Specializing Scala with Truffle

by

James You

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

Waterloo, Ontario, Canada, 2022

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

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

Abstract

Scala is a generic object-oriented programming language with higher-order abstractions. Programming abstractions in Scala exemplify reusability and extensibility in the context of type safety. Generic programming in particular allows user-defined data structures to behave identically irrespective of the types of their values while remaining free of type errors.

The implementation of reusability in Scala comes at cost; The standard implementation of Scala compiles to Java bytecode, where type erasure significantly reduces Scala program type information to create compatible Java bytecode. Consequently, autoboxing, operations that are needed when using primitive values in a generic context, is unnecessarily introduced into the final program.

Java bytecode is executed in the Java run-time environment, on a Java Virtual Machine. As a result, compiled Scala programs are reliant on JVMs for efficient execution using just-in-time compilation to low-level machine code. However, JVMs are predominantly optimized to improve the performance of Java programs. Current state of the art techniques on eliminating boxing and achieving optimal data representations at run-time, known as specialization, rely on static program analysis. Such techniques must mitigate the problem of code duplication as static optimizations are unable to make use of runtime information to best select which data structures to specialization

We propose a new approach to the specialization of Scala programs. Our approach integrates type information from a high-level source-like input language with the mechanisms of just-in-time compilation. We propose an ad-hoc specialization mechanism where specialized objects are compatible with non-specialized code. We demonstrate that our approach is viable and produces improvements in throughput for simplified implementations of real-world Scala programs.

Acknowledgements

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

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Abbreviations

AST Abstract Syntax Tree 6

DSL Domain Specific Language 18

IR Intermediate Representation 6

JIT Just-in-time 1, 3, 16, 32

JVM Java Virtual Machine 3, 24

TASTy Typed Abstract Syntax Tree 2, 3, 6, 21

Chapter 1

Introduction

Just-in-time (JIT) compilation has seen great success in the implementation of runtimes for objected-oriented programming languages. It has been effective in generating efficient machine code in the presence of virtual dispatch arising from *subtype* polymorphism. While a call site may statically have many possible call targets, JIT compilation is a able to incorporate dynamic run-time information to speculative optimize the most frequently invoked call targets. These speculative optimizations often enable compiled code to be inlined, a critical transformation in the context of JIT compilation. Inlining compiled code generates opportunities for many further optimizations.

Many object-oriented languages have since incorporated the notion of generic programming , otherwise known as *parametric* polymorphism. Parametric polymorphism enables programs to more modular and reusable as a functions and data structures behave identically[**?**] regardless of the types of their inputs. Implementations of generic programming often comes at the expense of program complexity and performance. Static compilers for object-oriented languages with parametric polymorphism must compromise when selecting an appropriate data representation for polymorphic data types and functions. This trade-off comes down to more optimal data layouts at the expense of space or uniform data layouts which are not optimal for every type at the expense of performance.

The selection of an optimal data representation, or *specialization*, of a polymorphic data structure relies on information typically found in the type-rich source language of programming languages. Representations must be consistent throughout the whole program as code which manipulates such data structures assume their representations to be consistent. Consequent, the specialization problem is best suited to compilers which have access to whole program information during compilation This not the case for object-oriented languages such as Java and Scala, which statically generate a uniform data representation for their polymorphic definitions to guarantee consistency throughout the whole program. Additionally, static compilers do not have sufficient runtime information that is critical in making favourable optimization decisions when compared to JIT compilers. On the other hand, JIT compilers are ill-suited to whole program optimizations as they are best at the dynamic optimization of small regions of a program. The problem of specialization therefore falls in between static compilation and JIT compilation.

This thesis introduces TastyTruffle, an interpreter and JIT compiler which incorporates rich source-level type information with speculative optimizations to specialize data representations for the Scala programming language. TastyTruffle is implemented in Truffle, a framework which simplifies the implementation of a JIT compiler for a source language through the implementation of a interpreter for that language. Our source language is the Typed Abstract Syntax Tree (TASTy) serialization format emitted by the Scala 3 compiler. TASTy is an abstract syntax tree format emitted after parsing and type checking of Scala programs. By using TASTy is a suitable source language, we are to access source-level type information without having parse and type check a Scala source program.

The contributions of this thesis are as follows: The implementation of an interpreter for the TASTy format using Truffle and the necessary transformations to make a TASTy program executable. An extension to the interpreter to support ad-hoc data representations for generic data structures. The evaluation of the interpreter on simple and realistic programs that are present a challenge on existing state-of-the-art techniques.

We describe the layout of the remainder of this thesis. Chapter 2 provides an overview on the many intermediate representations of Scala from compilation to execution. It explores the advantages and drawbacks of each intermediate representation with respect to specialization. Chapter 3 details the implementation of TastyTruffle. It covers the translation of TASTy into a more suitable IR for execution in an interpreter where each polymorphic data structure has a uniform representation. The chapter then provides extensions to the interpreter to the support just-in-time specialization of polymorphic data structures. Chapter 4 evaluates the interpreter with and without extensions for dynamic specialization on simple but realistic data structures. The chapter provides the performance of these evaluated data structures in the context of the standard implementation of Scala with the underlying JIT compiler of our interpreter without any augmentation. Chapter 5 explores related work in various implementations of parametric polymorphism as well as other Truffle interpreters. Chapter 6 discusses possible extensions to TastyTruffle to better integrate source-level type semantics with JIT compilation. Chapter 7 concludes the thesis.

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 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[54] is an objected-oriented, generic and statically typed programming language. Scala uses a *pure* object-oriected programming model[36] and addresses many of the shortcomings[34] 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[57]. 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[24] 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*[66]. Polymorphic classes may define behavior independent of their data, allowing them to be reused extensively for multiple types of data. In this thesis, we will interchangeably use the term *parametric polymorphism* to refer to generics.

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.

```
abstract class List[+T] {
    def head: T
    def tail: List[T]
    def length: Int
    def isEmpty: Boolean = length == 0
    def contains[T1 >: T](elem: T1): Boolean
}
```

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.

```scala
case class Cons[+T](head: T, tail: List[T]) extends List[T] {
    override def length: Int = 1 + tail.length

    override def contains[T1 >: T](elem: T1): Boolean = {
        var these: List[T] = this
        while (!these.isEmpty)
            if (these.head == elem) return true
            else these = these.tail
        false
    }

    override def hashCode(): Int = {
        var these: List[T] = this
        var hashCode: Int = 0
        while (!these.isEmpty) {
            val headHash = these.head.## // Compute hashcode
            if (these.tail.isEmpty) hashCode = hashCode | headHash
            else hashCode = hashCode | headHash >> 8
            these = these.tail
        }
        hashCode
    }
}
```

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.

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

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[47], optimization[47][23], or execution[48][49].

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 B). 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 represenations are significant in section 2.5.

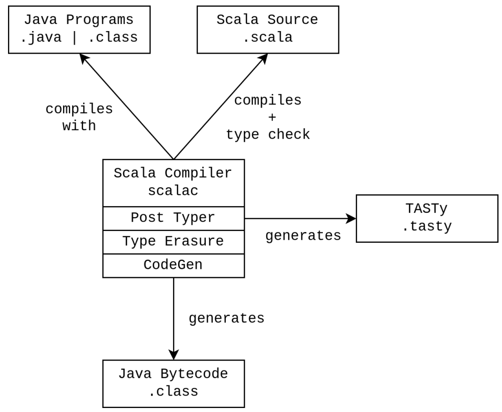


Figure 2.4: TASTy in the context of the Scala compilation pipeline.

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[34] 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-

```
// Tree representing code written in the source
trait Tree {
    def symbol: Symbol
}
trait Statement extends Tree       // Tree representing a statement in the source code
trait Definition extends Statement // Tree representing a definition in the source code.

// Tree representing a class definition.
case class ClassDef(
    name:        String,
    constructor: DefDef,
    parents:     List[Tree],
    self:        Option[ValDef],
    body:        List[Statement]
) extends Definition
// Tree representing a method definition in the source code
case class DefDef(
    name:      String,
    params:    List[ParamClause],
    returnTpt: TypeTree,
    rhs:       Option[Term]
) extends Definition
// Tree representing a value definition in the source code.
case class ValDef(name: String, tpt: TypeTree, rhs: Option[Term]) extends Definition
// Tree representing a type (parameter or member) definition in the source code
case class TypeDef(name: String, rhs: Tree) extends Definition
```

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

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.

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

Figure 2.6: Simplified implementation of the `object Nil`

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[15][22] 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.

```
ClassDef(
    // name
    "List",
    // constructor
    DefDef("<init>", List(TypeParams(TypeDef("T", TypeBoundsTree(_, _)), TermParams(Nil)), _, None)),
    // parents
    List(Apply(Select(New(_, "<init>"), Nil))),
    // self
    None,
    // body
    List(
        TypeDef("T", TypeBoundsTree(_, _)),
        DefDef("head", Nil, TypeIdent("T"), None),
        DefDef("tail", Nil,Applied(TypeIdent("List"), List(TypeIdent("T"))),None),
        DefDef("length", Nil, TypeIdent("Int"), None),
        DefDef("isEmpty", Nil, TypeIdent("Boolean"), None),
        DefDef(
            "contains",
            List(
                TypeParams(TypeDef("T1", TypeBoundsTree(TypeIdent("T"), _))),
                TermParams(ValDef("elem", TypeIdent("T1"), None))
            ),
            TypeIdent("Boolean"),
            None
        )
    )
)
```

Figure 2.7: Tree structure for the definition of `List` . For brevity, we use _ to represent inferred[26] 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 contains 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

```scala
// Tree representing an expression in the source code
trait Term extends Statement {
    def tpe: Type
}
// Tree representing a reference to definition
trait Ref extends Term

// Tree representing an assignment lhs = rhs in the source code
case class Assign(lhs: Term, rhs: Term) extends Term
// Tree representing new in the source code
case class New(tpt: TypeTree) extends Term
// Tree representing a block `{ ... }` in the source code
case class Block(statements: List[Statement], expr: Term) extends Term
// Tree representing a while loop
case class While(cond: Term, body: Term) extends Term
// Tree representing an if/then/else if (...) ... else ... in the source code
case class If(cond: Term, thenp: Term, elsep: Term) extends Term
// Tree representing a return in the source code
case class Return(expr: Term, from: Symbol) extends Term
// Tree representing a selection of definition with a given name on a given prefix
case class Select(qualifier: Term, selector: String) extends Term
// Tree representing an application of arguments.
case class Apply(applicator: Term, arguments: List[Term]) extends Term
// Tree representing an application of type arguments
case class TypeApply(fun: Term, args: List[TypeTree]) extends Term
// Tree representing a reference to definition with a given name
case class Ident(name: String) extends Ref
// Tree representing constant value
case class Constant(value: Int | ... | String) extends Term
```

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 intreprets 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.

```scala
// Type tree representing a type written in the source
trait TypeTree extends Tree {
    def tpe: Type
}

// Type tree representing a reference to definition with a given name
case class TypeIdent(name: String) extends TypeTree
// Type tree representing a type application
case class Applied(tpt: TypeTree, args: List[TypeTree | TypeBoundsTree]) extends TypeTree
// Type tree representing a type bound written in the source
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 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*[17], in polymorphic classes or polymorphic methods. However, type bounds are also inserted by the Typer because type parameters in TASTy are universally contraints. 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.

```
trait Type                          // A type, type constructors, type bounds
trait NamedType extends Type        // Type of a reference to a type or term symbol
case class TypeRef extends NamedType // Type of a reference to a type symbol
case class AppliedType extends Type  // A higher kinded type applied to some types T[U]
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 complilation 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 reprensentation amenable for Java bytecode generation and the necessary additional runtime overhead associated with this transformation.

2.5 Type Erasure

Type erasure[53] 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 Scala compilation that renders TASTy incomplete. Figure 2.12 shows the `Cons` class after type erasure.

```scala
case class Cons(head: Any, tail: List) extends List {
    override def length: Int = 1 + tail.length

    override def contains(elem: Any): Boolean = {
        var these: List = this
        while (!these.isEmpty)
        if (these.head == elem) return true
        else these = these.tail
        false
    }

    override def hashCode(): Int = {
        var these: List = this
        var hashCode: Int = 0
        while (!these.isEmpty) {
            val headHash = these.head.##
            if (these.tail.isEmpty) hashCode ||= headHash
            else hashCode |= headHash >> 8
            these = these.tail
        }
        hashCode
    }
}
```

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[54] 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 primitives types and reference types on the JVM[48]. 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.

```
// Before type erasure
val lst: List[Int] = Cons(1, Nil)
val head: Int = lst.head
// After type erasure
val lst: List = Cons(box(1), Nil)
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[59]. 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[74] is an implementation of a JVM. Traditionally, the JVM is responsible for the majority of the performance optimizations in Java programs[58] 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[33][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 infrastrucure.

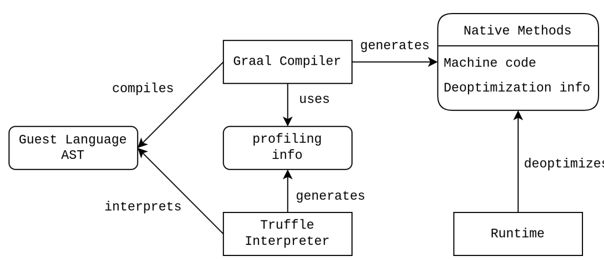


Figure 2.14: GraalVM overview[29].

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[58] 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[37], there is a direct translator for Java programs in GraalVM which parses Java bytecode into Graal IR.

Graal IR[29][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[21] and satisfies the *static single-assignment*[23] property. A sea of nodes is an abstraction based on a directed graph structure which relates the control flow graph[8] of a program to its data flow graph[7]. 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[43].

GraalIR enables Graal to speculatively compile only the *hot* branches[30], 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*[41] 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 signficantly 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

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

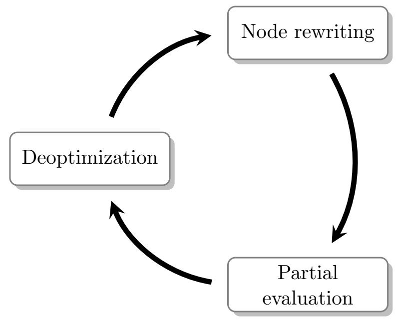


Figure 2.15: Adaptive optimization loop of GraalVM

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.

```
abstract class EqualsNode extends BinaryOpNode {
    @Specialization
    def equalsInt(lhs: Int, rhs: Int): Boolean = lhs == rhs

    @Specialization(replaces="equalsInt")
    def equals(lhs: Any, rhs: Any): Boolean = if (lhs == null) rhs == null else lhs.equals(rhs)
}
```

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

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 implemenation 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).

```
@GeneratedBy(EqualsNode.class)
public final class EqualsNodeGen extends EqualsNode {
    @Child
    private TermNode lhs_;
    @Child
    private TermNode rhs_;
    @CompilationFinal
    private int state_0_;

    private EqualsNodeGen(TermNode lhs, TermNode rhs) {
        this.lhs_ = lhs;
        this.rhs_ = rhs;
    }

    public Object execute(VirtualFrame frame) {
        int state = this.state_0_;
        return (state & 2) == 0 && state != 0 ?
            this.execute_int_int0(state, frame) :
            this.execute_generic1(state, frame);
    }
```

Figure 2.17: Generated code by the Truffle DSL for the `AnyEqNode`.

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.

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)[32]. 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)[72]. 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 which detail the implementation of a Truffle interpreter and extensions for parametric polymorphism. The first half of this chapter will describe the methods used to transform TASTy to make it suitable for a Truffle interpreter, TastyTruffle, *without* polymorphism. In particular, the first section covers how to translate the organization of data and code in the `DefDef`, `ClassDef`, and `Term` tree nodes into a Truffle implementation which is amenable to execution and JIT optimization. The second half of this chapter will then discuss extensions to our implementation support parametric polymorphism and cover the techniques we use to specialize nodes to eliminate autoboxing in the presence of polymorphism.

3.1 The Monomorphic Interpreter

Scala programs in TASTy format are unsuitable for execution in a Truffle interpreter. Programs in TASTy 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 B) that would otherwise normalize the IR are not present. Instead, we implement TastyTruffle IR to represent a canonicalized executable intermediate representation which can later be specialized on demand.

Figure 3.1 gives an evaluation loop that is common in other interpreters in the context of this one. A top level tree is any tree without a parent. In our subset of TASTy, a top

```scala
def evaluate(tree: Tree): Object = tree match {
    case vdef: ValDef   =>
        lazy val obj = initializeObject(vdef)
        registerObject(vdef.symbol, obj)
    case cdef: ClassDef =>
        registerShape(cdef.tpe, parseClassDef(cdef))
    case _ => ()
}
```

Figure 3.1: Pseudocode to evaluate every top level tree.

level tree may be a `ValDef` (a singleton object) or a `ClassDef`. Here we only present the pseudocode sufficient to traverse a program in TASTy. Each top level definition is parsed and saved. Note that we intentionally omit *how* to execute the program. Entry points in `TASTy` are defined by a special method `main`. As multiple entry points may exist in a given program, we consider the selection of entry points an implementation-specific detail. 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.

3.1.1 Converting the `DefDef` tree into a Truffle `RootNode`

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 abstraction 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[64] to profile its number of invocations. When a root node has been frequently invoked 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.

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

```
abstract class RootNode(desc: FrameDescriptor) {
    def execute(frame: VirtualFrame): Object
    def getCallTarget: CallTarget
}
```

Figure 3.2: Pseudocode of a root node.

arguments supplied during invocation and storage slots for local variable definitions in the body of the method. By default, all frames start off *virtual*. Virtual frames are Truffle abstractions that provide guest languages an opportunity to exploit escape analysis. Escape analysis[44] reasons about the dynamic scope of object allocations. Truffle and Graal both exploit the observations of *Partial Escape Analysis*[65], a path-sensitive variant of escape analysis, to enable the following optimizations for guest languages:

Region Allocation[12][69] The substitution of heap allocations with stack allocations to eliminate unnecessary garbage collection.

Scalar Replacement[45] The complete elimination of an object allocation, where the fields of the replaced object are substituted by local variables.

The virtual frame abstraction allows guest languages to read and write to a frame without the requirement to optimize their object allocations. Instead, escape analysis and subsequently scalar replacement is responsible for optimizing guest language object allocations during partial evaluation.

```
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 **??**

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

def getFrameSlotKind(tpe: Type): FrameSlotKind =
    if (tpe.isPrimitive)
        getPrimitiveSlotKind(tpe) // Int => FrameSlotKind.Int, ..., Double => FrameSlotKind.Double
    else
        FrameSlotKind.Object
```

Figure 3.4: Simplified implementation of `FrameSlotKind`

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 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 3.1.3), are copied into their respective parameter frame slots. Frames contains separate regions for values of each frame slot kind. Based on the frame slot

```
case class LocalFrameVal(slot: FrameSlot, kind: FrameSlotKind)

class DefDefNode(desc: FrameDescriptor, params: Array[LocalFrameVal], body: TermNode) extends RootNode(desc) {
    override def execute(frame: VirtualFrame): Object = {
        copyArgumentsToFrame(frame)
        try {
            body.execute()
        } catch {
            case ex: ReturnException => ex.getValue
        }
    }

    def copyArgumentsToFrame(frame: VirtualFrame): Unit =
        for ((param, arg) <- params zip frame.getArguments)
            param.kind match {
                case FrameSlotKind.Int =>
                    frame.setInt(param.slot, arg.asInstanceOf[Int])
                ...
                case FrameSlotKind.Double =>
                    frame.setDouble(param.slot, arg.asInstanceOf[Double])
                case _ =>
                    frame.setObject(param.slot, arg)
            }
}
```

Figure 3.5: Pseudocode for `DefDefNode` and `Parameter`

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`. In addition to the parameters which are explictly 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.

```scala
def parseDefDef(ddef: DefDef): DefDefNode = {
    val desc = new FrameDescriptor
    val parameters = self :: ddef.params.map {
        case vdef: ValDef => generateLocal(vdef, desc)
    }

    val body = parse(ddef.rhs)
    new DefDefNode(desc, parameters, body)
}

def generateLocal(vdef: ValDef, desc: FrameDescriptor): LocalFrameVal = {
    val kind = getFrameSlotKind(vdef.tpt.tpe)
    val slot = desc.addSlot(kind)
    Parameter(slot, kind)
}
```

Figure 3.6: Pseudocode for parsing `DefDef` into `DefDefNode`

3.1.2 Deriving a `Shape` from a `ClassDef`

```scala
class ClassDef(
    name:        String,
    constructor: DefDef,
    parents:     List[Tree],
    _:           Option[ValDef],
    body:        List[Statement]
) extends Definition
```

(a) Pseudocode of a `ClassDef`.

```scala
class ClassShape(
    symbol:  Symbol,
    parents: Array[Symbol],
    fields:  Array[Field]
    methods: Map[MethodSignature, CallTarget]
    vtable:  Map[MethodSignature, Symbol]
)
```

(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*[20][73]. 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[25]. However, programs which dynamically change the layout of their objects[2] are out of the scope of this thesis.

```
def parseClassDef(cdef: ClassDef): ClassShape = {
    val parents = cdef.parents.map(_.symbol)

    val fields = cdef.body map {
        case vdef: ValDef => generateField(vdef)
    }

    val methods = (cdef.constructor :: cdef.body) map {
        case ddef: DefDef => ddef.symbol.signature -> parseDefDef(ddef)
    }

    val vtable = cdef.symbol.methodMembers map {
        symbol => symbol.signature -> symbol
    }

    new ClassShape(cdef.symbol, parents, fields, init ++ methods, vtable)
}

def generateField(vdef: ValDef): Field = vdef match {
    case ValDef(_: String, tpt: TypeTree, rhs: Option[Term]) => new Field(vdef.symbol, vdef.tpt.tpe)
}
```

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

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. Figure 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`.

```scala
class Field(symbol: Symbol, tpe: Type) extends StaticProperty {
    override def getId: String = symbol.name

    def get(instance: Object): Any =
        if (tpe == Int) getInt(instance)
        else if ...
        else if (tpe == Double) getDouble(instance)
        else getObject(instance)


    def set(instance: Object, value: Any): Unit =
        if (tpe == Int) setInt(instance, value.asInstanceOf[Int])
        else if ...
        else if (tpe == Double) setDouble(instance, value.asInstanceOf[Double])
        else setObject(instance, value)
}
```

Figure 3.9: Pseudocode of the field property.

```scala
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*[66], where methods share the same name but have different arguments. When combined with parametric polymorphism, method signatures must also be able to disamibguate 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 signficant 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 Transforming `Terms` into `Nodes`

```
abstract class TermNode extends Node with InstrumentableNode {

    def execute(frame: VirtualFrame): Object
    def executeInt(frame: VirtualFrame): Int = execute(frame).asInstanceOf[Int]
    ...
    def executeDouble(frame: VirtualFrame): Double = execute(frame).asInstanceOf[Double]

}
```

Figure 3.11: Pseudocode of a TermNode.

In this section we will cover the conversion of a `Term` trees into Truffle nodes. The Truffle `Node` abstraction allows guest languages to implement executable fragments of an AST. Figure 3.11 is our subclass of a Truffle `Node`. Subclasses of the `TermNode` will define node-specific semantics encapsulating a particular functionality of the interpreter. The `TermNode` takes advantage of Truffle's autoboxing elimination by defining companion `execute[TYPE]` methods to allow subclasses to declare when an expected result from a child node must conform to a specific primitive type. In the following subsections, we give the subclasses which individually implement functionality of the monomorphic interpreter.

Creating Instances

The `New` tree represents the allocation of an instance of a `ClassDef`. The Truffle equivalent allocate node given in figure 3.12 is not so different, but it allocates an instance 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 initializer, `<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.

```
def parseNew(new: New): NewNode = new NewNode(new.tpe.symbol)

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

Figure 3.12: Pseudocode of a `NewNode` and how it is parsed.

Function Application

```
def parseApply(apply: Apply): ApplyNode = {
    val signature = apply.symbol.signature
    apply match {
        case Apply(Select(qualfier, _), arguments) =>
            if (qualifier.tpe.isPrimitve)
                if (args.length == 0) unaryOp(signature, qualifier)
                else                  binaryOp(signature, qualifier, args(0))
            else if (qualifier.tpe.isArray)
                arrayOp(signature, qualifier, arguments)
            else
                new ApplyNode(signature, parse(qualifier), arguments.map(parse))
        }
    }
```

Figure 3.13: 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 3.13 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 an Truffle intrinsic node intrinsic which represent a similar JVM equivalent. 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[48] 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 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[38] 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

```
@NodeChild("receiver")
@NodeField("signature", MethodSignature.class)
class ApplyNode(@Children args: Array[TermNode]) extends TermNode {
    final val INLINE_CACHE_SIZE: Int = 5;

    @Specialization(guards = "instance.getShape == shape", limit = "INLINE_CACHE_SIZE")
    def cached(
        frame: VirtualFrame,
        instance: ClassInstance,
        @Cached("instance.getShape") shape: ClassShape,
        @Cached("create(resolveCall(instance, signature)") callNode: DirectCallNode
    ): Object = callNode.call(evalArgs(frame, instance));

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

Figure 3.14: Simplified implementation of the call node with a polymorphic inline cache used in TastyTruffle.

a polymorphic inline cache[40] to eliminate the overhead of resolving virtual calls for JIT compilation.

Figure 3.14 shows a simplified Truffle call node in TastyTruffle which implements a polymorphic inline cache. The `ApplyNode` is declared using the Truffle DSL. The `@NodeChild` and `@NodeField` annotations declare that the DSL should generate children and properties of those names and types respectively. The `@Specialization` annotation declares the node writing semantics for method invocation. Because we have defined a limit on the number of specializations, the DSL will also generate additional code for a polymorphic inline cache. This cache saves call targets based on the type of receiver seen at the call site.

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 and therefore the call target does change between invocations. Furthermore, this allows the calls site to be inlined, allowing a feedback loop of intraprocedural optimizations[71][10] 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 call as the cost of inline cache lookup will outweigh the penalty of an indirect call.

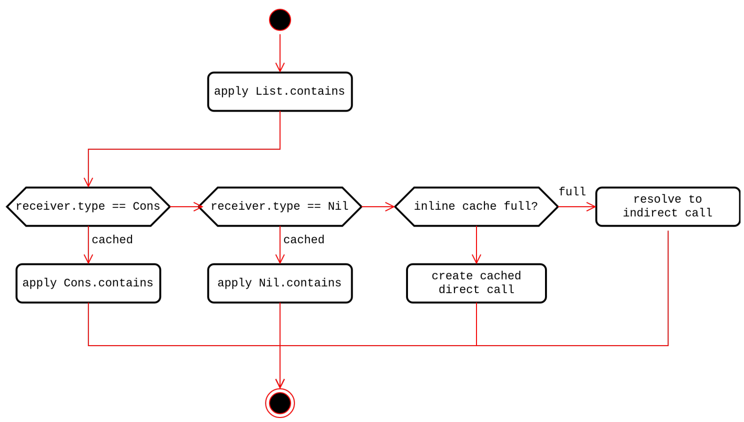


Figure 3.15: A possible polymorphic inline cache for a `List.contains` callsite.

Figure 3.15 shows a data flow diagram of the application of a polymorphic inline cache to a call site of `contains` when the receiver type is 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[27]. 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.

Accessing Fields

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 value definition.

```
@NodeChild("receiver")
@NodeField("symbol", Symbol.class)
abstract class ReadFieldNode extends TermNode {
    final val INLINE_CACHE_SIZE: Int = 3;

    @Specialization(guards = "instance.getShape == shape", limit = "INLINE_CACHE_SIZE")
    def cached(
        instance: ClassInstance,
        @Cached("instance.getShape") shape: ClassShape,
        @Cached("lookupField(shape)") field: Field
    ): Object = field.getContents(instance)

    @Specialization(replaces = "cached")
    def virtual(instance: ClassInstance): Object = {
        val field = lookupField(instance.getShape)
        field.getContents(instance)
    }

    private def lookupField(shape: ClassShape): Field = shape.getField(symbol)
}
```

Figure 3.16: Pseudocode of field read node with a polymorphic inline cache.

Figure 3.16 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.

Accessing Locals and Globals

The `Ident` tree is a name that refers to either a local value or a global value. Local values take the form of a local variable or a method parameter. Global values refer to a top level `object` definition. We differentiate between a local and a global based on whether the symbol of the `Ident` tree refers to an `object` definition (shown in figure 3.17).

Figure 3.18 provides the implementations the `ReadGlobalNode` and `ReadLocalNode`. In our interpreter, local variables and method parameters are uniformly represented by

```
def parse(ident: Ident): TermNode = {
    if (ident.symbol.isObjectDef)
        new ReadGlobalNode(symbol)
    else
        new ReadLocalNode(localOf(symbol))
}
```

Figure 3.17: Pseudocode to parse an `Ident` tree.

```
object Globals {
    val values: Map[Symbol, ClassInstance] = ???
}

class ReadGlobalNode(symbol: Symbol) extends TermNode {
    override def execute(frame: VirtualFrame): Object = Globals.values.get(symbol)
}

class ReadLocalNode(local: Local) extends TermNode {
    override def execute(frame: VirtualFrame): Object = frame.getObject(local.index)
}
```

Figure 3.18: Pseudocode of local and global value read nodes.

the frame slot abstraction. During parsing, it is sufficient to maintain a mapping from symbols to a `Local` to adequately resolve which local variable is read. Truffle does not provide an abstraction for storing global values. Instead, we retain a mapping of symbols to instances for all global object value definitions. Recall from figure 3.1 that a top level value definition is registered. When the symbol of an `Ident` refers a `object` definition, the value is resolved by using the symbol.

Mutating Values

The `Assign` tree has context-dependent semantics based on the structure of its left-hand side term. Figure 3.19 contains the simplified logic to resolve `Assign` trees into the 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. The `WriteFieldNode` and `WriteLocalNode` mirror their read node counterparts but instead of reading from their respective locations,

```
def parseAssign(assign: Assign): TermNode = assign match {
    case Assign(select: Select, rhs) =>
        new WriteFieldNode(parse(select.qualifier), select.symbol, parse(rhs))
    case Assign(ident: Ident, rhs) =>
        new WriteLocalNode(localOf(ident.symbol), parse(rhs))
}
```

Figure 3.19: Pseudocode to parse an `Assign` tree.

they update the value at their locations instead.

Conditionals

```
def parseIf(i: If): IfNode = new IfNode(parse(i.cond), parse(i.thenp), parse(i.elsep))

class IfNode(@Child cond: TermNode, @Child t: TermNode, @Child f: TermNode) extends TermNode {
    val cp = ConditionProfile.create();

    override def execute(frame: VirtualFrame): Object = {
        if (cp.profile(cond.executeBoolean(frame)))
            t.execute(frame)
        else
            f.execute(frame)
    }
}
```

Figure 3.20: Pseudocode for parsing an `If` into an `IfNode`

The implementation of conditional control flow in our interpreter is quite simple. Two execution paths exists for the two possible results from evaluating the condition term; The path taken depends on the boolean after evaluation. An `IfNode` is derived from an `If` tree (given in figure 3.20), which allows for divergence in program control flow. The implementation of the TastyTruffle IR mirrors the semantics given by its original TASTy tree. In order to take advantage of conditional speculative optimization, we add a `ConditionProfile` onto the result of the condition term. A condition profile records the likelihood that a branch is either true or false. Graal then speculatively optimizes the frequently true or false branches of an `IfNode` using its condition profile.

Loops

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 `LoopNode` abstraction for implementations of guest language loop structures. The loop node abstraction allows guest languages to take advantage of *On-Stack Replacement*[31]. 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.

```
def parseWhile(tree: While): WhileNode = new WhileNode(parse(tree.cond), parse(tree.body))

class WhileNode(@Child cond: TermNode, @Child body: TermNode) extends TermNode {

    @Child val loopNode: LoopNode =
        Truffle.getRuntime.createLoopNode(new WhileRepeatingNode(cond, body))

    override def execute(frame: VirtualFrame): Object = {
        loopNode.execute(frame)
        ()
    }

    class WhileRepeatingNode(
        @Child cond: TermNode,
        @Child body: TermNode
    ) extends Node with RepeatingNode {
        val cp = ConditionProfile.create()

        override def executeRepeating(frame: VirtualFrame): Boolean =
            if (cp.profile(cond.executeBoolean(frame))) {
                body.execute(frame)
                true
            } else false

    }
}
```

Figure 3.21: Pseudocode for a `WhileNode`

So far in this thesis, the root node has been the primary compilation unit in Graal. Root nodes profile their invocation count and get JIT compiled when they've been invoked frequently. However, loop constructs which are executed for many iterations also justify JIT compilation. The loop node is an additional type of JIT compilation unit which Graal can compile. A key difference between loop nodes and root nodes 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 loop nodes are used in the next iteration after they are JIT compiled. As on-stack replacement is not a central focus of this thesis, we will only discuss it briefly because loop nodes are the recommended abstraction for guest languages to implement loop structures in Truffle.

Figure 3.21 contains the implementation of a `WhileNode` and its derivation from a `While` tree. Like our implementation of the `IfNode`, we add a condition profile onto the node which evaluates the termination condition inside `WhileRepeatingNode`. Truffle will automatically instrument the `WhileNode`. After sufficient iterations of the `WhileRepeatingNode`, the repeating node is compiled and the next iteration of the `WhileNode` will use the compiled repeating node.

Blocks

```
def parseBlock(block: Block): BlockNode = {
    val desc = getParentFrameDescriptor(block)

    val terms = block.statements map {
        case vdef: ValDef => generateBlockLocal(desc, vdef)
        case term => term
    }

    new BlockNode(terms, parse(block.expr))
}

def generateBlockLocal(desc: FrameDescriptor, vdef: ValDef): TermNode = {
    val local = generateLocal(vdef)
    new WriteLocalNode(local, parse(vdef.rhs))
}
```

Figure 3.22: Pseudocode for parsing `Block` into `BlockNode`

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 subset of TASTy, this means that a block may contain declarations of local variables as well as executable terms. Figure 3.22 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.

```
class BlockNode(stats: Array[TermNode], last: TermNode) extends TermNode {
    @ExplodeLoop
    override def execute(frame: VirtualFrame): Object = {
        for (stat <- stats)
            stat.execute(frame)

        last.execute(frame)
    }
}
```

Figure 3.23: Pseudocode of the `BlockNode`

Because terms always return a value, the `Block` tree must follow the same semantics. Figure 3.23 gives the pseudocode for our implementation of a `BlockNode`. The `@ExplodeLoop` is a Truffle DSL directive which guides Graal to unroll[9] the loop for execution of each child node. Unrolled loops simplify partial evaluation as each iteration of the loop is treated as an individual statement and thus they reveal constant values which are easier to partial evaluate. As the number of children in a `BlockNode` is known before execution, it makes sense to unroll this loop in order to simplify optimization.

Returns

A `Return` trees ends the execution of the current method and passes a value back to the caller. The semantics of returning control flow in Truffle is implemented as a program *exception*. An exception is a unexpected disruption of program control flow. Recall in figure 3.5 that a body of a `DefDefNode` is executed and a `ReturnException` is possibly caught. The implementation of the `ReturnException` and `ReturnNode` is given in figure 3.24. The `ReturnException` is a subclass of the `ControlFlowException`. Control flow exceptions are a special exception which Truffle treats different from other JVM exceptions for the purposes of control flow analysis. A return exception is thrown with the return value evaluated from a return node. The exception is then caught by the executing `DefDefNode`, where the return value is passed back to the caller as the result.

```
class ReturnException(result: Object) extends ControlFlowException

class ReturnNode(@Child term: TermNode) extends TermNode {
    override def execute(frame: VirtualFrame): Object = {
        val result = term.execute(frame)
        throw new ReturnException(result)
    }
}
```

Figure 3.24: Pseudocode of `ReturnException` and `ReturnNode`

Putting it All Together

```
Block(
    List(
        ValDef("these", _, This),
        While(
            Apply(Select(Ident("these"), "empty"), "!", List.empty),
            If(
                Apply(Select(Select(Ident("these"), "head")), "==", List(Ident("elem")))
                Return(Constant(true)),
                Assign(Ident("these"), Select(Ident("these"), "tail"))
            )
        )
    ),
    Constant(false)
)
```

Figure 3.25: TASTy of `Cons.contains`

In this section, we summarize all the tree transformations introduced for the monomorphic variant of our interpreter. Figure 3.25 is the structure of the `Cons.contains` method in TASTy. We have omitted the type tree which has been declared inside the local variable definition. We use the `Cons.contains` method as an example to summarize the transformations described in this section.

Figure 3.26 is the Truffle equivalent AST of `Cons.contains`. We use simple strings to represent symbols and method signatures in order to avoid unnecessary detail in the example. Notice that many TASTy nodes have an equivalent TastyTruffle IR which closely mirrors their structure. However, other TASTy nodes must be simplified to a representation more suitable for runtime. In particular, `ValDef` trees are eliminated and replaced by an

```
BlockNode(
    Array(
        WriteLocalNode("these", ReadLocalNode("this")),
        WhileNode(
            UnaryOpNode("!", ApplyNode("these", "List.isEmpty[0]()", Array.empty)),
            IfNode(
                ApplyNode(
                    FieldReadNode(ReadLocalNode("these"), "head"),
                    "Any.==[0]()",
                    ReadLocalNode("elem")
                ),
                ReturnNode(ConstantNode(true)),
                WriteLocalNode("these", ReadFieldNode(ReadLocalNode("these"), "tail")),
            )
        )
    ),
    ConstantNode(false)
)
```

Figure 3.26: `Cons.contains` as a Truffle AST

initializer node which assumes that the frame slot for the local variable definition was added during parsing. In the second half of this chapter we will describe the challenges of using these trees in the presence of parametric polymorphism and their associated performance overhead.

3.2 The Polymorphic Interpreter

```
abstract class TypeNode extends Term {
    override final def execute(frame: VirtualFrame): Object = resolveType(frame)
    def resolveType(frame: VirtualFrame): Type
}
```

Figure 3.27: An abstract type node.

In this section, we extend our interpreter to support the execution of polymorphic trees. To that end, we introduce the notion of *reified* type nodes. In essence, to implement specialization of polymorphic classes and methods, we make types a *first-class* value. Like the `TermNode` represents the `Term` tree node from TASTy, the `TypeNode` represents the

`Type` from TASTy but instead of producing a value from evaluation, it produces a *type*. To better illustrate this concept, figure 3.27 contains the implementation of the node superclass which evaluates to a type and not a value.

```
parseType(tpe: Type): TypeNode = tpe match {
    case ref: TypeRef => TypeRefNode(ref)
}

class TypeRefNode(ref: TypeRef) extends TypeNode {
    override def resolveType(frame: Frame): Type = ref
}
```

Figure 3.28: A `TypeNode` for handling type references.

Figure 3.28 gives the simplified implementation to reify type references in the polymorphic interpreter. For now, we will limit the scope of reified type to the simplest and introduce concepts which integrate reified types with Truffle abstractions further in the chapter. Figure 3.29 extends the `NewNode` to support the creation of object instances using reified type nodes. Because a type reference essentially reifies statically available type information, very little changes in the implementation of a `NewNode`.

```
def parseNew(new: New): NewNode = new NewNode(parseType(new.tpe))

class NewNode(@Child typeNode: TypeNode) extends TermNode {
    override def execute(frame: VirtualFrame): Object = {
        val tpe = typeNode.resolveType(frame)
        shapeOf(tpe).newInstance
    }
}
```

Figure 3.29: Extension to the `NewNode` for the polymorphic interpreter.

Because the underlying type of a type parameter is only known during runtime, introducing types during execution will allow data layouts to be determined at runtime. The type node is the abstraction we use to encapsulate this concept. The principal idea behind the type node is to allow for the resolution of types during runtime. Introducing a mechanism to resolve types during runtime avoids the pitfalls of type erasure. In this half of the chapter, whenever we discuss the advantages of the polymorphic interpreter, we will use a monomorphic interpreter where the code has undergone type erasure as our frame of reference.

Using the newly available type information during runtime, data layout can be specialized based on the types seen. In the following subsections, we will focus on specific instances of boxing using Graal IR of compiled code executed using the monomorphic interpreter. Then we introduce subclasses of the `TypeNode` and show how reified types can be utilized to specialize the data layouts from the monomorphic interpreter.

3.2.1 Specializing Methods

```
class DefDefTemplate(
    desc:    FrameDescriptor
    tparams: Int,
    vparams: List[ValDef | LocalFrameVal],
    locals:  List[ValDef | LocalFrameVal],
    rhs:     Term
) extends RootNode(desc) {
    def execute(frame: VirtualFrame): Object = ???
    def specialize(types: Array[Type]): DefDefNode = ???
}
```

Figure 3.30: Pseudocode for a `DefDefTemplate`.

Polymorphic methods in Scala can be polymorphic under class type parameters, method type parameters, or both. In this section, we will focus only on the specialization of methods which are polymorphic under their own type parameters. We defer the discussion of the specialization of class-polymorphic methods until the next section. We will introduce the concept of a *template*; Templates retain sufficient information about the data layout of a definition in TASTy to generate their runtime representations dynamically. Instead of a `DefDefNode`, a `DefDefTemplate` (given in figure 3.30) is a root node that represents a polymorphic method. When a `DefDefTemplate` is specialized, the result is a monomorphic `DefDefNode` specialization.

The specialization of a `DefDefTemplate` begins at invocation. Because type arguments are introduced at specific polymorphic call sites, method specializations must be created at or after invocation. When a method template is invoked with both type and value arguments, it forwards the value arguments to appropriate specialization based on the type arguments.

The specialization of a method template is the ad-hoc creation of a root node with a specialized frame descriptor. A `DefDefTemplate` retains number of type parameters

```
def parseDefDef(ddef: DefDef): DefDefNode | DefDefTemplate = {
    val tparams = ddef.params.filter(_.isInstanceOf[TypeDef]).length
    if (tparams == 0)
        createDefDefNode(ddef)
    else {
        val vparams = ddef.filter(_.isInstanceOf[ValDef]) map {
            case vdef @ ValDef(_, tpt, rhs) =>
                if (tpt.tpe.isTypeParameter)
                    vdef
                else
                    generateLocal(vdef)
        }

        val locals = liftLocals(ddef.rhs)
        new DefDefTemplate(desc, tparams, vparams, locals, ddef.rhs)
    }
}

def createDefDefNode(ddef: DefDef): DefDefNode // a monomorphic DefDef

...
```

Figure 3.31: Pseudocode for parsing `DefDef` into `DefDefNode`

it owns; this is sufficient to resolve type arguments for creation and dispatching to specializations and type parameters never collide by name. Source information about value parameters is stored on a template instead of abstracted local frame values. The type of value parameter can potentially be resolved from a method type parameter. Since the frame descriptor is unpopulated because value parameters are possibly polymorphic, it is not yet appropriate to create executable term nodes which may read from or write to the frame slots of polymorphic value parameters. Figure 3.31 extends the transformation of a `DefDef` to include method templates.

Invoking Polymorphic Methods

In this section we demonstrate when and where polymorphic methods are invoked. For this demonstration, we will show one of the natural benefits of executing TASTy. A polymorphic method invocation in TASTy is always an `Apply` tree node where the qualifier is a `TypeApply`. The `TypeApply` tree node represents a *type application*. Without delving into great detail, a type application is the process of producing a monomorphic method from a polymorphic method by matching type parameters to type arguments. Analogous to normal applications which accept values as arguments and produces values as results,

```scala
def parseApply(apply: Apply): ApplyNode = {
    val signature = apply.symbol.signature
    apply match {
        case Apply(Select(qualfier, _), arguments) => ... // monomorphic trees
        case Apply(TypeApply(Select(qualifier, _), targs), args) =>
            new ApplyNode(signature, parse(qualifier), (targs ++ args).map(parse))
    }
}
```

Figure 3.32: Extension to parsing a polymorphic `Apply` tree.

type applications accept types as arguments and produce types as a result. With this in mind, `TypeApply` nodes are a naturally suitable site to invoke and create specializations for methods.

Figure 3.32 extends the transformation of `Apply` tree nodes to include polymorphic applications. The application of a polymorphic method follows the same semantics as the application of a monomorphic method. The actual specialization of the frame layout occurs inside the template that a polymorphic `ApplyNode` invokes. This allows the invocation of polymorphic methods even in the presence of dynamic dispatch. In the next section, we will describe the additional machinery that is added *after* a polymorphic inline cache has resolved virtual dispatch to handle type application and how to make such mechanisms amenable for partial evaluation.

Typed Dispatch Chains

Dispatch chains[62] are multi-layered inline caches. We introduce the notion of *typed dispatch chains*. Typed dispatch chains integrate the semantics of type applications via a second inline cache after virtual call resolution. Figure 3.33 contains the simplified implementation of the execution semantics in a `DefDefTemplate`.

Specializations of polymorphic methods are created on demand then cached based on their reified type signatures. One particular challenge of making caching mechanism fold away in partial evaluation is that the cache must be a *compilation constant*. Truffle provides the `CompilationFinal` directive which indicates that a value which may not be a constant in the guest language implementation *will* be a constant when being partially evaluated. Type arguments at type application sites are always stable, i.e. their respective type nodes evaluate to the same type, the look up of the specialized call node should have no overhead when JIT compiled with aid of partial evaluation. To make this possible, we

```
class DefDefTemplate(...) extends RootNode(...) {

    @CompilerDirectives.CompilationFinal
    val specializations: Array[(Array[Type], DirectCallNode)] = Array.empty

    def execute(frame: VirtualFrame): Object = {
        val typeArguments = resolveArguments
        dispatchCached(frame, types)
    }

    @ExplodeLoop
    def dispatchCached(frame: VirtualFrame, typeArguments: Array[Type]): Object = {
        for ((typeSignature, specialization) <- specializations)
            if (typeSignature == typeArguments)
                return specialization.call(frame.getArguments)
        CompilerDirectives.transferToInterpreterAndInvalidate()
        dispatchNew(frame, typeArguments)
    }

    def dispatchNew(frame: VirtualFrame, typeArguments: Array[Type]): Object = {
        val specialization = specialize(typeArguments)
        val callNode = DirectCallNode.create(specialization)
        specializations += (typeArguments -> callNode)
        callNode.call(frame.getArguments)
    }

    ...
}
```

Figure 3.33: Pseudocode for typed dispatch inside a `DefDefTemplate`.

exploit a simple array of type signature and specialized call node pairs. When the loop for looking up a cache entry in the array is unrolled during partial evaluation (directed by `ExplodeLoop`), the loop is transformed into a block of conditional expressions for each cache entry. This unrolled loop combined with the injected knowledge that type argument values are compilation constants results in the conditional elimination[14] of checks for non-matching cache entries. Once the appropriate specialization is found, the call is forwarded to the root node which contains the specialized term nodes.

When a combination of type arguments have not yet been encountered and their corresponding specialization is unavailable, the specialization must be generated and then invoked. To prevent this *slow* path of execution from being JIT compiled, we direct the compiler to *bail out* of JIT compilation with the `transferToInterpreterAndInvalidate` directive. The directive allows guest languages to insert their own deoptimization points into the control flow of a program. This ensures code of the slow branch when creating the specialization is never compiled. Note that in the first case where a type argument lookup succeeds (the fast path), the directive is unreachable because the control flow of the code returns and therefore will not be part of compiled code.

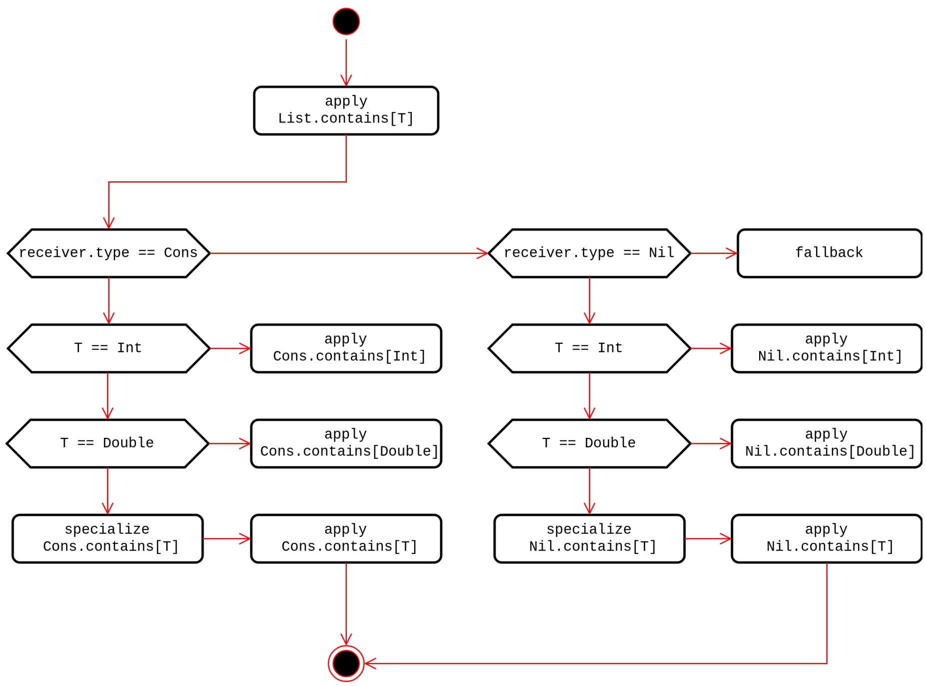


Figure 3.34: The typed dispatch chain for a `List.contains` call site

Figure 3.34 is an extension of the example given in figure 3.15 with typed dispatch. The example assumes the type arguments for `Int` and `Double` for `Cons.contains[T]` and `Nil.contains[T]` has previously been specialized and cached. After the polymorphic inline cache resolves the receiver to an exact type, the corresponding specialization is looked up. While this example may seem deceptively large, only the path taken in control flow is compiled after partial evaluation. For example, consider the invocation `List.contains[Int]` when the receiver is an instance of `Cons`, the corresponding compiled code will not contain the check that type parameter is an `Int`. Because all other program logic is eliminated during partial evaluation, the inlining of calls is also straightforward. After the partial evaluation, the typed dispatch mechanism is eliminated and the specialized method is the only code remaining.

Specializing Polymorphic Parameters

```
class DefDefTemplate(...) extends RootNode(...) {
    def execute(frame: VirtualFrame): Object = ...

    def specialize(types: Array[Type]): DefDefNode = {
        val desc = this.desc.copy()

        val parameters = self :: vparams.map(specializeValDef(types, desc))
        locals.foreach(specializeValDef(types, desc))

        val body = parse(rhs)
        new DefDefNode(desc, parameters, body)
    }

    def specializeValDef(types: Array[Type], desc: FrameDescriptor)v: LocalFrameVal | ValDef): LocalFrameVal = v match {
        case vdef: ValDef => generateLocal(types, vdef, desc)
        case v => v
    }
}
```

Figure 3.35: Pseudocode for on-demand specialization inside a `DefDefTemplate`.

The data layout of a method is given by the frame descriptor of its root node. A specialized method will have a specialized frame descriptor. Specialized frame descriptors will have the appropriate primitive frame slot kinds assigned to value definitions that have their polymorphic resolve to a primitive type. Therefore, the core principle behind method specialization is generation of a `TermNode` tree from a `Term` tree using a specialized frame descriptor. Apart from extensions given earlier in this section, the parsing of `Term` nodes do not differ from their monomorphic counterparts.

Figure 3.36 gives an extension to generate frame slots from type parameters and polymorphic value definitions atop the method to generate frame slots from monomorphic value definitions. Like its counterpart in the monomorphic interpreter, the abstraction for a local value in a frame in the polymorphic interpreter both have a slot. However, a local frame value in the polymorphic interpreter retains a type instead of a frame slot kind. If the type of a value parameter can be resolved with the type arguments supplied during specialization, the specialized frame slot is created and added to the descriptor. A mapping of types (via their symbols) to their respective index in the type argument array is sufficient to handle this resolution. Because we are only discussing methods polymorphic under their own type parameters, there are no polymorphic value parameters which are not resolvable in this context. The derivation of local frames values from monomorphic value definitions remain unchanged.

```
def generateLocal(types: Array[Type], defn: ValDef | TypeDef, desc: FrameDescriptor): LocalFrameVal =
    defn match {
        case tdef: TypeDef =>
            val kind = FrameSlotKind.Object
            val slot = desc.addSlot(kind)
            LocalFrameVal(slot, ReifiedType)
        case vdef: ValDef =>
            val idx = indexOf(vdef.tpt.tpe)
            val tpe  = if (idx != -1) types(idx) else vdef.tpt.tpe
            val kind = getFrameSlotKind(tpe)
            val slot = desc.addSlot(kind)
            LocalFrameVal(slot, tpe)
    }
```

Figure 3.36: Extension to pseudocode that generates frame slots to include polymorphic definitions.

$$index(\tau) = \begin{cases} i & \textbf{def } f[t_0, \ldots, t_i, \ldots, t_n](\ldots) \text{ if } t_i = \tau, owner(\tau) = f \\ -1 & \text{otherwise} \end{cases}$$

A type definition is treated in the same manner as a value definition, it is assigned a frame slot with a `Object` frame slot kind. This allows for the storage of types in the frame of a method, allowing for the resolution of types after the invocation of a method template. So far we have only discussed the resolution of type arguments in a intraprocedural context. Storing reified types in the frame during execution allows from the resolution in a interprocedural context. We will detail why this is important in the following subsection.

Propagating Type Arguments

Polymorphic code has a habit of using other polymorphic code. As a result, polymorphic invocations often occur inside the definition of a polymorphic class or a polymorphic method. That is to say that the type argument at a type application site could be a type parameter. Figure 3.37 is an example where a type application occurs inside the definition of a polymorphic method and derives its type argument from a type parameter.

We introduce a subclass of a type node that retrieves method type arguments which are stored on the frame. Because type parameters are treated in the same manner as value parameters, they are stored in the frame of the method. The resolution of type arguments which are parameters from a method follows the same mechanism as the resolution of local

```scala
def subset[T](a: List[T], b: List[T]): Boolean = {
    var curr: List[T] = a
    while (!curr.isEmpty) {
        if (!b.contains[T](curr.head)) return false
        curr = curr.tail
    }
    true
}
```

Figure 3.37: An example where type arguments are derived from type parameters.

```scala
class MethodParamTypeNode(@Child readLocal: ReadLocalNode) extends TypeNode {
    override def resolveType(frame: VirtualFrame): Type =
        readLocal.execute(frame).asInstanceOf[Type]
}
```

Figure 3.38: The type node for dynamically resolving method type parameters.

variables. This mechanism enjoys the same Truffle virtualization optimizations of value reads when propagating type arguments interprocedurally. In the next section, we will describe where and why this particular mechanism is able to specialize programs well and cases where Truffle is able to achieve similar optimizations.

Case Study: A List Constructor

Truffle conveniently profiles the types of frame arguments to speculatively eliminate the unboxing of boxed values when reading frame values (including arguments). Figure 3.39 is an example of such a speculative optimization. This speculative optimization relies on a `TrustedBoxedValue` to unbox the primitive. A `TrustedBoxedValue` represents injected information from an external source. In this particular case, it is known by the compiler that boxed instance comes from a cache; Unique `int` values may be mapped to a unique `Integer` instance in the Java runtime, which eliminates unnecessary boxed object creation. The unbox operation in node 848 will be 'floated' up the graph such that all subsequent dominated by the read of a boxed frame value has no autoboxing.

In contrast, write operations of polymorphic frame values cannot be speculatively eliminated. Because Truffle does not specialize data layouts, i.e. frames are determined by their descriptors, which in turn are determined by the guest language implementation, frame

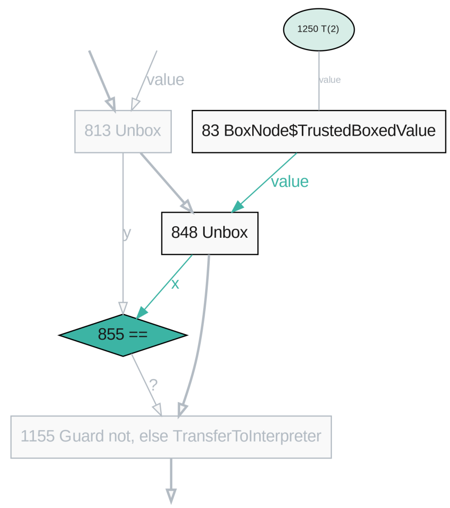


Figure 3.39: Graal IR with speculative unboxing of `elem` based on a type profile of its frame slot in `List.contains`

writes of polymorphic values will always have to be boxed. The elimination of unnecessary boxed polymorphic writes from frame descriptor specialization is one of the major benefits when compared to the monomorphic interpreter. Code that has polymorphic code which reads and writes to a frame frequently will no longer have to unbox, compute primitive operations on unboxed values, then box those values back into their respective slots. As our running example does not make use of this particular pattern of autoboxing, we will take this opportunity to showcase an instance of a polymorphic value read operation that Truffle is unable to speculatively optimize without specialization.

In both Scala and the JVM, arrays of primitive types are invariant with respect to the $\top$ (the universal supertype) type, `Any` and `Object` in Scala and Java respectively. That is to say, the type `Array[Int]` is neither a subtype or supertype of the type `Array[Any]`. On the other hand, the type `AnyRef` is covariant to the `Array[Any]` type. This contradiction in the presence of code which creates or operates on polymorphic arrays requires runtime bridge methods in order to appear seamless to a programmer. These bridge methods combined with the nature of Scala's type system obscures opportunities for speculative optimizations.

To showcase an instance of when polymorphic array code must be bridged, we define an alternate constructor for a `List[T]` to use an as example. Figure 3.40 gives a constructor that creates a polymorphic list from a polymorphic array. We focus on the term

```
1  object List {
2      def apply[T](array: Array[T]): List[T] = {
3          var i = array.length - 1
4          var these: List[T] = Nil
5          while (i >= 0) {
6              these = new Cons[T](array(i), these)
7              i -= 1
8          }
9          these
10     }
11 }
```

Figure 3.40: An alternate static constructor that converts an `Array[T]` to a `List[T]`

`array.length` which computes the length for a polymorphic array on line 3. When the Typer detects an array operation on a polymorphic array value, it automatically inserts the array runtime bridge method that is responsible for handling the operation. For example, line 3 after the Typer would be transformed into `var i = array_length(array) - 1`. We give the implementation of `array_apply` in figure 3.41.

```
def array_length(array: AnyRef): Int = {
    if (array.isInstanceOf[Array[AnyRef]])       array.asInstanceOf[Array[AnyRef]].length
    else if (array.isInstanceOf[Array[Int]])     array.asInstanceOf[Array[Int]].length
    else if (array.isInstanceOf[Array[Double]])  array.asInstanceOf[Array[Double]].length
    else if (array.isInstanceOf[Array[Long]])    array.asInstanceOf[Array[Long]].length
    else if (array.isInstanceOf[Array[Float]])   array.asInstanceOf[Array[Float]].length
    else if (array.isInstanceOf[Array[Char]])    array.asInstanceOf[Array[Char]].length
    else if (array.isInstanceOf[Array[Byte]])    array.asInstanceOf[Array[Byte]].length
    else if (array.isInstanceOf[Array[Short]])   array.asInstanceOf[Array[Short]].length
    else if (array.isInstanceOf[Array[Boolean]]) array.asInstanceOf[Array[Boolean]].length
    else throw new NullPointerException
}
```

Figure 3.41: Implementation of `array_length`

Notice the type of the argument in `array_length` is `AnyRef`; Because the types of primitive arrays and reference arrays are invariant, the direct supertype is `AnyRef`. To compute the length for a polymorphic, `array_length` switches over every similar but unrelated array type. In the body of every type check condition, the argument must be be cast to the appropriate array type after the type check succeeds before the length is finally computed. We will examine how this code looks in the context of our alternate constructor after JIT compilation in Graal IR.

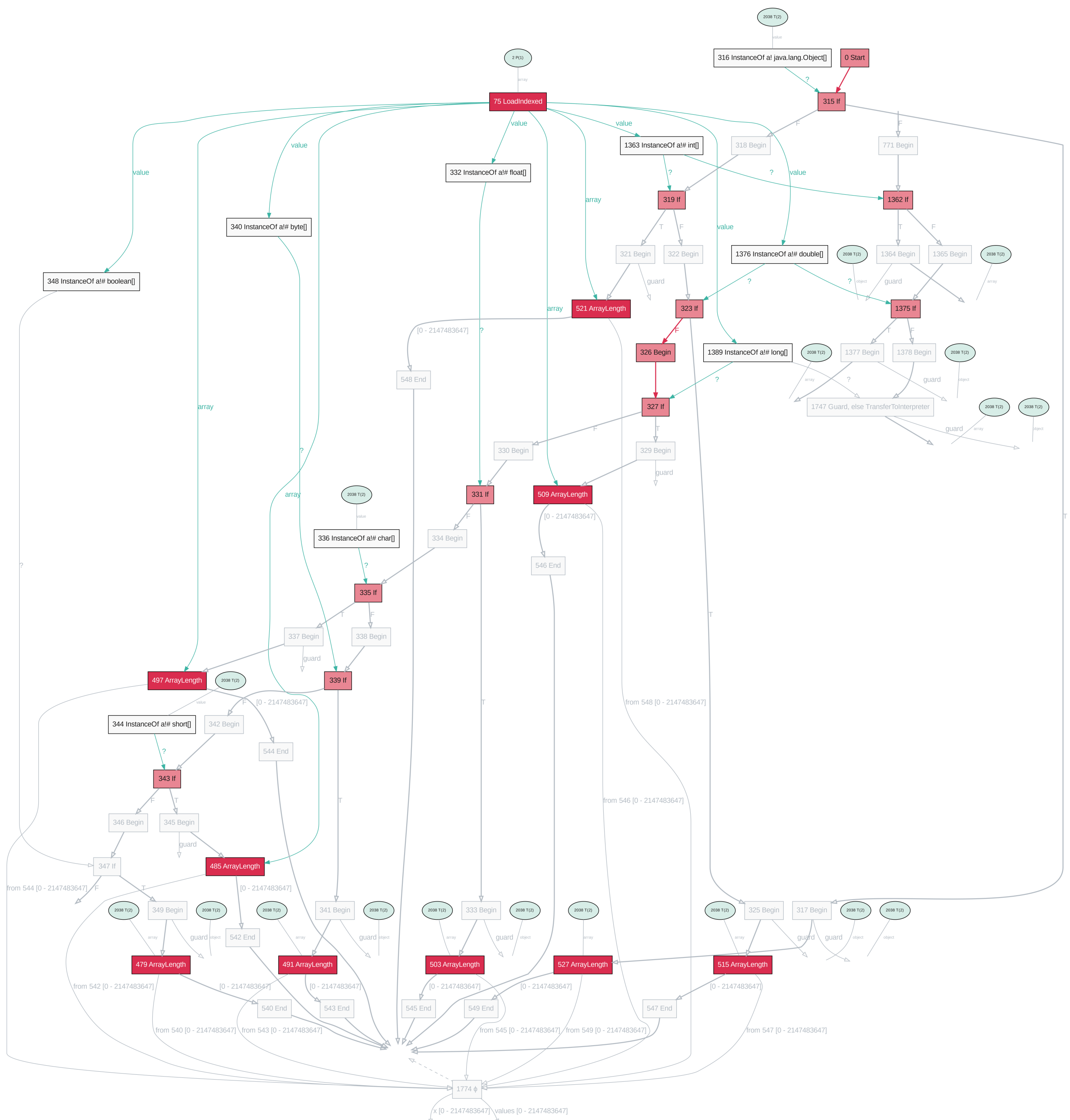


Figure 3.42: Graal IR of `array_length` in the context of `List.apply[T](array: Array[T])`

Figure 3.42 contains the Graal IR of `array_length` inlined into `List.apply[T]`. Notice that the `instanceof` type checks nodes (white) that are succeeded by an `ArrayLength` node (red) for each of the branches in `array_length`. The numerous consecutive conditional expressions complicate the control flow analysis in JIT compilation. These conditional checks add unnecessary branching and burdens JIT compilation when the type of a specialized array could be known from specialization. We introduce a method to vastly simplify the Graal IR of such instances of array bridge methods when specialized methods would have type-specific information to augment JIT compilation.

```
import CompilerDirectives.castExact
def copyArgumentsToFrame(frame: VirtualFrame): Unit =
    for ((param, arg) <- params zip frame.getArguments)
        param.tpe match {
            case Int =>
                frame.setInt(param.slot, arg.asInstanceOf[Int])
                ...
            case Double =>
                frame.setDouble(param.slot, arg.asInstanceOf[Double])
            case tpe: Array[AnyRef] | tpe: Array[Int] | ... | tpe: Array[Double] =>
                frame.setObject(param.slot, castExact(arg, getClass(tpe)))
            case _ =>
                frame.setObject(param.slot, arg)
        }
```

Figure 3.43: Pseudocode for `DefDefNode` and `Parameter`

In order to accomplish this, we extend the way that frame arguments are copied into the frame from figure 3.5. Because a parameter now retains its type instead of a frame slot kind, we introduce a special operation when copying arguments that are arrays. The `castExact` directive is a type narrowing operation that hints to Graal that a value is an instance of a type. We will examine how the insertion of this directive for array types simplify the layout of Graal IR graphs.

Figure 3.44 contains the simplified Graal IR of `array_length` inlined into `List.apply[T]`. Notice that there is a single `ArrayLength` node that is dominated by a π node. A π node[13] enforces a bound on a value. In the case of Graal, a π node enforces bound on the type of a value. More specifically in our example, the π node narrows the type of the 2nd parameter of `List.apply[T]` to a monomorphic array type. When the type of the parameter is narrowed, the type checks that enforce array types from figure 3.42 are eliminated because the type is now known.

This method does not consider polymorphic scenarios where an array of boxed primi-

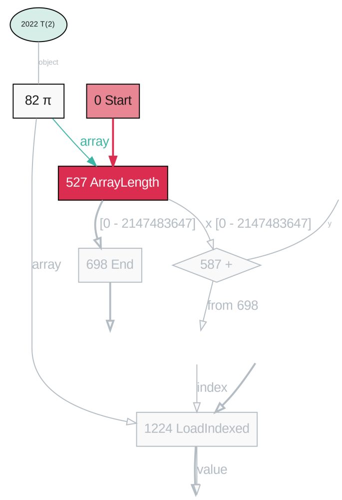


Figure 3.44: Graal IR of `array_length` in the context of `List.apply[T](array: Array[T])` augmented with a π node

tives are used interchangeably with an array of primitives. Such scenarios would require the insertion of additional autoboxing nodes or an intraprocedural transformation where boxed arrays are converted to primitive arrays. While these solutions are possible in the context of Truffle, we consider these kinds of scenarios out of the scope of this thesis.

3.2.2 Specializing Classes

In this section, we detail the specialization of classes and class members that have polymorphic semantics based on class type parameters. Previously we discussed the specialization of methods that are solely polymorphic under their own parameters without the mention of methods which are class-polymorphic. The reasoning behind this decision can be explained as thus: The invocation of a class-polymorphic method requires a look up into that class's shape, by extension that class must be specialized before such a polymorphic invocation may occur. As class specialization does not share the demands when it comes to runtime mechanisms as method specialization, we will adopt a rewrite driven approach to specializing class definitions on demand. That is, we will use techniques to monomorphize, at least partially, polymorphic TASTy trees instead of altering the transformation of a `ClassDef` to a `ClassShape`. With this rationale, we are able to adapt many elements of the monomorphic interpreter for polymorphism.

```scala
def parseClassDef(cdef: ClassDef, types: Array[Type]]): ClassShape = {
    val parents = cdef.parents.map(_.symbol)

    val fields = cdef.body map {
        case vdef: ValDef => generateField(vdef, types)
    }

    val methods = (cdef.constructor :: cdef.body) map {
        case ddef: DefDef => ddef.symbol.signature -> parseDefDef(ddef)
    }

    val vtable = cdef.symbol.methodMembers map {
        symbol => symbol.signature -> symbol
    }

    new ClassShape(cdef.symbol, parents, fields, init ++ methods, vtable)
}
```

Figure 3.45: Extensions to specialize a `ClassDef`.

```scala
class ClassParamTypeNode(@Child readField: ReadFieldNode) extends TypeNode {
    override def resolveType(frame: VirtualFrame): Type =
        readLocal.execute(frame).asInstanceOf[Type]
}
```

Figure 3.46: The type node for dynamically class method type parameters.

Figure 3.45 provides an overview on the steps that are required to create a specialized monomorphic `ClassDef` from a polymorphic origin. Similar to how type definitions become parameters when reified in the context of a `DefDef`, type definitions in the context of `ClassDef` become fields when reified. The rationale is that instances of specialized polymorphic classes store their specialized type fields in order propagate types. Figure 3.46 gives the pseudocode for a type node which resolves a type parameter from an instance of a specialized class. We rewrite both value definitions as well method definitions to transform polymorphic class definitions into monomorphic class definitions.

Creating Specialized Instances

The `AppliedType` is the analogue of `TypeApply` for type applications when creating object instances. We are able to derive a specialization site for class definition by the

```
parseType(tpe: Type): TypeNode = tpe match {
    ...
    case AppliedType(con, targs) => new AppliedTypeNode(parseType(con), targs map parseType)
}

class AppliedTypeNode(@Child con: TypeNode, @Children targs: Array[TypeNode]) extends TypeNode {
    @ExplodeLoop
    override def resolve(frame: VirtualFrame): Type {
        val types = Array.empty[Type]
        for (targ <- targs)
            types += targ.resolve(frame)

        AppliedType(con.resolve(frame), types)
    }
}

def shapeOf(tpe: Type): ClassShape = tpe match {
    ...
    case AppliedType(con, targs) =>
        val cdef = getClassDef(con)
        parseClassDef(cdef, targs)
}
```

Figure 3.47: The `AppliedTypeNode` and its derivation from an `AppliedType`.

reification of an applied type into an `AppliedTypeNode`. Figure 3.47 is an overview of the `AppliedTypeNode` and its derivation from its TASTY type counterpart. In our subset of TASTy, an applied type represents an instantiation of a polymorphic type.

```
new List[Int]
```

Figure 3.48: Example of creating instance of an applied type.

For example, consider the term, given in figure 3.48, that returns an instance of a polymorphic type The type `List[Int]` is the result of the type application of `List[T]` to `Int`. We will refer to polymorphic applied types, such as `List[T]`, as *polymorphic* applied types. The data representation of a polymorphic applied type is undetermined; Depending the types argument supplied during type application, the data representation will vary.

When executable nodes are derived from polymorphic applied types, given in figure 3.49, type arguments are resolved during runtime before application to their type constructor. We refer to the instantiations resulting from the application of type arguments

```
NewNode(AppliedTypeNode(TypeRefNode("List"), Array(TypeRefNode("Int"))))
```

Figure 3.49: TastyTruffle IR of creating an instance of an applied type.

to polymorphic applied types as *monomorphic* applied types (e.g. List[Int]). Having a monomorphic applied type provides the opportunity to generate a specialized shape. Therefore, each group of monomorphic applied types have a unique data representation. For example, a `List[Int]` and `List[Double]` will each have a unique data representation and the underlying layout of their instances will be different. However, a `List[String]` and `List[List[Int]]` will share the same data representation as their type arguments are reference types and will not see any benefit from independent specialization. Therefore, the creation of an object instance with a polymorphic class definition will have its shape determined when its created.

$$index(\tau) = \begin{cases} i & \textbf{def } f[t_0, \ldots, t_i, \ldots, t_n](\ldots) \text{ if } t_i = \tau, owner(\tau) = f \\ j & \textbf{class } C[t_0, \ldots, t_j, \ldots, t_m](\ldots) \text{ if } t_j = \tau, owner(\tau) = C \\ -1 & \text{otherwise} \end{cases}$$

This approach avoids the issue of *name mangling*. Name mangling is a technique to disambiguate distinct entities in a program that share the same name but do not inhabit the same namespace (e.g. a `package`). In the context of parametric polymorphism and specialization, many approaches to specialization require require the specialized classes and methods to have mangled names. Creation of polymorphic classes and and call sites of polymorphic methods must be rewritten to refer to the correct specialization. In our approach, operations on object instances with a polymorphic type are unaffected by its underlying shape. In the next section, we give extensions on how to generate a static shape from a monomorphic applied type after type application.

Specializing Class Members

In this section, we extend the translation scheme for generating shapes from class definitions to include polymorphic class definitions. There are two elements of data layout that must be determined when generating the shape of a polymorphic class definition. Fields constitute the portion of data layout on an object instance that must be resolved with monomorphic types for value definitions. Local frame values whose frame slot kinds

```
def generateField(vdef: ValDef, types: Array[Type]): Field = vdef match {
    case ValDef(_: String, tpt: TypeTree, rhs: Option[Term]) =>
        val idx = indexOf(tpt.tpe)
        val tpe = if (idx > 0) types(idx) else tpt.tpe
        new Field(vdef.symbol, tpe)
}
```

Figure 3.50: Extensions to generate a field from a polymorphic value definition.

must be derived from polymorphic type parameters constitute the other portion of data layout in a class definition that must be specialized.

The underlying type of a polymorphic field, and therefore its data representation as part of its static shape, cannot be determined statically. When the field translation scheme is supplied with type arguments, we are able to generate the specialized monomorphic field. Figure 3.50 extends the pseudocode that generates fields for shapes to include polymorphic value definitions. If it is beneficial to specialize a field, e.g. `val x: T` or `val x: Array[T]`, we resolve the type parameter from the type arguments to generate a specialized field property. Otherwise, we default to the monomorphic implementation for generating a field.

```
def parseDefDef(ddef: DefDef, types: Array[Type]): DefDefNode | DefDefTemplate = {
    val tparams = ddef.params.filter(_.isInstanceOf[TypeDef]).length

    val vparams = ddef.filter(_.isInstanceOf[ValDef]) map {
        case vdef @ ValDef(_, tpt, rhs) => specializeValDef(desc, vdef, types))
    }

    val locals = liftLocals(ddef.rhs) map {
        case vdef @ ValDef(_, tpt, rhs) => specializeValDef(desc, vdef, types))
    }

    if (vparams.forall(_.isInstanceOf[LocalFrameVal] && locals.forall(_.isInstanceOf[LocalFrameVal]))
        new DefDefNode(desc, vparams, ddef.rhs)
    else
        new DefDefTemplate(desc, tparams, vparams, locals, ddef.rhs)
}
```

Figure 3.51: Extension to parse a `DefDef` with class type arguments.

Polymorphic methods present a challenge to specialization because they can be polymorphic under two different sets of type parameters. As a result, dynamically resolved types are not available for specialization at the *same* time; Class type arguments are available at object creation and method type arguments are available at invocation. To address this, we need to be able to `partially specialize` methods from the class perspective. Figure 3.51 extends the translation of `DefDef` nodes with class type arguments. If the method is polymorphic under class type parameters, the layout of a frame for the root node of a `DefDef` must partially determined. We assume that `indexOf` is capable of resolving the index of a type parameter to a corresponding type argument array in a context-sensitive manner (i.e. whether type arguments originate from a type application of a class or a method).

After the class specialization of a `DefDef`, it is still possible that a `DefDef` contains polymorphic semantics. However, all polymorphism which is derived from a class type parameter has been specialized and such terms and parameters are now monomorphic. Therefore, a `DefDef` that is still polymorphic is only polymorphic under its own type parameters. The remaining polymorphic data layout will be specialized when the method template is invoked.

Example: `Cons[Int]`

TODO

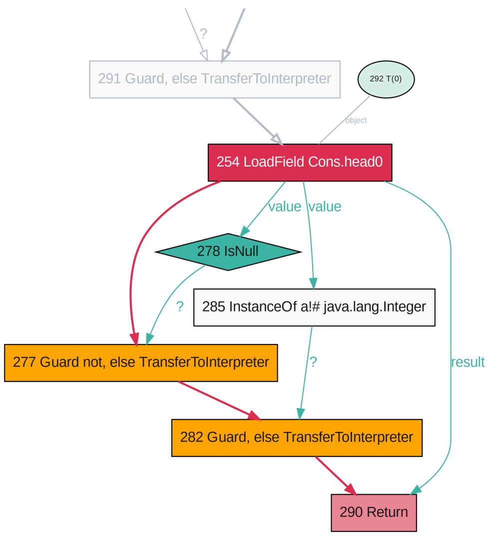
Case Study: `Cons.head`

```
val list: List[Int] = ???
list.contains(0)
```

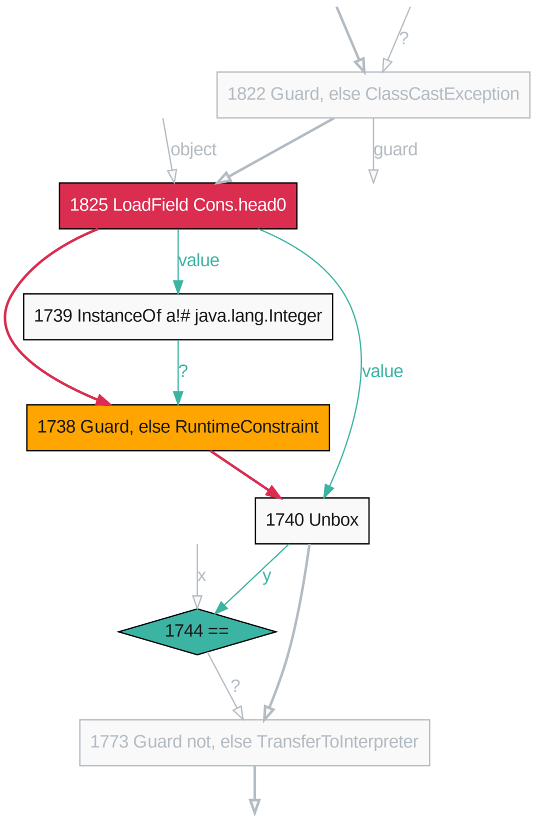
Figure 3.52: Example invocation of `Cons.contains[Int]`

We introduce an example, given in figure 3.52, which motivates the specialization of storage layouts in shapes. The example exercises polymorphic method invocations where the polymorphism is with class and method type parameters.

We examine in detail the Graal IR focusing on the `List.head` accessor method in our `List` running example. For this particular example, we focus on unboxing that occurs when the `head0` is accessed by the `List.head`. This unboxing can be seen in 3.53a.



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



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

We can see that *guard* nodes are inserted by Graal into the compiled graph during JIT compilation. A guard node ensures a speculative assumption still holds during execution. Because the default storage type of a polymorphic field without specialization is an `Object`, Graal makes two runtime assumptions about the field in the JIT compiled `contains` method to ensure the compiled method does not throw a runtime exception if the return value needs to be unboxed. The first guard, identifiable by node 278, checks that the value is not the `null` reference. As the `null` value is only compatible with reference types, attempting to unbox a `null` value produces a runtime exception. The second guard, with the identifier 282, is a type check that the value is an `Integer` object. Notice that the predecessor node is the type check `instanceof a!# java.lang.Integer` and not `instanceof java.lang.Integer`. `instanceof` nodes in Graal IR checks against *stamps* instead of normal JVM types identifiers. A stamp is much like a type identifier but has additional descriptors attached. For example, the stamp `a!# java.lang.Integer` has the following descriptors:

(a) Asserts that the stamp marks a reference type identifier. In the case of this stamp, the stamp marks the boxed reference type `java.lang.Integer`.

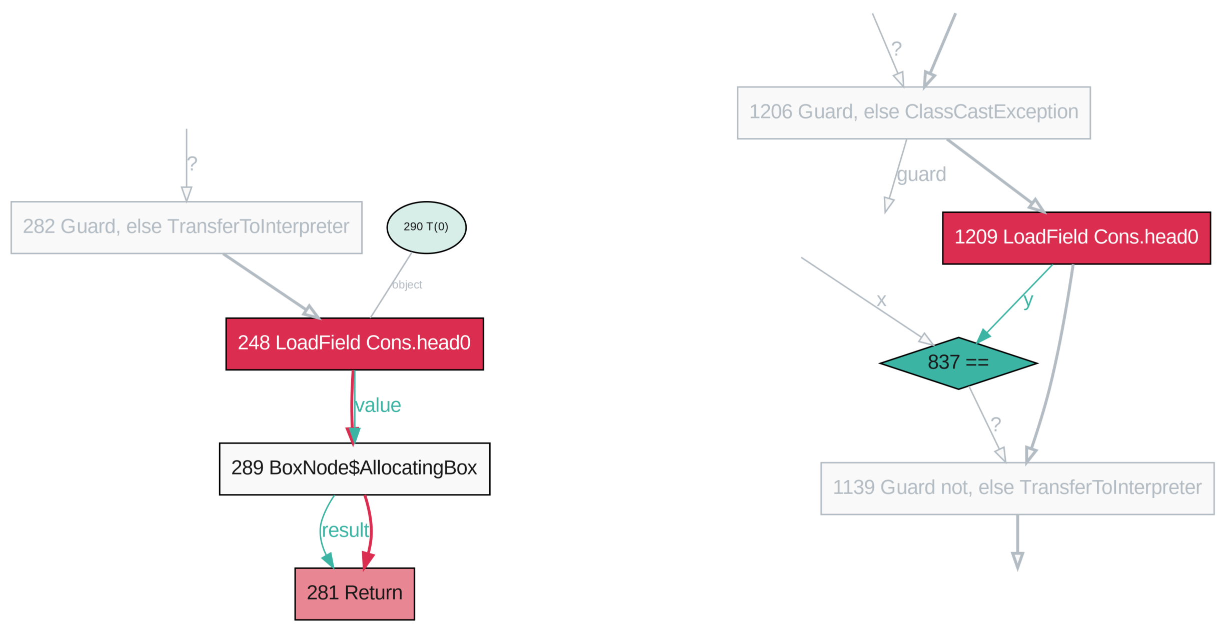
(!) Asserts that value is not the `null` reference value. The stamp contains this descriptor because it is preceded by a non-null guard.

(#) Asserts that value marked by the stamp is *exactly* an instance of the type identifier described by the stamp and not an instance of a subclass of the type identifier

In more succinct terms, the `instanceof` node 285 checks that value is precisely the instance of a `java.lang.Integer` and is not the `null` value. If the assumptions are not violated in compiled code, the boxed integer value is then returned from the compiled code. Note that no unboxing happens because the value of `head0` has not yet been used in a polymorphic context.

When the access or method `List.head` is inlined into its callsite in `Cons.contains` (see figure 3.53b), an unbox operation is introduced because the equality operation in node 1744 compares primitives and not references. Notice that the two guards node previously seen in figure 3.53a are folded into one node because the `instanceof` node is an extension of the null check node. Because polymorphic field values are stored as an reference on the object instance, these speculative assumptions are necessary in order to generate compiled code. To eliminate the overhead of the unbox operation and the accompanying guard nodes, The polymorphic fields of a class must be specialized.

Figure 3.54a contains the field access of `head0` after the field has been specialized and has the appropriate storage type in the storage layout of `Cons[Int]`. Notice that a box node has been introduced prior to the value of `head0` prior to the return node of `Cons.head`. Because the `execute` method of a `DefDefNode` returns an `Object`, the return value is preemptively boxed when inspecting the IR of the method. However, after inlining into the body of `Cons.contains`, the box operation is no longer necessary as the boxed value will be immediately unboxed. Graal will automatically eliminate this type of autoboxing. When a specialized class instance is used in place of a generic class instance, the field access sub graph of `head0` is completely simplified.



(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

In this chapter, we will evaluate and discuss the performance of our polymorphic interpreter on six microbenchmarks. We use an existing set of benchmarks from [70] as they exercise many features of the Scala runtime that require specialization to perform optimally. We will evaluate performance of these benchmarks on the monomorphic interpreter as well as Scala bytecode on GraalVM as points of comparison for relative performance. Finally, we will discuss the results of the benchmarks.

4.1 Benchmarks

In this section, we will introduce an additional program on top of our running example for benchmarking. We will also summarize the motivations from [70] for the selection of these benchmarks.

Each microbenchmark exercises unique polymorphic operations which are typically performance bottlenecks[59][63] in Scala programs. The `ArrayBuffer` class is an implementation of a resizable buffer backed by an array. It contains three microbenchmarks which stress polymorphic operations in the context of contiguous memory access.

The `List` class is the implementation of a linked list that we have used as the running example in this thesis. We use the `List` class to evaluate polymorphic operations in the context of random heap access. Like the `ArrayBuffer` benchmarks, there is an `append` and `contains` microbenchmark. We will use lists to test the performance of polymorphic hash computations using `List.hashCode`.

```
class ArrayBuffer[T] {
    protected def initialSize: Int = 16
    var size0 = 0
    var array: Array[T] = newArray[T](Math.max(initialSize, 1))

    def length: Int = size0

    private def get(i: Int): T = array(i)
    private def set(i: Int, elem: T): Unit = array(i) = elem

    def contains(elem: T): Boolean = {
        var i = 0
        while (i < size0) {
            if (array(i) == elem) return true
            i += 1
        }
        false
    }

    def reverse(): Unit = {
        var pos = 0
        while (pos * 2 < size0) {
            swap(pos, size0 - pos - 1) // swaps two elements in the array
            pos += 1
        }
    }

    def append(elem: T): Unit = {
        val newSize0 = size0 + 1
        ensureSize(newSize0)
        set(size0, elem)
        size0 = newSize0
    }

    // Ensure that the internal array has at least `n` cells.
    def ensureSize(n: Int): Unit = {
        val arrayLength: Long = array.length // Use a Long to prevent overflows
        if (n > arrayLength) {
            var newSize: Long = arrayLength * 2
            while (n > newSize)
                newSize = newSize * 2
            // Clamp newSize to Int.MaxValue
            if (newSize > lang.Int.MaxValue) newSize = lang.Int.MaxValue

            val resized = newArray[T](newSize.toInt)
            var i = 0
            while (i < size0) {
                resized(i) = get(i)
                i += 1
            }
            array = resized
        }
    }
```

Figure 4.1: Code of the `ArrayBuffer` benchmark.

4.2 Methodology

Performance measurement of just-in-time compiled programs are is infamously difficult issue[39][35]. Many non-deterministic effects, such as speculative optimization, garbage collection, thread scheduling to name a few, affect the performance of programs executing on the Java Virtual Machine. As result, the JVM must be *warmed up* prior to measure of program performance. A benchmarking routine is warmed up with several iterations of invocations in order for profiling data to be collected and JIT compilation to finished. Therefore the measured performance of a microbenchmark will record the program executing the stable JIT compiled code instead of code executing in the interpreter.

Each benchmark method in this chapter is warmed up with 10 iterations of warmup lasting 10 seconds each. Results of these is measured in throughput, the number of executions that successfully completed in a second. The results are averaged from 10 measurements iterations for a period of 10 seconds each. We evaluate our microbenchmarks on input sizes between one hundred thousand and one million elements to account for factors of memory in our benchmarks. Each benchmark is run on three different implementations, Scala on GraalVM (Graal), the monomorphic interpreter (Mono), and the polymorphic interpreter (Poly).

4.3 Experimental Results

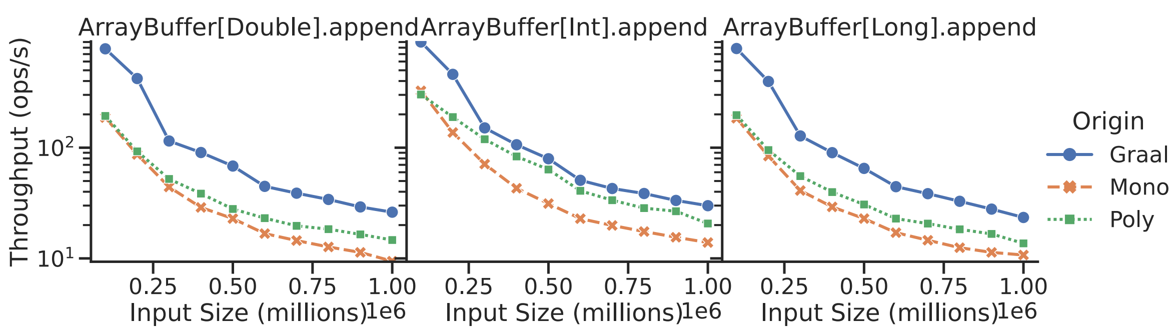


Figure 4.2: Benchmark results for `ArrayBuffer.append`.

The benchmark for `ArrayBuffer.append` inserts a sequence of elements into a newly initialized array buffer. This benchmark stresses array memory movement. Each time the backing array is too small for an additional element, the backing array is resized by creating a new larger and copying over existing elements. This resizing operation (`ensureSize` in 4.1) dominates the time spent in execution. Because of this bottleneck, executing compiled Scala bytecode on GraalVM is up to 4 times faster than the monomorphic and polymorphic interpreter.

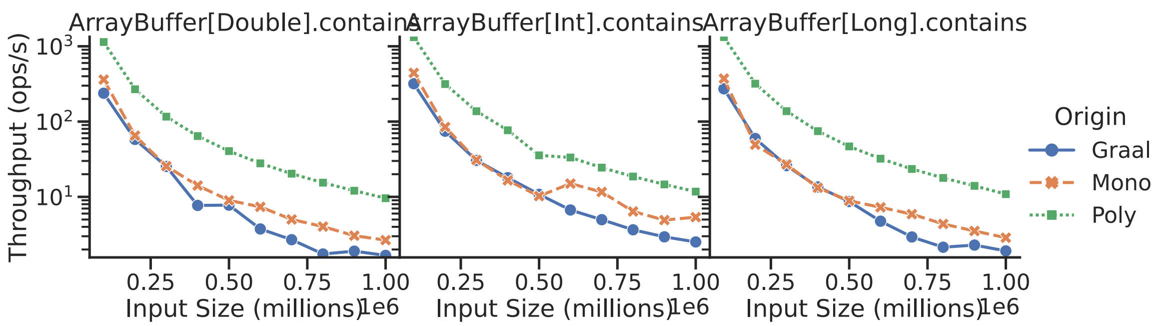


Figure 4.3: Benchmark results for `ArrayBuffer.contains`.

The `ArrayBuffer.contains` benchmark tests array operations in isolation. The benchmark checks an array buffer for the existence of a element. It exercises a polymorphic array access followed by a polymorphic equality operation (e.g. `(x: T) == (y: T)`). A polymorphic equality operator is dispatched the `equals` method of its left hand side argument. This results in the boxing of one or both arguments in equality checks between polymorphic values.

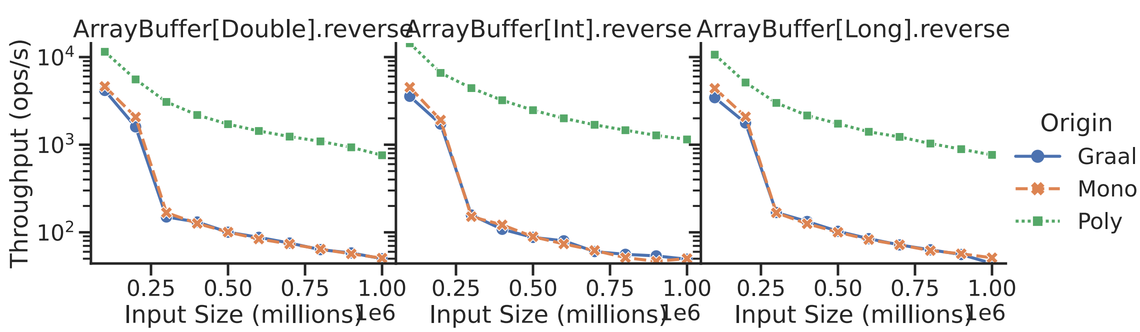


Figure 4.4: Benchmark results for `ArrayBuffer.reverse`.

`ArrayBuffer.reverse` reverses the order of the elements in the array buffer. Reversing an array is performance-bound by the loop of swap operations. A swap operation (given in 4.5) consists of two polymorphic value definitions (frame writes) initialized from polymorphic array accesses followed by the inverse of those two operations.

```
def swap(i: Int, j: Int): Unit = {
    val tmp1: T = get(i)
    val tmp2: T = get(j)
    set(i, tmp2)
    set(j, tmp1)
}
```

Figure 4.5: Code to swap two elements in an array buffer

The optimization of this microbenchmark proved to be the most difficult benchmark in terms of matching hand written monomorphic code in [70]. The performance between the monomorphic interpreter and GraalVM is roughly equal between all types; Neither implementation is able to specialize the polymorphic reads and array accesses. The polymorphic interpreter has up to 25 times more throughput than the monomorphic interpreter and GraalVM.

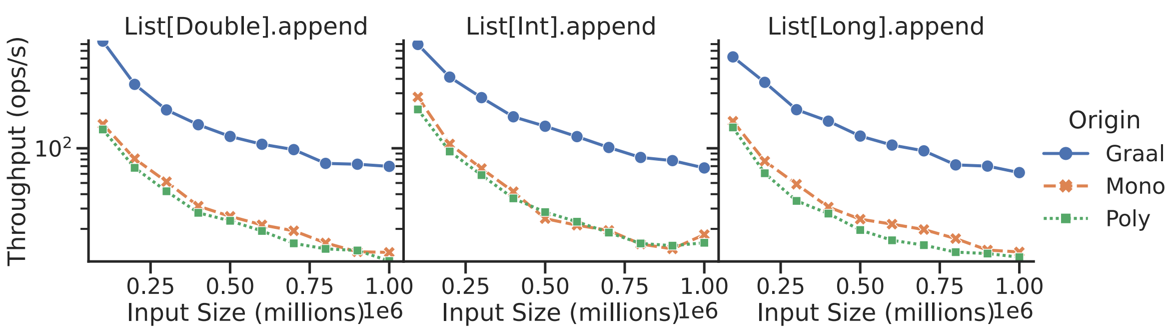


Figure 4.6: Benchmark results for `List.append`.

The `List.append` benchmark constructs a list from an array. As the creation of polymorphic instances is predominantly memory-bound and not compute-bound, there is no significant improvement in throughput from specialization. In fact, executing Scala via Java bytecode on the JMV results in substantially greater throughput.

Like `ArrayBuffer.contains` exercises the same performance-bottlenecks as the `ArrayBuffer.contai`
except under the context of random heap access for a list. The polymorphic interpreter

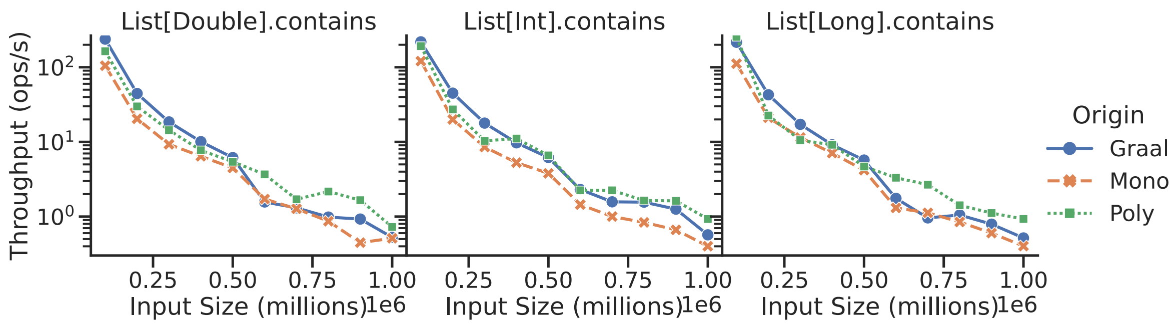


Figure 4.7: Benchmark results for `List.contains`.

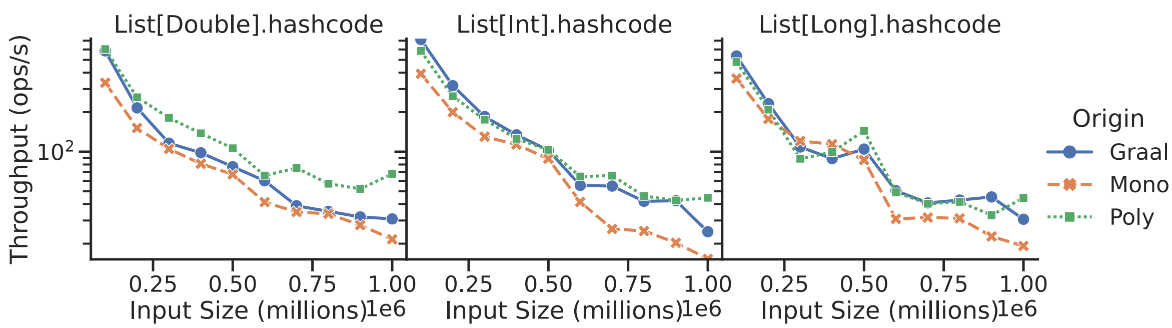


Figure 4.8: Benchmark results for `List.hashCode`.

```java
public static int anyHash(Object x) {
    if (x == null)           return 0;
    if (x instanceof Long)   return longHash(((Long) x).longValue());
    if (x instanceof Double) return doubleHash(((java.lang.Double) x).doubleValue());
    if (x instanceof Float)  return floatHash(((Float) x).floatValue());

    return x.hashCode();
}
```

Figure 4.9: Implementation of the `anyHash` function.

`List.hashCode` tests the specialization of the hash code function. Every class in Scala inherits the `hashCode` function from the $\top$ type. When the `hashCode` method is invoked in a polymorphic context, the Scala compiler inserts the `anyHash` bridge. The semantics of computing hash codes between on the same values with differing types such as `Int` and `Long` necessitates the insertion of this bridge, which complicates JIT complication.

Figure 4.9 gives the implementation of the `anyHash` function. In the `Int` and `Long` invocations of `List.hashCode`, the polymorphic interpreter keeps parity with the implementation of Scala on GraalVM. In the `Double` invocation of `List.hashCode`, the throughput on the polymorphic interpreter is 2.5 times greater than that the implementation of the monomorphic interpreter.

Chapter 5

Related Work

This chapter discusses previous academic and industrial work related to this thesis. The first section provides an introduction on the various implementations of parametric polymorphism The second section covers related work on the implementation on polymorphism in Java. The third section of this chapter provides an overview on previous and state-of-the-art efforts to specialize Scala. The last section presents prior and ongoing efforts in the implementation of other Truffle interpreters.

5.1 Implementations of Parametric Polymorphism

Implementations of parametric polymorphism can be divided into two broad categories[16]:

Homogeneous Translation This approach provides a single data representation for each polymorphic type. Examples of this implementation is the type erasure transformation applied in the Java and Scala compilation pipelines. Morrison et al. also refers to this form of polymorphism as the `uniform polymorphism`[?].

Heterogeneous Translation In contrast to homogeneous translation, the heterogeneous translation ensures there is a unique data representation for every instantiation of a polymorphic type. The heterogeneous translation can also be referred to as *textual polymorphism.*

In this section, we will cover various approaches in implementing parametric polymorphism in the context of these two forms. As polymorphism in Java and Scala are more relevant to the central themes of this thesis, we will first focus on implementations of parametric polymorphism for other languages.

Parametric polymorphism was first studied in functional programming languages[51][26]. Leroy proposed an approach where type coercion operations inserted between polymorphic operations and monomorphic data. The coercion operations in this approach is quite similar to the notion of boxing and unboxing, which Leroy describes as *wrapping* and *unwrapping*.

The heterogeneous translation is the more prevalent implementation of parametric polymorphism in object-oriented programming languages. The `template` concept in the C++ programming language popularized parametric polymorphism in objected-oriented programming languages. Templates define a generic definition of some kind in C++. The C++ compiler will generate heterogeneous translations based on every set of concrete type arguments supplied to during compilation. The implementation of polymorphism in the Common Language Runtime[50][49] by Kennedy and Syme makes use of reified types in a polymorphic bytecode IR during execution. Polymorphic class definitions are loaded as templates; Templates generate specialize class layouts on an ad-hoc basis based on the reified type arguments seen during bytecode execution. Their approach relies on extensions within the CLR to support types that is not present in existing JVM implementations. Our approach shares many similarities with the approach described by Kennedy and Syme. One drawback of their approach is the polymorphic bytecode IR does not support the full set of operations on types. For example, reflection is necessary to differentiate between a `List[Int]` and a `List[String]`. Our implementation differs as such operations are possible because the IR could potentially incorporate the full type language of TASTy.

5.2 Generics and Java

Prior efforts to implement generics in Java have been based on static compilation techniques restricted by the *open world assumption*. The open world assumption is an assumption that the program under compilation is *incomplete*, extra parts of the program will be supplied in a future iteration of compilation. This form of compilation is commonly known as *separate compilation*. As such, compilation results of the current parts of program must be interoperable with the compilation results of the remaining yet-to-be determined parts.

The Java language did not initially support parametric polymorphism in its initial release. As a result, many different approaches were proposed before a uniform polymorphism became the accepted implementation for Java. Pizza[56] was a superset of Java that supported heterogeneous and homogeneous translations of polymorphic definitions into Java. Agesen, Freund, and Mitchell proposed a heterogeneous translation for parametric polymorphism for Java during load-time instead of compile-time[4]. NextGen[18] separates the translation of polymorphic classes into monomorphic and polymorphic components. In NextGen, Only the polymorphic members of a class definition are specialized; These specialized classes inherit the implementation of their monomorphic members from a common parent class. Finally, GJ[?] proposed the foundations for what is now the accepted implementation of parametric polymorphism in Java. Polymorphic class definitions have a single uniform data representation after type erasure. All of these approaches determine the data representation of polymorphic definitions in a static context. Our approach is based on the *closed world assumption* as the entire program must be available in order for it to be executed.

5.3 Specialization in Scala

The standard implementation of parametric polymorphism follows that of Java, generic class definitions have their type parameters erased. All previous approaches attempt avoid the problem of bytecode explosion, where the specialization of polymorphic data with every possible type creates an exponential number of unique data representations. Dragos describes the earliest efforts to specialize Scala programs with the aid of annotations[28]. Annotations avoids unnecessarily specializing polymorphic data through knowledge injected by a programmer. Ureche, Talau, and Odersky expand upon this approach by reducing unnecessary duplication among specializations through sharing[70]. Sharing exploits the insight that specializations of some value types may be reused for the specializations of other value types. For example, the representation of `ArrayBuffer[Long]` could be used, with the addition of some glue code, for the specialization of `ArrayBuffer[Int]` instead of generating an additional specialized representation. Both approaches mix the implementation of uniform polymorphism with user-guided specialization directives. Our generates a heterogeneous translation of a generic class definition on an ad-hoc basis;

5.4 Truffle Interpreters

There many Truffle interpreters in active development at the time of writing. In this section, we will attempt to provide a brief survey of Truffle interpreters. TruffleRuby[62][25],FastR, Graal.js, Graal.Python,[42] are some of the industrial implementations of dynamically typed languages implemented with Truffle. They all make substantial use of Truffle facilities, some discussed earlier in this thesis, to speculative optimize program execution. Espresso[37] is an implementation of a Java bytecode interpreter in Truffle. Espresso is a metacircular implementation of a Java Virtual Machine. Because Espresso executes the same Java bytecode format as other JVM implementations, it makes use of same approaches to optimizing polymorphic data layout as the conventional implementation of Java on GraalVM.

Chapter 6

Future Work

TastyTruffle is intended to be framework for dynamic whole-program approaches to optimizing Scala. In this section, we discuss some possible extensions to the interpreter that further takes advantage of Truffle mechanisms. A substantial penalty of heterogeneous translations of polymorphic programs is *code explosion*. For large polymorphic programs, the penalty of heterogeneous translation is twofold; The first is the cost of increased memory usage. Having many specialized data representations incurs extra storage, unless of course, these data representations are regenerated every time a specialization is needed. The second is the hidden computational overhead of specialization. Like other computational overheads of managed runtimes such as garbage collection, time spent generating specialized variants of polymorphic classes or methods means time not spent executing the program. We propose several methods to augment *when* a specialization is created.

Many prior approaches to specialization have already attempted to minimize the number of specializations to mitigate performance degradation for complex polymorphic definitions, where is there often a $O(t^n)$ space complexity worse case[1], and very large programs. These approaches balance the tradeoff between performance and code size to optimistically generate *only* the specializations required to eliminate performance bottlenecks. Because of the work done in [28], many existing Scala programs are already user-annotated with a specialization directive. Similarly, our approach could be extended to include the semantics of this annotation, generating specializations with user-guide information only where needed. The translation of non-annotated definitions will simply use a shared type-erased data representation. However, mixing annotated and non-annotated programs will presents

[1] t is the number of value types combined with the reference type, n is the number of type parameters in a generic definition.

missed optimization opportunities in the non-annotated portions of the program.

Truffle offers many existing mechanisms for profiling values and types. Some of this profiling instrumentation is automatically done by Truffle, such as the profiling of argument types for node rewrites. While other instrumentation, such as condition profiles, are added by the guest language implementer. These profiles augment partial evaluation and enable speculative assumptions to augment optimizations such as conditional elimination. We propose instrumenting specialization sites to profile type arguments. Type argument profiles could then be used to decide the specific instantiation to specialize. A *profile-guided* approach to specialization could limit specializations to only the most frequently used instantiations.

Often a polymorphic instantiation is not sufficiently frequent to warrant specialization, the default homogeneous data representation will be shared among unspecialized instantiation. A type-erased homogeneous data representation may still be tagged with the underlying applied type. We can further augment type-erased polymorphic fields and frame slots to profile reads from and writes to their respective storage locations. These two pieces of dynamic information can be combined to allow the specialization of certain instantiations that are frequently manipulated, but not frequently created.

With inspiration from the work done by Ureche et al. in [70], we can apply the same optimization to sharing data layouts between certain specializations. Because the additional operations that adapt shared specializations to their original type contexts are simple neglible in terms of performance [70], these operations make sense to intrinsify as Truffle nodes that will be further optimized by JIT compilation.

Chapter 7

Conclusions

This thesis introduced TastyTruffle, a Truffle interpreter that is a platform for experimenting with ad-hoc data representations. The thesis described methods to translate TASTy, a tree serialization format for Scala 3, into an executable IR that is suitable for execution in an optimizing interpreter. We show in this thesis how to exploit the type information present in a input source language such as TASTy in order to generate specialized data representations for polymorphic data structures. We demonstrate that these techniques can substantially improve the performance of simple Scala programs in an experimental when compared to a state-of-the-art Java virtual machine.

A particular challenge in the implementation of TastyTruffle was the translation of TASTy into TastyTruffle IR. Because TASTy is emitted after parsing and type checking, no other compiler transformations typical in other intermediate representations are present. Many features of the Scala programming language are built as abstractions of simpler constructs that must be further simplified by the compiler. Without the existing compiler transformations to simplify these abstractions, TASTy can be at times *extraneously* high-level for the purposes of execution. While this did not significantly impact the evaluation of simple Scala programs for our experiments, it limits the *breadth* of programs that are executable by our interpreter. A possible solution to this hurdle is to read TASTy, perform a subset of Scala compiler transforms, then execute the program using our translation. While we will have to avoid the type erasure transformation and all subsequent transformations which depend on the results of type erasure, a much larger portion of Scala programs become available for execution on our interpreter.

The specialization of classes with both class-polymorphic and method-polymorphic semantics proved to be a complex implementation detail. The gap between the specialization of classes (at object creation) and the specialization of methods (at method invocation) required the selection of appropriate intermediate representation to encapsulate the *partial* specialization. Partial specializations have been specialized but also still contain polymorphic semantics which must resolved at a future specialization site. In this thesis, we chose to use a high-level approach to aid the translation of TASTy definition with TASTy type arguments. However, many approaches and mechanisms are possible to solve address this complexity. TODO(accepting ideas)

In this thesis we have evaluated TastyTruffle on simple but nonetheless difficult to specialization data structures exhibiting bulk memory access as well random heap access. The elimination of autoboxing in the list data structure resulted in incremental performance improvements where autoboxing proved to be a performance bottleneck. The elimination of autoboxing in the context of data structures back by polymorphic arrays resulted in performance improvements by an order of magnitude. TastyTruffle validates that there are many opportunities for data representation optimizations that bridge static compilation and just-in-time compilation.

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] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. *ACM SIGPLAN Notices*, 32(10):49–65, October 1997.

[5] Alfred V Aho, Jeffrey D Ullman, et al. *Principles of compiler design.* Addision-Wesley Pub. Co., 1977.

[6] 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.

[7] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, mar 1976.

[8] 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.

[9] John R. Allen and Ken Kennedy. Automatic loop interchange. *ACM SIGPLAN Notices*, 19(6):233–246, June 1984.

[10] B. Alpern, M. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88*, 1988.

[11] 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.

[12] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, nov 2003.

[13] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. *ACM SIGPLAN Notices*, 35(5):321–333, May 2000.

[14] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 146–158, New York, NY, USA, May 1997. Association for Computing Machinery.

[15] 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.

[16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, oct 1998.

[17] 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.

[18] Robert Cartwright and Guy L. Steele. Compatible genericity with run-time types for the Java programming language. *ACM SIGPLAN Notices*, 33(10):201–215, October 1998.

[19] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, may 1995.

[20] 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.

[21] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

[22] 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.

[23] 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.

[24] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[25] Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. Efficient and thread-safe objects for dynamically-typed languages. 11 2016.

[26] 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.

[27] 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.

[28] Iulian Dragos, editor. *Compiling Scala for Performance*. EPFL, Lausanne, 2010.

[29] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. February 2013.

[30] Gilles Duboscq, Thomas Würthinger, 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.

[31] 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.

[32] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[33] 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.

[34] 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.

[35] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, October 2007.

[36] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[37] Ekaterina Goltsova, editor. *Optimizing Java on Truffle*. 2022.

[38] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison-Wesley Professional, 2000.

[39] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 111–121, New York, NY, USA, June 2006. Association for Computing Machinery.

[40] 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.

[41] 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.

[42] Christian Humer. *Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters*. PhD thesis, Johannes Kepler University Linz, Linz, Austria, 2016. Publisher: Unpublished.

[43] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI '93*, 1993.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29(7), 2001.

[50] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29(7), 2001.

[51] 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.

[52] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371, jul 1991.

[53] Maurice Naftalin and Philip Wadler. *Java Generics and Collections: Speed Up the Java Development Process.* " O'Reilly Media, Inc.", 2006.

[54] 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.

[55] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[56] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 146–159, New York, NY, USA, January 1997. Association for Computing Machinery.

[57] 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.

[58] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[59] 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.

[60] Manuel Rigger, Roland Schatz, Jacob Kreindl, Christian Häubl, and Hanspeter Mössenböck. Sulong, and thanks for all the fish. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming'18 Companion, pages 58–60, New York, NY, USA, April 2018. Association for Computing Machinery.

[61] Ben Sander and AMD SENIOR FELLOW. Hsail: Portable compiler ir for hsa. In *Hot Chips Symposium*, volume 2013, pages 1–32, 2013.

[62] Chris Seaton. *Specialising Dynamic Techniques for Implementing the Ruby Programming Language | Research Explorer | The University of Manchester*. PhD thesis.

[63] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. *ACM SIGPLAN Notices*, 46(10):657–676, October 2011.

[64] 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.

[65] 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.

[66] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.

[67] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.

[68] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[69] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[70] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '13*, pages 73–92, Indianapolis, Indiana, USA, 2013. ACM Press.

[71] 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.

[72] 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.

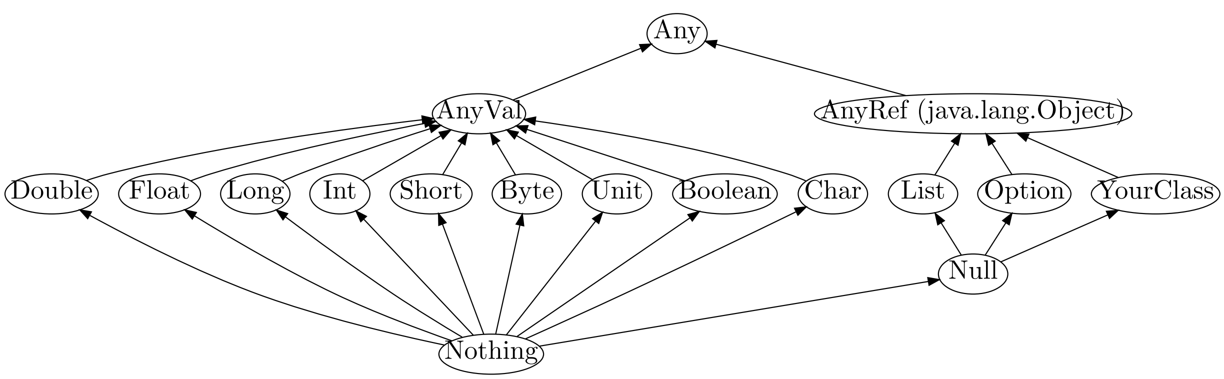
[73] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual machines, Languages, and Tools - PPPJ '14*, pages 133–144, Cracow, Poland, 2014. ACM Press.

[74] 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 Unified Type System



Appendix B

Scala 3 Compiler Phases

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

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

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

```
        new CheckStatic,            // Check restrictions that apply to @static members
        new BetaReduce,             // Reduce closure applications
        new init.Checker) ::        // Check initialization of objects
    List(
        new ElimRepeated,           // Rewrite vararg parameters and arguments
        new ExpandSAMs,             // Expand single abstract method closures to anonymous classes
        new ProtectedAccessors,     // Add accessors for protected members
        new ExtensionMethods,       // Expand methods of value classes with extension methods
        new UncacheGivenAliases,    // Avoid caching RHS of simple parameterless given aliases
        new ByNameClosures,         // Expand arguments to by-name parameters to closures
        new HoistSuperArgs,         // Hoist complex arguments of supercalls to enclosing scope
        new SpecializeApplyMethods, // Adds specialized methods to FunctionN
        new RefChecks) ::           // Various checks mostly related to abstract members and overriding
    List(
        // Turn opaque into normal aliases
        new ElimOpaque,
        // Compile cases in try/catch
        new TryCatchPatterns,
        // Compile pattern matches
        new PatternMatcher,
        // Make all JS classes explicit (Scala.js only)
        new sjs.ExplicitJSClasses,
        // Add accessors to outer classes from nested ones.
        new ExplicitOuter,
        // Make references to non-trivial self types explicit as casts
        new ExplicitSelf,
        // Expand by-name parameter references
        new ElimByName,
        // Optimizes raw and s string interpolators by rewriting them to string concatentations
        new StringInterpolatorOpt) ::
    List(
        new PruneErasedDefs,        // Drop erased definitions from scopes and simplify erased expressions
        new InlinePatterns,         // Remove placeholders of inlined patterns
        new VCInlineMethods,        // Inlines calls to value class methods
        new SeqLiterals,            // Express vararg arguments as arrays
        new InterceptedMethods,     // Special handling of `==`, `|=`, `getClass` methods
        new Getters,                // Replace non-private vals and vars with getter defs (fields are added later)
        new SpecializeFunctions,    // Specialized Function{0,1,2} by replacing super with specialized super
        new LiftTry,                // Put try expressions that might execute on non-empty stacks into their own methods
        new CollectNullableFields,  // Collect fields that can be nulled out after use in lazy initialization
        new ElimOuterSelect,        // Expand outer selections
        new ResolveSuper,           // Implement super accessors
        new FunctionXXLForwarders,  // Add forwarders for FunctionXXL apply method
        new ParamForwarding,        // Add forwarders for aliases of superclass parameters
        new TupleOptimizations,     // Optimize generic operations on tuples
        new LetOverApply,           // Lift blocks from receivers of applications
        new ArrayConstructors) ::   // Intercept creation of (non-generic) arrays and intrinsify.
    List(new Erasure) ::            // Rewrite types to JVM model, erasing all type parameters, abstract types and refineme
    List(
        new ElimErasedValueType,    // Expand erased value types to their underlying implmementation types
        new PureStats,              // Remove pure stats from blocks
        new VCElideAllocations,     // Peep-hole optimization to eliminate unnecessary value class allocations
        new ArrayApply,             // Optimize `scala.Array.apply([....])` and `scala.Array.apply(..., [....])` into `[...
        new sjs.AddLocalJSFakeNews, // Adds fake new invocations to local JS classes in calls to `createLocalJSClass`
        new ElimPolyFunction,       // Rewrite PolyFunction subclasses to FunctionN subclasses
        new TailRec,                // Rewrite tail recursion to loops
        new CompleteJavaEnums,      // Fill in constructors for Java enums
```

```
        new Mixin,                  // Expand trait fields and trait initializers
        // Expand lazy vals
        new LazyVals,
        // Add private fields to getters and setters
        new Memoize,
         // Expand non-local returns
        new NonLocalReturns,
        // Represent vars captured by closures as heap objects
        new CapturedVars) ::
    List(
        new Constructors,           // Collect initialization code in primary constructors
        // Note: constructors changes decls in transformTemplate, no InfoTransformers should be added after it
        new Instrumentation) ::     // Count calls and allocations under -Yinstrument
    List(
        // Lifts out nested functions to class scope, storing free variables in environments
        new LambdaLift,
        // Note: in this mini-phase block scopes are incorrect. No phases that rely on scopes should be here
        // Replace `this` references to static objects by global identifiers
        new ElimStaticThis,
        // Identify outer accessors that can be dropped
        new CountOuterAccesses) ::
    List(
        // Drop unused outer accessors
        new DropOuterAccessors,
        // Lift all inner classes to package scope
        new Flatten,
        // Renames lifted classes to local numbering scheme
        new RenameLifted,
        // Replace wildcards with default values
        new TransformWildcards,
         // Move static methods from companion to the class itself
        new MoveStatics,
        // Widen private definitions accessed from nested classes
        new ExpandPrivate,
         // Repair scopes rendered invalid by moving definitions in prior phases of the group
        new RestoreScopes,
        // get rid of selects that would be compiled into GetStatic
        new SelectStatic,
        // Generate JUnit-specific bootstrapper classes for Scala.js (not enabled by default)
        new sjs.JUnitBootstrappers,
        // Find classes that are called with super
        new CollectSuperCalls) ::
    Nil
```

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