an object in a Truffle interpreter is described by a *shape* [15][56]. Shapes are a language-agnostic model for defining the properties of a object instance in Truffle. A property in a shape describes one member of an object instance; it has an identifier and a value. A Truffle object instance consists of *object storage*, which contains instance-specific data, and its shape. Shapes map property identifiers to object storage locations; guest languages interface with object storage indirectly through properties. In this thesis we use a *static shape*, an immutable variant of a shape. Normally, shapes are mutable and their list of properties may change throughout the lifetime of a program [20]. However, programs which dynamically change the layout of their objects [2] are out of the scope of this thesis.

Recall the definition of a `ClassDef` in figure 3.7a. Each `ClassDef` tree can be parsed into a corresponding `ClassShape`, given in Figure 3.7b. Figure 3.8 provides a very simplified implementation of the parsing steps to transform a `ClassDef` into a `ClassShape`. The `name` parameter of `ClassDef` alone is insufficient to be used as an identifier for a `ClassShape`. Names do not disambiguate between classes of the same name declared in different packages. Instead, we used the symbol of the `ClassDef` tree as the identifier for the `ClassShape`. For the remainder of this thesis, we will use a `ClassInstance` to refer to an object instance with properties described by a `ClassShape`.

A `ValDef` tree in the body of a `ClassDef` translates to a field definition in the `ClassShape`. A `ClassShape` has an collection of fields, which implement the static shape property. Fig-

```

1 def parseClassDef(cdef: ClassDef): ClassShape = {
2   val parents = cdef.parents.map(_.symbol)
3
4   val fields = cdef.body map {
5     case vdef: ValDef => generateField(vdef)
6   }
7
8   val methods = (cdef.constructor :: cdef.body) map {
9     case ddef: DefDef => ddef.symbol.signature -> parseDefDef(ddef)
10  }
11
12  val vtable = cdef.symbol.methodMembers map {
13    symbol => symbol.signature -> symbol
14  }
15
16  new ClassShape(cdef.symbol, parents, fields, init ++ methods, vtable)
17 }
18
19 def generateField(vdef: ValDef): Field = vdef match {
20   case ValDef(_: String, tpt: TypeTree, rhs: Option[Term]) => new Field(vdef.symbol, )
21 }

```

Figure 3.8: Pseudocode to convert a `ClassDef` into a `ClassShape`.

```

1 class Field(symbol: Symbol, tpe: Type) extends StaticProperty {
2   override def getId: String = symbol.name
3
4   def get(instance: Object): Any =
5     if (tpe == Int) getInt(instance)
6     else if ...
7     else if (tpe == Double) getDouble(instance)
8     else getObject(instance)
9
10
11   def set(instance: Object, value: Any): Unit =
12     if (tpe == Int) setInt(instance, value.asInstanceOf[Int])
13     else if ...
14     else if (tpe == Double) setDouble(instance, value.asInstanceOf[Double])
15     else setObject(instance, value)
16 }

```

Figure 3.9: Pseudocode of the field property.

ure 3.9 gives our implementation of a field. Fields define operations to read and write from the object storage on a `ClassInstance`. Like frames with frame slot kinds, object instances in Truffle have separate regions for storing values of each primitive type and one

for reference types. Following the same rules with types and frame slot kinds described in section 3.1.1, the data access of a field depends on the type of the `ValDef` tree from which the field originates. The remaining members of a `ClassShape` do not describe data which has to be stored in the object storage of a `ClassInstance`.

```
1 case class MethodSignature(symbol: Symbol, params: Int, types: Array[Type])
```

Figure 3.10: Pseudocode of a method signature.

After the constructor and the `DefDef` statements of a `ClassDef` are converted into root nodes, they are stored in the `ClassShape` mapped by a method signature. The pseudocode for a method signature is given in figure 3.10. Method signatures disambiguate method invocations in the presence of *ad hoc polymorphism* [51], where methods share the same name but have different arguments. When combined with parametric polymorphism, method signatures must also be able to disambiguate between methods sharing the same name but having different type parameters. However, method signatures do not have to disambiguate between different type parameters by name, only the number of type parameters that a method has. Because type erasure erases polymorphic type parameters from methods, methods which share the same number of parameters as well as the same arguments will conflict and therefore are invalid. As previously mentioned, methods are shared between all `ClassInstance` objects with the same shape, call targets are stored on the shape itself.

Often a shape will not contain the call target referenced by a signature because the dispatch is dynamic and the original type inherits the method. A `ClassShape` contains a *virtual method table*, which maps a method signature to the symbol of a shape which contains the call target matching the signature. If a method signature does not have a call target in the current shape, the shape which holds the target is indirectly resolved using the virtual method table during execution. While this resolution carries significant performance overhead both in Truffle and other implementations of programming languages, we will describe a technique which partially mitigates this overhead further on this half of chapter.

3.1.3 Creating Instances with the New Tree

The `New` tree represents the allocation of an instance of a `ClassDef`. The Truffle equivalent `allocate` node given in figure 3.11 is not so different, but it allocates an instance

```

1 class NewNode(symbol: Symbol) extends TermNode {
2   override def execute(frame: VirtualFrame): Object = shapeOf(symbol.tpe).newInstance
3 }

```

Figure 3.11: Pseudocode of a `NewNode`

with properties described by the `ClassShape` instead of a `ClassDef`. Note that a `NewNode` only *creates* an object; the parameters and fields of an object remain uninitialized. An object is *initialized* when the `<init>` method is invoked on a newly created object. TASTy is emitted with this sequence of events in mind, object creation is always followed by object initialization. Structurally, this means that a `New` tree is always the child of an initializer `Apply` tree.

3.1.4 Resolving Apply trees

```

1 def parseApply(apply: Apply): ApplyNode = {
2   val signature = apply.symbol.signature
3   apply match {
4     case Apply(Select(qualifier, _), arguments) =>
5       if (qualifier.tpe.isPrimitive)
6         if (args.length == 0) unaryOp(signature, qualifier)
7         else binaryOp(signature, qualifier, args(0))
8     else if (qualifier.tpe.isArray)
9       arrayOp(signature, qualifier, arguments)
10    else
11      new ApplyNode(signature, parse(qualifier), arguments.map(parse))
12  }
13 }

```

Figure 3.12: Pseudocode of parsing an `Apply` tree.

The `Apply` tree is a context-dependent tree which represents multiple types of operations. These operations are disambiguated by the types of their receiver. Figure 3.12 provides an overview on the transformations discussed in this section as pseudocode for parsing an `Apply` into TastyTruffle IR. We omit the implementations of `unaryOp`, `binaryOp`, `arrayOp` to remain concise; These methods generate the equivalent intrinsic but as a Truffle node. In the following subsections, we enumerate all possible semantics in our subset of TASTy:

Arithmetic and Logical Operators

In TASTy there are no unary and binary operators typically found in Java or other imperative languages. Unary and binary operators are actually an invocation of 0-argument (unary operator) or 1-argument (binary operator) method. For example, the following addition operator in Scala `1 + 2` is desugared to `1.+(2)`. That is, the binary operator `+` is represented as the invocation of the instance function `Int.+` on the receiver with value `1` and type `Int` with a single argument `2`. Normally in the Scala compilation pipeline, methods which operate on primitive types and have an equivalent bytecode instruction on the JVM[39] are replaced by those instructions in compiled program bytecode. This process of selecting efficient implementations for numerical or logical operations is commonly known as intrinsification. Similarly, TastyTruffle avoids implementing methods of primitive types with actual call semantics as primitive operations are frequently used and simplify optimization for Graal.

Array Access

The syntax for accessing array elements in Scala does not differ from the invocation of method on an array. In other imperative languages such as Java, the syntax for accessing arrays is commonly separate from the syntax of invoking a method. For example, the access `array(0)` is desugared to `array.apply(0)` once the program is emitted in TASTy.

Similar to unary and binary operators, the underlying implementation of array operations are intrinsified into JVM bytecode instructions where possible. However, using the bytecode provided in figure 2.11 as an analog, operations on polymorphic arrays **cannot** be intrinsified. Instead, polymorphic array operations are handled by functions in the Scala runtime library. The overhead of such operations are substantial and commonly represent the largest performance bottlenecks in array-bound programs. These costs are additionally abstracted from the user as they commonly arise when using array-backed collections from the Scala standard library.

To operate without specialization, the implementation of our interpreter also incorporates the same runtime code to handle polymorphic array operations. In the second half of this chapter, we will discuss the methods used to eliminate the runtime overhead of these polymorphic bridge methods.

Method Invocation

```

1
2 @NodeChild("receiver")
3 @NodeField("signature", MethodSignature.class)
4 class ApplyNode(@Children args: Array[TermNode]) extends TermNode {
5     final val INLINE_CACHE_SIZE: Int = 5;
6
7     @Specialization(guards = "inst.getShape == shape", limit = "INLINE_CACHE_SIZE")
8     def cached(
9         frame: VirtualFrame,
10        instance: ClassInstance,
11        @Cached("instance.getShape") shape: ClassShape,
12        @Cached("create(resolveCall(instance, signature)") callNode: DirectCallNode
13    ): Object = callNode.call(evalArgs(frame, instance));
14
15    @Specialization(replaces = "cached")
16    def virtual(
17        frame: VirtualFrame,
18        instance: ClassInstance,
19        @Cached callNode: IndirectCallNode
20    ): Object = {
21        val callTarget = resolveCall(instance.getShape, signature);
22        callNode.call(callTarget, evalArgs(frame, instance))
23    }
24 }

```

Figure 3.13: Simplified implementation of the call node with a polymorphic inline cache used in TastyTruffle.

Otherwise, the `Apply` tree actually encodes a ‘normal’ method invocation. Truffle provides two abstractions for call nodes, the *direct call node* is used when the call target can be statically resolved. In our subset of TASTy, this is the set of methods which have private or final modifiers[30] and class constructors. Otherwise, the Truffle *indirect call node* is used for calls where call targets must be dynamically resolved. Using indirect calls instead of direct calls comes with performance overhead as indirect call nodes cannot be inlined and inhibits Graal’s dynamic intraprocedural analyses. In this thesis, we describe a singular call node implementation for both statically and dynamically dispatched calls. In order to minimize the use of indirect call nodes, we take advantage of a polymorphic inline cache[31] to eliminate the overhead of resolving virtual calls for JIT compilation.

Figure 3.13 shows a simplified Truffle call node in TASTYTRUFFLE which implements a polymorphic inline cache. The Truffle DSL will then emit a cache which is searched linearly based on the type of receiver. When the type of receiver has not been seen in the inline cache, an additional cache entry is generated and appended to the cache for the next call. Because a polymorphic inline cache dispatches direct calls based on the type of the receiver

value seen, Graal is able to speculatively optimize the call site with the assumption that the receiver is always the same type. Furthermore, this allows the calls site to be inlined, allowing a feedback loop of intraprocedural optimizations[54][9] to propagate through the inlined tree. One important aspect to note is the size of an polymorphic inline cache must be kept reasonable such that the cost of searching the cache should not defeat the speedup afforded by using the cache. If the size of the cache exceeds the limit set, the call node is rewritten to use an indirect calls as the cost of inline cache lookups will outweigh the penalty of an indirect call.

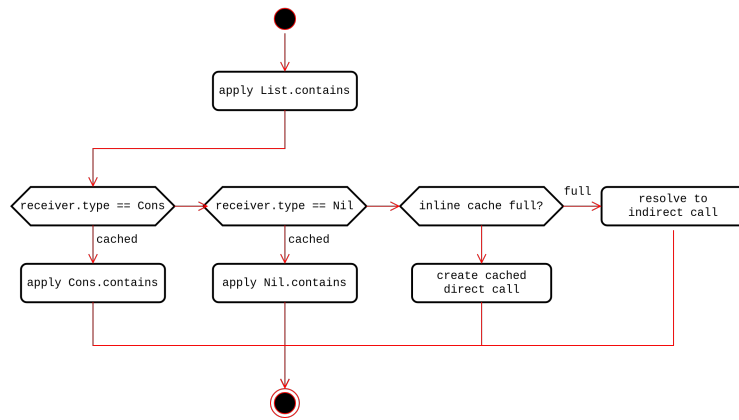


Figure 3.14: A possible polymorphic inline cache for a `List.contains` callsite.

Figure 3.14 shows a data flow diagram of the application of a polymorphic inline cache to a call site of `contains` when the receiver type statically known to be `List`. The diagram shown assumes that the call site has previously been called with a receiver where the dynamic type has been both `Cons` and `Nil`. The `ApplyNode` will first check if the type of receiver at the call site has the type `Cons`; If the check passes then the cached direct call node is invoked and the call is complete. It will then do the same for the type `Nil`. Otherwise, the type of the receiver has not been seen before and the call target is resolved virtually then cached for the next invocation at this call site.

When the polymorphic inline cache is applied to a monomorphic call site (where the type of the receiver does not change), it simplifies to a single element inline cache[22]. Because the type of the receiver at the call site remains stable, the cache look up of the call target based on the type always succeeds and the call site never fallbacks to using an indirect call node.

3.1.5 Field Access with the Select Tree

In our subset of TASTy, the **Select** tree represents a read of a field of a **ClassInstance**. Notice in the resolution of the **Apply** tree that an **Apply** tree represents a method invocation when the applicator is a **Select**. Because functions are first-class objects in Scala, the TASTy tree for a method invocation is the access of a method as if it were a field then the application of the subsequent function value read to a list of arguments. Since this case is handled previously when parsing the **Apply** tree, a **Select** tree always selects a field of an instance.

```
1 @NodeChild("receiver")
2 @NodeField("symbol", Symbol.class)
3 abstract class FieldReadNode extends TermNode {
4     final val INLINE_CACHE_SIZE: Int = 3;
5
6     @Specialization(guards = "instance.getShape == shape", limit = "INLINE_CACHE_SIZE")
7     def cached(
8         instance: ClassInstance,
9         @Cached("instance.getShape") shape: ClassShape,
10        @Cached("lookupField(shape)") field: Field
11    ): Object = field.getContents(instance)
12
13    @Specialization(replaces = "cached")
14    def virtual(instance: ClassInstance): Object = {
15        val field = lookupField(instance.getShape)
16        field.getContents(instance)
17    }
18
19    private def lookupField(shape: ClassShape): Field = shape.getField(symbol)
20 }
```

Figure 3.15: Pseudocode of field read node with a polymorphic inline cache.

Figure 3.15 gives a simplified implementation of a field read node. Like the virtual dispatch of call targets, fields are resolved dynamically with the shape of a **ClassInstance**. We apply a polymorphic inline cache to the lookup of field properties to eliminate the performance overhead associated with this kind of virtual dispatch.

3.1.6 Storing Values with the Assign Tree

The **Assign** tree has context-dependent semantics based on the structure of its left-hand side term. Figure 3.16 contains the simplified logic to resolve **Assign** trees into the

```

1 def parseAssign(assign: Assign): TermNode = assign match {
2   case Assign(select: Select, rhs) =>
3     new FieldWriteNode(parse(select.qualifier), select.symbol, parse(rhs))
4   case Assign(ident: Ident, rhs) =>
5     new LocalWriteNode(localOf(ident.symbol), parse(rhs))
6 }

```

Figure 3.16: Pseudocode to parse an `Assign` tree.

appropriate term nodes. If the left hand side term is a `Select` tree, the current tree mutates the field of a `ClassInstance`. Otherwise, the left hand side is an `Ident` which refers to local variable in the frame. We differentiate between which node to generate based on the type of the tree seen on the left hand side.

3.1.7 Loop Nodes with the While Trees

In our subset of TASTy, the `While` tree is the only looping construct. The control flow of the `While` tree is quite simple; the body term is executed as long as the condition term holds at the beginning of every iteration. Truffle provides the `RepeatingNode` abstraction for implementations of guest language loop structures. The repeating node abstraction allows guest languages to take advantage of *On-Stack Replacement*[\[25\]](#). On-stack replacement is a technique which switches control of part of a program running in the interpreter to compiled code while that part is executing.

The root node has been the primary compilation unit in JIT compilation discussed thus far. Root nodes profile their invocation count and get JIT compiled when they’ve been called a predetermined number of times. However, loop constructs which are executed for many iterations also justify JIT compilation. The repeating node is an additional type of JIT compilation unit which Graal can compile. A key difference between repeating node and a call target is when their compiled equivalents are utilized. While compiled root nodes are used in subsequent invocations of their call targets after they are JIT compiled, compiled repeating nodes are used in the next iteration after they are JIT compiled.

3.1.8 The Block tree

In this section, we cover the translation of the `Block` tree to its TastyTruffle IR equivalent. The `Block` is unique among term trees as it describes data as well as code. In our

```

1  def parseBlock(block: Block): BlockNode = {
2      val desc = getParentFrameDescriptor(block)
3
4      val terms = block.statements map {
5          case vdef: ValDef => generateBlockLocal(desc, vdef)
6          case term => term
7      }
8
9      new BlockNode(terms, parse(block.expr))
10 }
11
12 def generateBlockLocal(desc: FrameDescriptor, vdef: ValDef): TermNode = {
13     val local = generateLocal(vdef)
14     new WriteLocalNode(local, parse(vdef.rhs))
15 }
16

```

Figure 3.17: Pseudocode for parsing `Block` into `BlockNode`

subset of TASTy, this means that a block may contain declarations of local variables as well as executable terms. Figure 3.17 provides an overview on the transformations necessary to convert a `Block` tree into `BlockNode`. We divide the discussion of blocks into the resolution of local variables when encountering a `ValDef` tree and the execution of all other trees.

Local variables are variables which are bound to a *scope*. A scope represents the lifetime in which a variable can refer to an value. Similarly, uses of variables are only valid when used under the appropriate scope. Local variables and their use sites are represented in intermediate representations through a myriad of methods. In abstract syntax trees, local variables and their used are represented as nodes *dominated* by their scopes (which are themselves nodes). In our subset of TASTy, a `ValDef` dominated by a `Block` represents a local variable. When a `ValDef` tree is present in this context, the right hand side of the value definition will be non-empty.

Because terms always return a value, the `Block` tree must follow the same semantics. Figure 3.18 gives the pseudocode for our implementation of a `BlockNode`. The `@ExplodeLoop` is a Truffle DSL directive which guides Graal to unroll[8] the loop during partial evaluation at line 4.

3.1.9 Accessing Locals and Globals with Ident Tree

```

1 class BlockNode(stats: Array[TermNode], last: TermNode) extends TermNode {
2   @ExplodeLoop
3   override def execute(frame: VirtualFrame): Object = {
4     for (stat <- stats)
5       stat.execute(frame)
6
7     last.execute(frame)
8   }
9 }

```

Figure 3.18: Pseudocode of a BlockNode

3.2 Specialization

3.2.1 Specializing Object Layout with Applied type trees

```

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

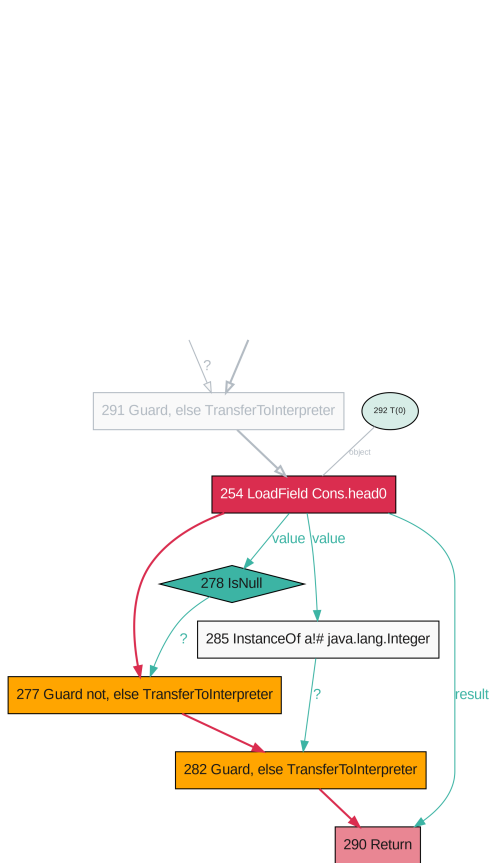
```

Figure 3.19: A placeholder node for polymorphic code in TASTYTRUFFLE

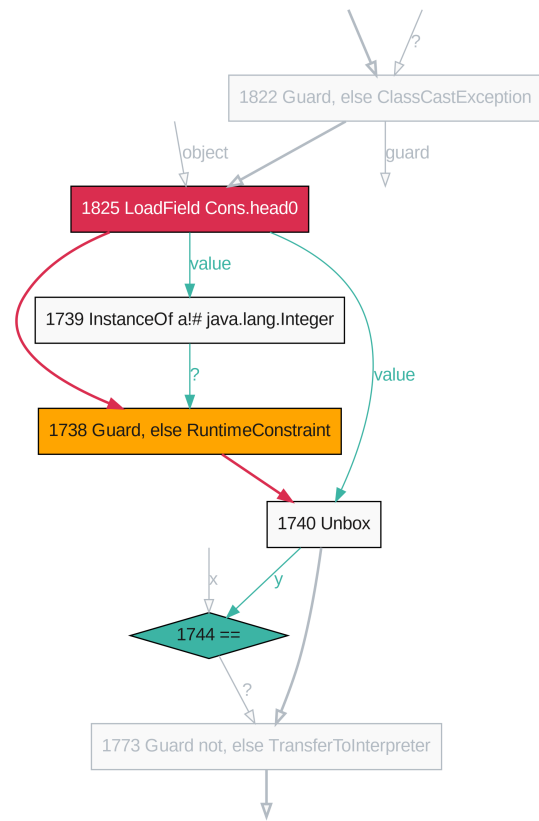
3.2.2 Specializing Call Sites with TypeApply trees

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 be specialized:

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



(a) Graal IR of `Cons.head` focused on field access of `head0`



(b) Graal IR of `Cons.head` after being inlined into `Cons.contains`

Method Parameters

Typed Dispatch Chains

Dispatch chains[?]

```
1 class TypeDispatchNode(parent: RootNode, params: Array[TypeNode]) extends TermNode {
2
3   type TypeArguments: Array[Type]
4   @CompilerDirectives.CompilationFinal
5   var cache: Map[TypeArguments, DirectCallNode]
6
7   override def execute(frame: VirtualFrame): Object = {
8     val types: TypeArguments = params.map(_.resolve(frame))
9     dispatch(frame, args);
10  }
11
12  def dispatch(frame: VirtualFrame, types: TypeArguments): Object = cache.get(types) match {
13    case Some(callNode) => callNode.call(frame.getArguments)
14    case None           => createAndDispatch(frame, types)
15  }
16
17  def createAndDispatch(frame: VirtualFrame, types: TypeArguments): Object = {
18    CompilerDirectives.transferToInterpreterAndInvalidate()
19    val specialization = parent.specialize(types)
20    val callNode = DirectCallNode.create(specialization)
21    cache = cache.updated(types, callNode)
22    callNode.call(frame.getArguments)
23  }
24 }
```

Figure 3.21: Simplified implementation of generic dispatch node based on reified type arguments.

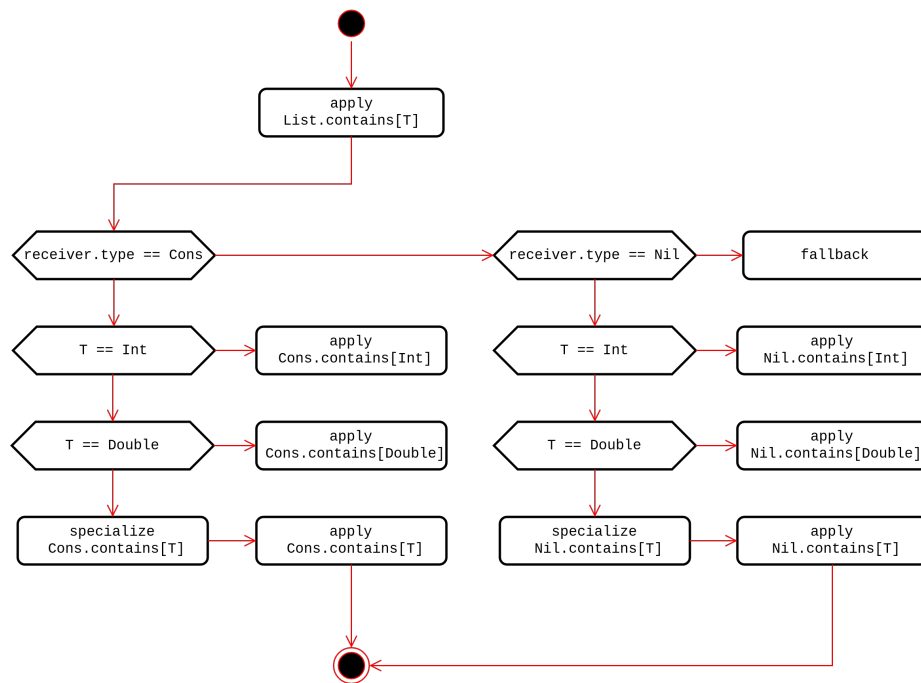


Figure 3.22: The typed dispatch chain for a `List.contains` call site

Code Duplication

Partial Evaluation

3.2.3 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

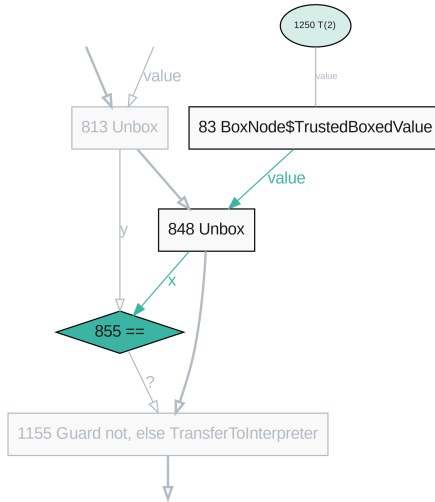
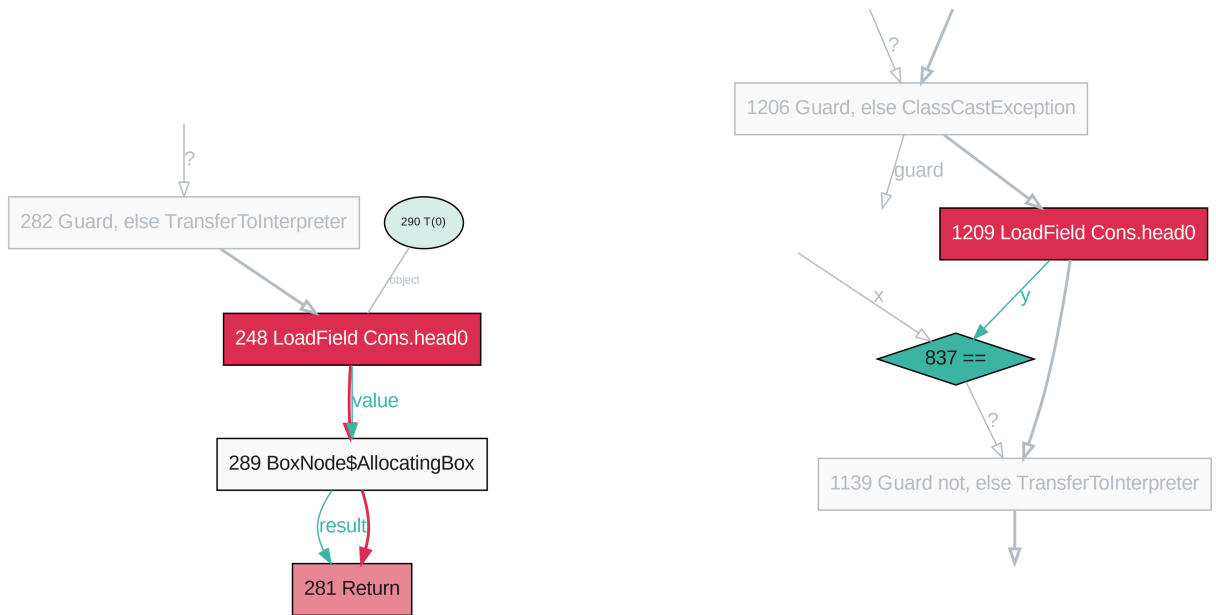


Figure 3.23: Graal IR of `List.head` after field read of `head0` is specialized.



(a) Graal IR of `List.head` after field read of `head0` is specialized.

(b) Graal IR of `Cons.head` after being inlined into `Cons.contains`

Chapter 4

Evaluation

```

1 class ArrayBuffer[T] {
2     protected def initialSize: Int = 16
3     var size0 = 0
4     var array: Array[T] = newArray[T](Math.max(initialSize, 1))
5
6     def length: Int = size0
7
8     private def get(i: Int): T = array(i)
9     private def set(i: Int, elem: T): Unit = array(i) = elem
10
11     def contains(elem: T): Boolean = {
12         var i = 0
13         while (i < size0) {
14             if (array(i) == elem) return true
15             i += 1
16         }
17         false
18     }
19
20     def reverse(): Unit = {
21         var pos = 0
22         while (pos * 2 < size0) {
23             swap(pos, size0 - pos - 1) // swaps two elements in the array
24             pos += 1
25         }
26     }
27
28     def append(elem: T): Unit = {
29         val newSize0 = size0 + 1
30         ensureSize(newSize0)
31         set(size0, elem)
32         size0 = newSize0
33     }
34
35     // Ensure that the internal array has at least `n` cells.
36     def ensureSize(n: Int): Unit = {
37         val arrayLength: Long = array.length // Use a Long to prevent overflows
38         if (n > arrayLength) {
39             var newSize: Long = arrayLength * 2
40             while (n > newSize)
41                 newSize = newSize * 2
42             // Clamp newSize to Int.MaxValue
43             if (newSize > lang.Int.MaxValue) newSize = lang.Int.MaxValue
44
45             val resized = newArray[T](newSize.toInt)
46             var i = 0
47             while (i < size0) {
48                 resized(i) = get(i)
49                 i += 1
50             }
51             array = resized
52         }
53     }

```

Figure 4.1: Code of the ArrayBuffer benchmark.

Chapter 5

Related Work

5.1 Truffle Interpreters

5.2 Specializing Scala

5.3 Specializing Other Languages

Chapter 6

Future Work

Chapter 7

Conclusions

References

- [1] Autoboxing and Unboxing (The Java™ Tutorials > Learning the Java Language > Numbers and Strings).
- [2] Using Java Reflection.
- [3] IBM Research | Technical Paper Search | The Jikes RVM Project: Building an Open Source Research Community(Search Reports), September 2016.
- [4] Alfred V Aho, Jeffrey D Ullman, et al. *Principles of compiler design*. Addison-Wesley Pub. Co., 1977.
- [5] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 174–185, New York, NY, USA, 1995. Association for Computing Machinery.
- [6] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, mar 1976.
- [7] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA, 1970. Association for Computing Machinery.
- [8] John R. Allen and Ken Kennedy. Automatic loop interchange. *ACM SIGPLAN Notices*, 19(6):233–246, June 1984.
- [9] B. Alpern, M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88*, 1988.

- [10] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 171–183, New York, NY, USA, 1996. Association for Computing Machinery.
- [11] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, nov 2003.
- [12] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, page 303–311, New York, NY, USA, 1990. Association for Computing Machinery.
- [13] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pages 750–770, Berlin, Heidelberg, 1991. Springer.
- [14] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, may 1995.
- [15] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '89, page 49–70, New York, NY, USA, 1989. Association for Computing Machinery.
- [16] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.
- [17] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *International Symposium on Mathematical Foundations of Computer Science*, pages 1–23. Springer, 2006.
- [18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.
- [19] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

- [20] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and thread-safe objects for dynamically-typed languages. 11 2016.
- [21] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.
- [22] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, page 297–302, New York, NY, USA, 1984. Association for Computing Machinery.
- [23] Gilles Duboscq, Lukas Stadler, Thomas Wuerthingner, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. February 2013.
- [24] Gilles Duboscq, Thomas Würthingner, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM workshop on Virtual Machines and Intermediate Languages - VMIL '13*, pages 1–10, Indianapolis, Indiana, USA, 2013. ACM Press.
- [25] S.J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 241–252, March 2003.
- [26] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
- [27] Etienne M Gagnon and Laurie J Hendren. Sable vm: A research framework for the efficient execution of java bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40, 2001.
- [28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.
- [29] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

- [30] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison-Wesley Professional, 2000.
- [31] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
- [32] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 32–43, New York, NY, USA, 1992. Association for Computing Machinery.
- [33] Christian Humer. *Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters*. PhD thesis, Johannes Kepler University Linz, Linz, Austria, 2016. Publisher: Unpublished.
- [34] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI '93*, 1993.
- [35] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, page 111–120, New York, NY, USA, 2005. Association for Computing Machinery.
- [36] Thomas Kotzmann and Hanspeter Mossenbock. Run-time support for optimizations based on escape analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, page 49–60, USA, 2007. IEEE Computer Society.
- [37] Peter J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.
- [38] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.
- [39] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition: Java Virt Mach Spec Java_3*. Addison-Wesley, 2013.

- [40] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29(7), 2001.
- [41] R. Milner, L. Morris, and M. Newey. A logic for computable functions with reflexive and polymorphic types. In *Proceedings of the Conference on Proving and Improving Programs*, pages 371–394. IRIA-Laboria, 1975.
- [42] Maurice Naftalin and Philip Wadler. *Java Generics and Collections: Speed Up the Java Development Process.* ” O’Reilly Media, Inc.”, 2006.
- [43] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. 2004.
- [44] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.
- [45] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 41–57, 2005.
- [46] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.
- [47] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive jit compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, page 29–40, New York, NY, USA, 2017. Association for Computing Machinery.
- [48] Ben Sander and AMD SENIOR FELLOW. Hsail: Portable compiler ir for hsa. In *Hot Chips Symposium*, volume 2013, pages 1–32, 2013.
- [49] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI ’94, page 196–205, New York, NY, USA, 1994. Association for Computing Machinery.

- [50] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 165–174, New York, NY, USA, 2014. Association for Computing Machinery.
- [51] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.
- [52] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
- [53] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
- [54] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.
- [55] 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.
- [56] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter 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*, pages 133–144, Cracow, Poland, 2014. ACM Press.
- [57] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New ideas, New Paradigms, and Reflections on Programming & Software - Onward! '13*, pages 187–204, Indianapolis, Indiana, USA, 2013. ACM Press.

APPENDICES

Appendix A

Scala 3 Compiler Phases

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

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

```
1  /** Phases dealing with the transformation from pickled trees to backend trees */
2  protected def transformPhases: List[List[Phase]] =
3      List(
4          new FirstTransform,                            // Some transformations to put trees into a canonical form
5          new CheckReentrant,                            // Internal use only: Check that compiled program has no data races involving global v
6          new ElimPackagePrefixes,                      // Eliminate references to package prefixes in Select nodes
7          new CookComments,                             // Cook the comments: expand variables, doc, etc.
8      )
```

```

8      new CheckStatic,           // Check restrictions that apply to @static members
9      new BetaReduce,           // Reduce closure applications
10     new init.Checker) ::       // Check initialization of objects
11   List(
12     new ElimRepeated,          // Rewrite vararg parameters and arguments
13     new ExpandSAMs,            // Expand single abstract method closures to anonymous classes
14     new ProtectedAccessors,    // Add accessors for protected members
15     new ExtensionMethods,      // Expand methods of value classes with extension methods
16     new UncacheGivenAliases,   // Avoid caching RHS of simple parameterless given aliases
17     new ByNameClosures,        // Expand arguments to by-name parameters to closures
18     new HoistSuperArgs,        // Hoist complex arguments of supercalls to enclosing scope
19     new SpecializeApplyMethods, // Adds specialized methods to FunctionN
20     new RefChecks) ::          // Various checks mostly related to abstract members and overriding
21   List(
22     // Turn opaque into normal aliases
23     new ElimOpaque,
24     // Compile cases in try/catch
25     new TryCatchPatterns,
26     // Compile pattern matches
27     new PatternMatcher,
28     // Make all JS classes explicit (Scala.js only)
29     new sjs.ExplicitJSClasses,
30     // Add accessors to outer classes from nested ones.
31     new ExplicitOuter,
32     // Make references to non-trivial self types explicit as casts
33     new ExplicitSelf,
34     // Expand by-name parameter references
35     new ElimByName,
36     // Optimizes raw and s string interpolators by rewriting them to string concatenations
37     new StringInterpolatorOpt) ::
38   List(
39     new PruneErasedDefs,        // Drop erased definitions from scopes and simplify erased expressions
40     new InlinePatterns,         // Remove placeholders of inlined patterns
41     new VCIInlineMethods,       // Inlines calls to value class methods
42     new SeqLiterals,            // Express vararg arguments as arrays
43     new InterceptedMethods,     // Special handling of `==`, `!=`, `getClass` methods
44     new Getters,                // Replace non-private vals and vars with getter defs (fields are added later)
45     new SpecializeFunctions,    // Specialized Function{0,1,2} by replacing super with specialized super
46     new LiftTry,                // Put try expressions that might execute on non-empty stacks into their own methods
47     new CollectNullableFields,   // Collect fields that can be nulled out after use in lazy initialization
48     new ElimOuterSelect,        // Expand outer selections
49     new ResolveSuper,           // Implement super accessors
50     new FunctionXXLForwarders,  // Add forwarders for FunctionXXL apply method
51     new ParamForwarding,        // Add forwarders for aliases of superclass parameters
52     new TupleOptimizations,     // Optimize generic operations on tuples
53     new LetOverApply,           // Lift blocks from receivers of applications
54     new ArrayConstructors) ::   // Intercept creation of (non-generic) arrays and intrinsify.
55   List(new Erasure) ::         // Rewrite types to JVM model, erasing all type parameters, abstract types and refinements
56   List(
57     new ElimErasedValueType,    // Expand erased value types to their underlying implementation types
58     new PureStats,              // Remove pure stats from blocks
59     new VCElideAllocations,     // Peep-hole optimization to eliminate unnecessary value class allocations
60     new ArrayApply,             // Optimize `scala.Array.apply([...])` and `scala.Array.apply(..., [...])` into `[...]`
61     new sjs.AddLocalJSFakeNews, // Adds fake new invocations to local JS classes in calls to `createLocalJSClass`
62     new ElimPolyFunction,       // Rewrite PolyFunction subclasses to FunctionN subclasses
63     new TailRec,                // Rewrite tail recursion to loops
64     new CompleteJavaEnums,      // Fill in constructors for Java enums

```

```

65     new Mixin,                // Expand trait fields and trait initializers
66     // Expand lazy vals
67     new LazyVals,
68     // Add private fields to getters and setters
69     new Memoize,
70     // Expand non-local returns
71     new NonLocalReturns,
72     // Represent vars captured by closures as heap objects
73     new CapturedVars) ::
74 List(
75     new Constructors,        // Collect initialization code in primary constructors
76     // Note: constructors changes decls in transformTemplate, no InfoTransformers should be added after it
77     new Instrumentation) :: // Count calls and allocations under -Yinstrument
78 List(
79     // Lifts out nested functions to class scope, storing free variables in environments
80     new LambdaLift,
81     // Note: in this mini-phase block scopes are incorrect. No phases that rely on scopes should be here
82     // Replace `this` references to static objects by global identifiers
83     new ElimStaticThis,
84     // Identify outer accessors that can be dropped
85     new CountOuterAccesses) ::
86 List(
87     // Drop unused outer accessors
88     new DropOuterAccessors,
89     // Lift all inner classes to package scope
90     new Flatten,
91     // Renames lifted classes to local numbering scheme
92     new RenameLifted,
93     // Replace wildcards with default values
94     new TransformWildcards,
95     // Move static methods from companion to the class itself
96     new MoveStatics,
97     // Widen private definitions accessed from nested classes
98     new ExpandPrivate,
99     // Repair scopes rendered invalid by moving definitions in prior phases of the group
100    new RestoreScopes,
101    // get rid of selects that would be compiled into GetStatic
102    new SelectStatic,
103    // Generate JUnit-specific bootstrapper classes for Scala.js (not enabled by default)
104    new sjs.JUnitBootstrappers,
105    // Find classes that are called with super
106    new CollectSuperCalls) ::
107 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

```
