

TastyTruffle: A Subtitle

by

James You

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

Waterloo, Ontario, Canada, 2022

© James You 2022

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

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

Abstract

This is the abstract.

Acknowledgements

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

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables	ix
List of Figures	x
Abbreviations	xii
1 Introduction	1
2 Background	2
2.1 Scala	2
2.2 Case Study: A List in Scala	3
2.3 Typed Abstract Syntax Trees	5
2.3.1 Definitions	5
2.3.2 Terms	6
2.3.3 Type Trees	6
2.4 Type Erasure	8
2.5 Java Bytecode	9
2.6 GraalVM	10
2.6.1 Truffle	11
2.6.2 Graal	12

3	Implementation	13
3.1	Execution	13
3.1.1	Generating Frame Slots from <code>ValDef</code> trees	14
3.1.2	The <code>DefDef</code> tree	15
3.1.3	Deriving Shapes from the <code>ClassDef</code> Tree	15
3.1.4	The <code>Assign</code> tree	16
3.1.5	Initializing Objects with the <code>New</code> Tree	16
3.1.6	The <code>Block</code> tree	16
3.1.7	The <code>While</code> tree	16
3.1.8	The <code>Select</code> tree	16
3.1.9	The <code>Apply</code> tree	16
3.1.10	The <code>Ident</code> tree	19
3.2	Specialization	19
3.2.1	19
3.2.2	Specializing Object Layout with <code>Applied</code> type trees	19
3.2.3	Specializing Call Sites with <code>TypeApply</code> trees	19
3.2.4	Specializing Terms	22
4	Evaluation	23
5	Related Work	24
5.1	Truffle Interpreters	24
5.2	Specializing Scala	24
5.3	Specializing Other Languages	24
6	Future Work	25
7	Conclusions	26
	References	27

APPENDICES	32
A Scala 3 Compiler Phases	33

List of Tables

List of Figures

2.1	Definition of <code>List</code> class	3
2.2	Implementation of <code>Con</code> class	4
2.3	Implementation of <code>Nil</code> class	4
2.4	TASTy in the context of the Scala compilation pipeline.	5
2.5	Pseudocode class definitions for a subset of TASTy trees.	6
2.6	Pseudocode class definitions for a subset of TASTy trees.	7
2.7	Pseudocode class definitions for a subset of TASTy type trees.	7
2.8	<code>Cons</code> class after type erasure	8
2.9	Example of autoboxing introduced for a list	9
2.10	GraalVM overview[12].	10
2.11	Adaptive optimization loop of GraalVM	11
3.1	Simplified <code>ValDef</code> tree	14
3.2	Simplified implementation of <code>FrameSlotKind</code>	14
3.3	Pseudocode for determining the frame slot kind of a type.	15
3.4	Simplified implementation of a shape.	15
3.5	Simplified implementation of the call node with a polymorphic inline cache used in TastyTruffle.	17
3.6	A possible polymorphic inline cache for a <code>List.contains</code> callsite.	18
3.7	A placeholder node for polymorphic code in TASTYTRUFFLE	19

3.8	Simplified implementation of generic dispatch node based on reified type arguments.	20
3.9	The typed dispatch chain for a <code>List.contains</code> call site	21
3.10	Graal IR of <code>List.head</code> after field read of <code>head0</code> is specialized.	22

Abbreviations

AST Abstract Syntax Tree [5](#)

DSL Domain Specific Language [11](#), [17](#)

IR Intermediate Representation [5](#), [14](#)

JIT Just-in-time [2](#), [9](#), [17](#)

JVM Java Virtual Machine [2](#), [15](#)

TASTy Typed Abstract Syntax Tree [2](#), [5](#), [13](#), [14](#)

Chapter 1

Introduction

Chapter 2

Background

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

2.1 Scala

Scala[30] is a object-oriented, generic and statically typed programming language. Scala uses a *pure* object-oriented programming model[17] and addresses many of the shortcomings[16] 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[32]. We describe the programming paradigms present in Scala in detail:

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 their class. Classes[10] 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*[37]. Polymorphic classes may define behavior independent of their state, allowing polymorphic classes to be reused extensively for multiple types of data.

Statically typed Static typing is a discipline where the type information about a program is known *before* it is executed. In order for a Scala program to compile successfully, it must be *well-typed*. 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. We believe these `List` implementations should represent a real-world use case as they are a scaled down and simplified version of the list implementation present in the Scala collections library. The `List` definition from the collections library is available by default to all Scala programs.

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

Figure 2.1: Definition of `List` class

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

```

1 case class Cons[+T](head: T, tail: List[T]) extends List[T] {
2     override def length: Int = 1 + tail.length
3
4     override def contains[T1 >: T](elem: T1): Boolean = {
5         var these: List[T] = this
6         while (!these.isEmpty)
7             if (these.head == elem) return true
8             else these = these.tail
9         false
10    }
11
12    override def hashCode(): Int = {
13        var these: List[T] = this
14        var hashCode: Int = 0
15        while (!these.isEmpty) {
16            val headHash = these.head.## // Compute hashCode
17            if (these.tail.isEmpty) hashCode = hashCode | headHash
18            else hashCode = hashCode | headHash >> 8
19            these = these.tail
20        }
21        hashCode
22    }
23 }

```

Figure 2.2: Implementation of Cons class

Figure 2.2 contains the implementation of a list node. The `Cons` implementation contains two polymorphic fields, `head` and `tail`. For specialization, we only care about `head` as a field does not change to store a `List[Int]` or `List[Double]`. On the other hand, the layout of the class would change if the `head` field was meant to store an integer instead of a byte.

```

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

```

Figure 2.3: Implementation of Nil class

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

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[25], optimization[25][9], or execution[26][27].

[Typed Abstract Syntax Tree \(TASTy\)](#) is a high-level [Intermediate Representation \(IR\)](#) which is produced and emitted after type checking phase of the Scala compiler (see appendix A). Figure 2.4 gives an overview of TASTy generation in the context of the Scala compilation pipeline, note that TASTy is only generated for Scala program sources. TASTy is a well-typed variation of an [Abstract Syntax Tree \(AST\)](#), an IR which is close to 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 this is significant in section 2.4.

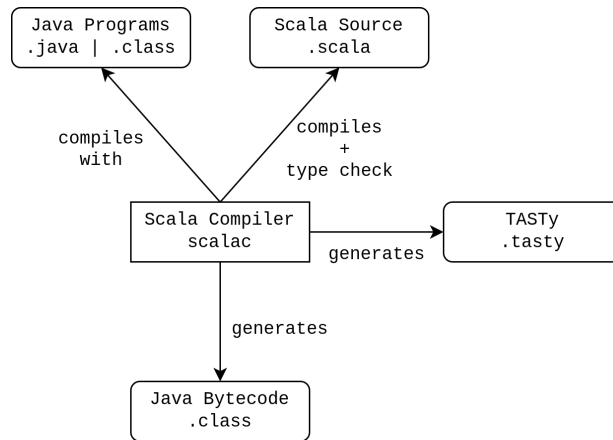


Figure 2.4: TASTy in the context of the Scala compilation pipeline.

In this thesis, we will use a limited subset of TASTy which is sufficient to represent the program 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, 2.6, and 2.7.

2.3.1 Definitions

```

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

```

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

Figure 2.5 will provides the pseudo implementations of all definitions in our subset of TASTy. Our subset of TASTy contains all definition trees from TASTy. For most of the definitions, they can be translated and be represented by a corresponding implementation in Truffle. A `ClassDef` always represents a top level class definition. A `DefDef` tree is always the definition of a method inside a class definition. Despite having every definition tree from regular TASTy available, we will omit the discussion of more complex definitions such as nested classes or closures.

2.3.2 Terms

Figure 2.6 gives the implementation for terms in our subset of TASTy. Terms represent an executable atom of code which returns a value. 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 such as object creation, instance method invocation, and instance field access. Our subset also supports the creation of polymorphic classes as well as the invocation of polymorphic methods.

2.3.3 Type Trees

```

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

```

Figure 2.6: Pseudocode class definitions for a subset of TASTy trees.

```

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

```

Figure 2.7: Pseudocode class definitions for a subset of TASTy type trees.

Type trees are a subset of trees which represent types as they are declared in Scala source code. Figure 2.7 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 monomorphic `ClassDef`. An `Applied` type tree represents a type constructor, which accept type arguments and produce a new type. For example, the `Cons[T]` would be represented as an applied type tree, where `Cons` would be the constructor and `T` would be the type argument. A `TypeBounds` tree represents the type expression `Lo <: T <: Hi`, a constraint where `T` must be a subtype of type `Hi` and supertype of type `Lo`. In the context of this thesis, we can use subtype to mean *subclass of* and supertype of

superclass of.

We will go over each tree and their relevance for execution in our interpreter in chapter 3. As part of normal Scala compilation pipeline, TASTy is eventually transformed by the Scala compiler and converted into Java bytecode. In the next section, we detail a critical compilation phase which enables the Scala compiler generate correct Java bytecode.

2.4 Type Erasure

Type erasure^[29] is a transformation which converts polymorphic classes methods in Scala to monomorphic classes and methods. This conversion is necessary because the JVM does not support polymorphic classes during runtime. Erasure ensures that any given polymorphic class and method has a single representation in practice. Type erasure is a crucial part of the Scala compilation which renders TASTy incomplete. Figure 2.8 shows the `Cons` class after type erasure.

```
1 case class Cons(head: Any, tail: List) extends List {
2   override def length: Int = 1 + tail.length
3
4   override def contains(elem: Any): Boolean = {
5     var these: List = this
6     while (!these.isEmpty)
7       if (these.head == elem) return true
8     else these = these.tail
9     false
10  }
11
12  override def hashCode(): Int = {
13    var these: List = this
14    var hashCode: Int = 0
15    while (!these.isEmpty) {
16      val headHash = these.head.##
17      if (these.tail.isEmpty) hashCode = hashCode | headHash
18      else hashCode = hashCode | headHash >> 8
19      these = these.tail
20    }
21    hashCode
22  }
23 }
```

Figure 2.8: `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^[30] abstract type

representing the super type of primitive and reference types. In Java bytecode, the `Any` type resolves to the `Object` type, the super type 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[26]. 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 generic value is expected. *Unboxing* occurs when a generic value must be used where a primitive value is expected. Figure 2.9 shows a simple example of inserted autoboxing operations when using the polymorphic `Cons` class after type erasure.

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

Figure 2.9: 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 a 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[34]. The elimination of this overhead through optimizing autoboxing operations is one of the central goals of this thesis.

2.5 Java Bytecode

Java bytecode is a portable and compact intermediate language and instruction set used by the Java Virtual Machine to execute programs. Traditionally, the JVM is responsible for much of the performance optimizations of Java program[33] through *Just-in-time* (JIT)

compilation. JIT compilation is an adaptive optimization which occurs after a program has begun execution. JIT compilation is concerned with optimizing and eliminating *hotspots* or portions of the program which are the slowest.

Many different JIT compilers[15][2] employ *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.

After most of Scala’s language constructs are eliminated from programs by the compiler, Java bytecode of the program is emitted. The resulting Java bytecode is considered an *incomplete* IR of Scala source programs, as the type information found in the program source or generated during 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.

2.6 GraalVM

GraalVM[41] is an implementation of a JVM. 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 components. The first is *Truffle*, a framework for translating the semantics of a source language, also called a *guest language*, to take advantage of the Graal infrastructure. The second is *Graal*, a language-agnostic JIT compilation infrastructure which handles speculative optimizations and generation of high-performance machine code.

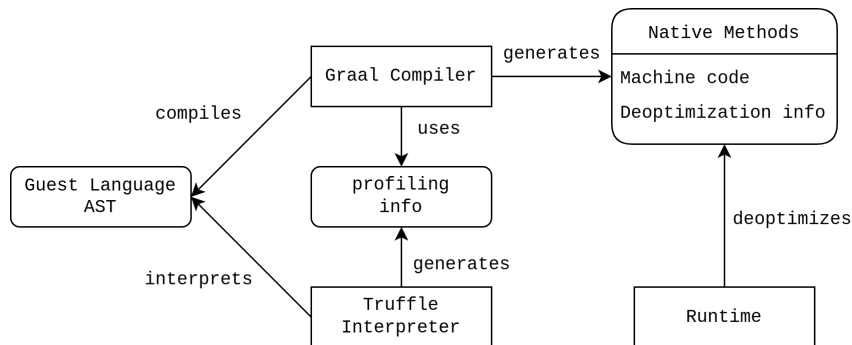


Figure 2.10: GraalVM overview[12].

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 Truffle

Truffle is a [Domain Specific Language \(DSL\)](#) framework for guest language implementation. A guest language is a language which is expected to run on Graal and requires an implementation in the *host language*, which provides the Truffle DSL. In this thesis, the guest language is TASTy (which represents Scala) and the host language is Java, the implementation language of Truffle. A guest language implementation always takes the form of an executable Truffle AST.

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 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 serves two purposes. The first is to dynamically incorporate guest language semantics into the executing program. The second is to augment the AST for JIT compilation. In this thesis, we will focus heavily on node rewriting the execution of TASTy trees in a Truffle interpreter to augment JIT compilation.

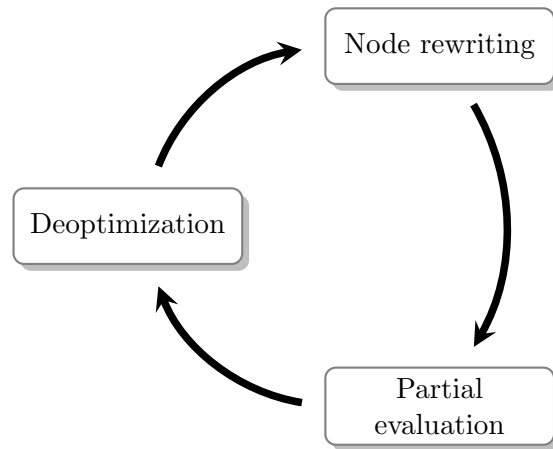


Figure 2.11: Adaptive optimization loop of GraalVM

2.6.2 Graal

After node rewriting, Graal JIT compiles Truffle ASTs into native machine code using *partial evaluation*. Partial evaluation is a program optimization technique for specializing an interpreter for a given program[14]. When a Truffle AST is submitted for compilation to Graal, it is considered *stable*. This AST, previously considered an input fed into the interpreter for execution, can now be considered static data in the interpreter. We can say that the partial evaluation of our interpreter with a given AST will produce an interpreter that can only execute the AST.

After partial evaluation, a Truffle AST is simplified to *Graal IR*[12]¹. Graal IR 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[8] and satisfies the *static single-assignment*[9] property. A sea of nodes is an abstraction based on a directed graph structure which relate the control flow graph[5] of the program to its data flow graph[4]. 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[21].

GraalIR enables Graal to speculatively compile only the *hot* branches[13], or branches that are most frequently taken, in the control flow portion of the IR and their transitive data dependencies. When a compiled program enters an unexpected state, execution is *deoptimized* and control of the program is transferred back to the interpreter. 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 environment.

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

Chapter 3

Implementation

This chapter is divided into two parts. The first half of this chapter will describe the execution of TASTy trees in a Truffle interpreter. The second half of this chapter covers the techniques we will use to eliminate autoboxing Truffle itself is unable to eliminate.

3.1 Execution

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

In the following sections, we will describe the features of TASTy and why it is directly unsuitable for execution and how to simplify their nodes into TastyTruffle IR. We will begin with an explanation of how data is encoded and defined in TASTy.

Types

Types are a set of properties and rules for reasoning about the behaviour of programs. In the Scala type system, types can be distinguished between *value types* and *type constructors*. Value types refer to the definition of a *class*. Type constructors accept type parameters as arguments and produce a resulting type.

3.1.1 Generating Frame Slots from ValDef trees

The `ValDef` tree is a multi-purpose node which represents value definitions in many contexts. This section will only cover the `ValDef` tree in the context of local variables. Sections 3.1.2 and 3.1.3 will cover the remaining contexts in which `ValDef` trees appear.

Local variables are variables which are bound to a *scope*. A scope represents the lifetime in which a variable can refer to an entity. 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). Unlike more simplified IR, abstract syntax trees do not encode any data dependence between definitions and uses[9]. In order to execute the tree, name binding must be resolved when ???

In `TASTy`, a local variable is represented by the `ValDef` tree node:

```
1    case class ValDef(name: String, tpt: TypeTree, rhs: Option[Term]) extends Tree
```

Figure 3.1: Simplified `ValDef` tree

The `ValDef` tree represents the site of a local variable declaration when the node is dominated by a `Block` node. A `ValDef` contains the simple, unqualified name of the declaration, the type as represented in the source program and the initializer. When a `ValDef` is owned by a `Block`, the initializer will always be non-empty.

Each unique variable declaration has a corresponding frame slot in the frame descriptor of its root node. Truffle permits each frame slot in a frame descriptor be described by a *frame slot kind*. At the time of writing, a frame slot kind can be implemented as:

```
1    object FrameSlotKind extends Enumeration {
2      type FrameSlotKind = Value
3      val Object, Long, Int, Double, Float, Boolean, Byte = Value
4    }
```

Figure 3.2: Simplified implementation of `FrameSlotKind`

There is a corresponding frame slot kind for each [JVM](#) primitive and reference types. We determine the frame slot kind of a type using the following method:

```
1  def getFrameSlotKind(tpe: Type): Option[FrameSlotKind] = {
2      if (tpe.isMonomorphic && tpe.isPrimitive)
3          Some(primitiveSlotKindOf(tpe))
4      else if (tpe.isParameter)
5          None
6      else
7          Some(FrameSlotKind.Object)
8  }
```

Figure 3.3: Pseudocode for determining the frame slot kind of a type.

Truffle specializes local variable access based on the variable’s type during partial evaluation[40]. To eliminate the need to specialize read and writes of variables where types are monomorphic and statically refer to a primitive type, the primitive frame slot kind is matched in the frame descriptor. In all other cases, including when the type is not resolvable through a single type parameter, e.g. `val x: T`, we assign the frame slot the `Object` frame slot kind. We will defer discussion of variable declarations which have polymorphic types that cannot be resolved statically until section [3.2](#).

3.1.2 The DefDef tree

3.1.3 Deriving Shapes from the ClassDef Tree

In object oriented programming languages, *objects* are instances of a class.

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

Figure 3.4: Simplified implementation of a shape.

3.1.4 The Assign tree

3.1.5 Initializing Objects with the New Tree

Escape Analysis

Escape analysis[22] reasons about the dynamic scope of object allocations. Compiler implementations often exploit the observations of escape analysis to enable optimizations such:

Region Allocation[7][39] The substitution of heap allocations with stack allocations to eliminate unnecessary garbage collection.

Scalar Replacement[23] The complete elimination of an object allocation, where the fields of the replaced object are substituted by local variables.

GraalVM employs *Partial Escape Analysis*[36], a path-sensitive variant of escape analysis which is particularly effective when combined with optimizations described above as well other compiler optimization such as inlining. Truffle offers guest language implementations the `VirtualFrame` abstraction to allow guest language semantics to take advantage of partial escape analysis and subsequent optimizations.

3.1.6 The Block tree

3.1.7 The While tree

3.1.8 The Select tree

3.1.9 The Apply tree

As previously mentioned, method invocations exists in multiple forms because tree canonicalization happens immediately after the TASTy picking phase in the compilation pipeline. The result is that TASTy trees retain some syntactic elements from their Scala sources. For example, Truffle provides two abstractions for call nodes, the *direct call node* is used when the call target can be statically resolved. In TASTy, this includes the set of methods with private or final modifiers[18] and class constructors. Otherwise, the *indirect call node*

is used for calls which have dynamically resolved call targets. TASTYTRUFFLE uses a singular call node implementation for both monomorphic and polymorphic calls. we utilize a polymorphic inline cache[19] to eliminate the overhead of resolving polymorphic calls for JIT compilation. Figure 3.5 shows a simplified Truffle call node in TASTYTRUFFLE which implements a polymorphic inline cache.

```

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

```

Figure 3.5: Simplified implementation of the call node with a polymorphic inline cache used in TastyTruffle.

The Truffle DSL emits a cache which is searched linearly based on the type of receiver. When the type of receiver has not been seen in the inline cache, an additional cache entry is generated and appended to the cache for the next call. The size of an polymorphic inline cache must be kept reasonable ????. The generated inline cache can be used to inline code and JIT optimized based on the type of the receiver seen at a call site.

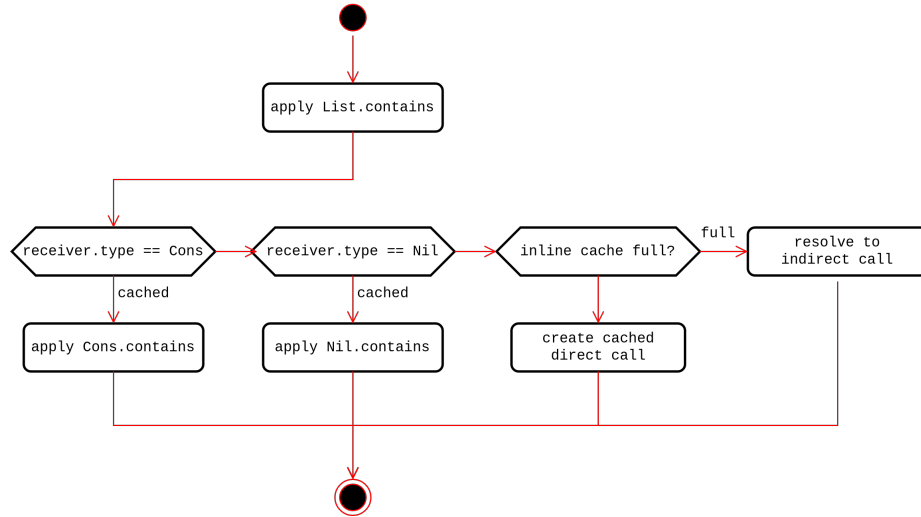


Figure 3.6: A possible polymorphic inline cache for a `List.contains` callsite.

When the polymorphic inline cache is applied to a monomorphic call site, it simplifies to a single element inline cache[11]. 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.

Unary and Binary Expressions

Unary and Binary operations in Scala are syntactic sugar for function invocation. For example, the following addition `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 underlying implementation on the JVM[26], e.g. in a bytecode instruction, are replaced by those instructions in compiled program bytecode. Similarly, TastyTruffle avoids implementing methods of primitive types with actual call semantics as primitive operations are frequently used and simple to optimize as intrinsic implementations exist on many Java virtual machines.[?]

3.1.10 The Ident tree

3.2 Specialization

3.2.1

3.2.2 Specializing Object Layout with Applied type trees

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

Figure 3.7: A placeholder node for polymorphic code in TASTYTRUFFLE

3.2.3 Specializing Call Sites with TypeApply trees

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

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

Method Parameters

Typed Dispatch Chains

Dispatch chains[?]

```

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

```

Figure 3.8: Simplified implementation of generic dispatch node based on reified type arguments.

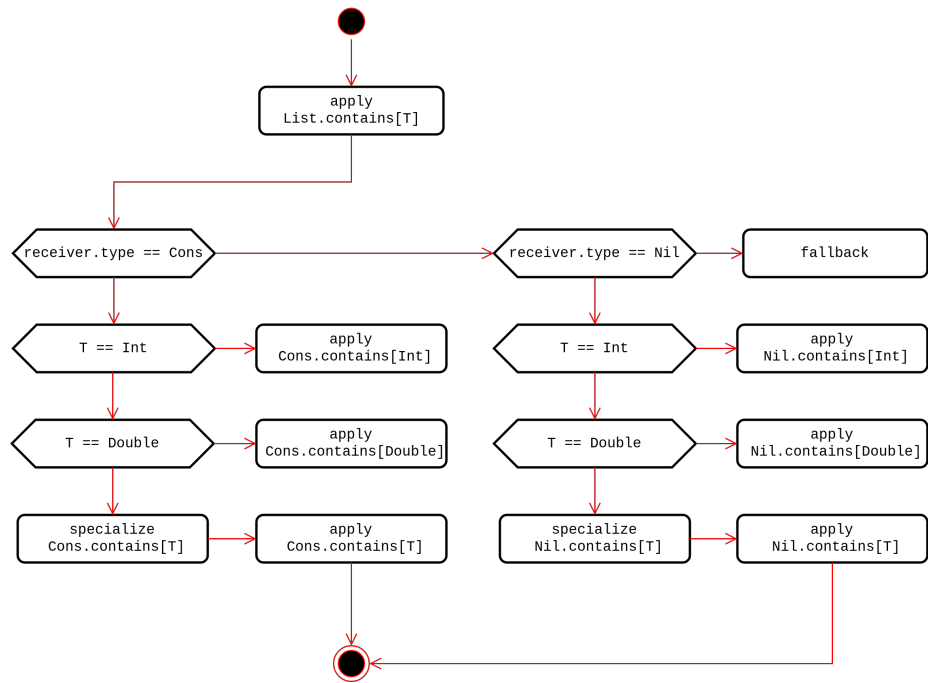


Figure 3.9: The typed dispatch chain for a `List.contains` call site

Code Duplication

Partial Evaluation

3.2.4 Specializing Terms

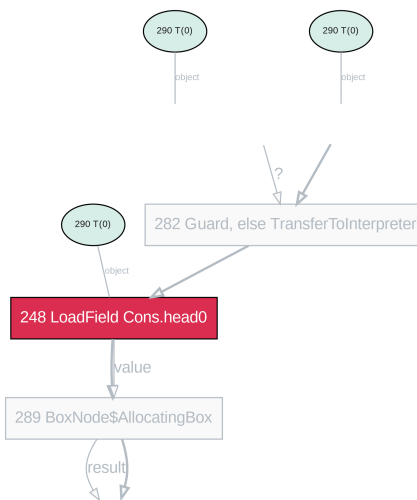


Figure 3.10: Graal IR of `List.head` after field read of `head0` is specialized.

The basic polymorphic unit of code in Scala are terms whose types are derived directly from a type parameter `T` or indirectly from a type constructor such as `Array[T]`. Polymorphic terms can be divided into the following categories:

Polymorphic local access

Polymorphic field access

Polymorphic method call

Polymorphic instantiation

Chapter 4

Evaluation

Chapter 5

Related Work

5.1 Truffle Interpreters

5.2 Specializing Scala

5.3 Specializing Other Languages

Chapter 6

Future Work

Chapter 7

Conclusions

References

- [1] Autoboxing and Unboxing (The Java™ Tutorials > Learning the Java Language > Numbers and Strings).
- [2] IBM Research | Technical Paper Search | The Jikes RVM Project: Building an Open Source Research Community(Search Reports), September 2016.
- [3] 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.
- [4] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, mar 1976.
- [5] 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.
- [6] 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.
- [7] Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25(6):713–775, nov 2003.
- [8] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, March 1995.

- [9] 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.
- [10] Ole-Johan Dahl and Kristen Nygaard. Simula: an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.
- [11] 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.
- [12] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal IR: An Extensible Declarative Intermediate Representation. February 2013.
- [13] 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.
- [14] Yoshihiko Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.
- [15] 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.
- [16] 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.
- [17] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.
- [18] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison-Wesley Professional, 2000.

- [19] 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.
- [20] Christian Humer. *Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters*. PhD thesis, Johannes Kepler University Linz, Linz, Austria, 2016. Publisher: Unpublished.
- [21] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI '93*, 1993.
- [22] 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.
- [23] 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.
- [24] Peter J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.
- [25] 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.
- [26] 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.
- [27] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29(7), 2001.
- [28] 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.

- [29] Maurice Naftalin and Philip Wadler. *Java Generics and Collections: Speed Up the Java Development Process.* " O'Reilly Media, Inc.", 2006.
- [30] 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.
- [31] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.
- [32] 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.
- [33] Michael Paleczny, Christopher Vick, and Cliff Click. The java {HotSpot™} server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.
- [34] 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.
- [35] Ben Sander and AMD SENIOR FELLOW. Hsail: Portable compiler ir for hsa. In *Hot Chips Symposium*, volume 2013, pages 1–32, 2013.
- [36] 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.
- [37] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.
- [38] Gerald Jay Sussman and Guy L Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.
- [39] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

- [40] 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.
- [41] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New ideas, New Paradigms, and Reflections on Programming & Software - Onward! '13*, pages 187–204, Indianapolis, Indiana, USA, 2013. ACM Press.

APPENDICES

Appendix A

Scala 3 Compiler Phases

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

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

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

```

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

```

```

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

```

```

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

```
