TastyTruffle: A Subtitle

by

James You

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

Waterloo, Ontario, Canada, 2022

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

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

Abstract

This is the abstract.

Acknowledgements

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

Dedication

This is dedicated to the one I love.

Table of Contents

List of Tables

List of Figures

Abbreviations

DSL Domain Specific Language 10

IR Intermediate Representation 4, 8

JIT Just-in-time 2, 10

JVM Java Virtual Machine 2, 8

TASTy Typed Abstract Syntax Tree 2, 6, 8

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 executable on their default platform the Java Virtual Machine (JVM). We will detail GraalVM Just-in-time (JIT) compiler infrastructure, an alternative JVM implementation. We then describe Truffle[10], a guest language implementation framework for GraalVM which we use to implement a runtime for Scala in this thesis.

Scala[17] is a objected-oriented, generic and statically typed programming language. Scala inherits the *pure* object-oriected programming model from Smalltalk. Scala is intended to address many of the shortcomings[7] in Java while still being *Java-like* for interoperability.

Scala is *object-oriented*. Every value in Scala is an object and every operation is method invocation on an object.

Scala is *generic*. ???

Scala is *statically typed*. Some of the behaviour in Scala programs can be verified to be correct by the Scala compiler before the program is executed.

2.1 Case Study: A List in Scala

```scala
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 an abstract `List` class

```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.##
            if (these.tail.isEmpty) hashCode = hashCode | headHash
            else hashCode = hashCode | headHash >> 8
            these = these.tail
        }
        hashCode
    }
}

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 2.2: Implementations of `List` class

2.2 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[14], optimization[14][4], or execution[15][16].

```
trait Tree {
    def symbol: Symbol
    def isExpr: Boolean
}
```

2.3 Type Erasure

```
abstract class List {
    def head: Any
    def tail: List
    def length: Int
    def isEmpty: Boolean = length == 0
    def contains(elem: Any): Boolean
}
```

Figure 2.3: Abstract `List` 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 = hashCode | headHash
            else hashCode = hashCode | headHash >> 8
            these = these.tail
        }
        hashCode
    }
}
```

Figure 2.4: `Cons` class after type erasure

2.4 Java Bytecode

2.5 GraalVM

2.6 Truffle

Chapter 3

Implementation

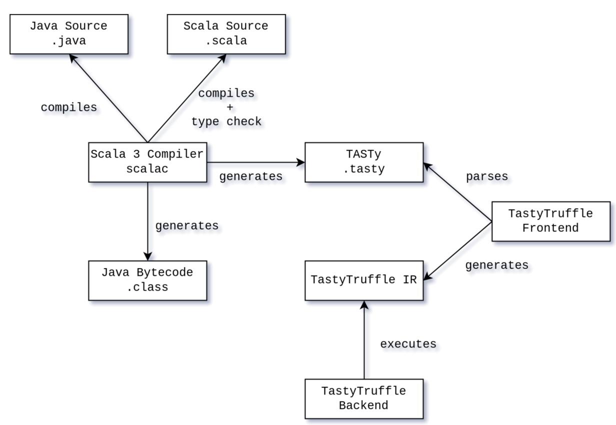

Figure 3.1: TastyTruffle in the context of the Scala compilation pipeline.

3.1 TastyTruffle Intermediate Representation

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

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 a 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.

Objects

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

Escape Analysis

Escape analysis[11] reasons about the dynamic scope of object allocations. Compiler implementations often exploit the observations of escape analysis to enable optimizations such:

Region Allocation[3][22] The substitution of heap allocations with stack allocations to eliminate unnecessary garbage collection.

Scalar Replacement[12] The complete elimination of an object allocation, where the fields of the replaced object are substituted by local variables.

GraalVM employs *Partial Escape Analysis*[20], 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.

Local Variables and Values

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[4]. In order to execute the tree, name binding must be resolved when ???

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

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

Figure 3.2: 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 intializer. When a `ValDef` is owned by a `Block`, the intializer 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:

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

Figure 3.3: 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:

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

Figure 3.4: Pseudocode for determining the frame slot kind of a type.

Truffle specializes local variable access based on the variable's type during partial evaluation[23]. 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.

Terms

Object Manipulation

Control Structures

Method Invocation

In object-oriented programming languages such as Scala, method invocation in polymorphic classes is resolved by the *dynamic dispatch* mechanism. Method invocation in TASTy is expressed via the following two term trees:

```
case class Select(qualifier: Term, selector: String) extends Term
case class Apply(applicator: Term, arguments: List[Term]) extends Term
```

Figure 3.5: Pseudocode for the `Select` and `Apply` trees.

As previously mentioned, method invocations exists in multiple forms because tree canocalization 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[8] 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 polymorhic calls. we utilize a polymorphic inline cache[9] to eliminate the overhead of resolving polymorphic calls for JIT compilation. Figure 3.6 shows a simplified Truffle call node in TastyTruffle which implements a polymorphic inline cache.

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

    final val INLINE_CACHE_SIZE: Int = 5;

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

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

Figure 3.6: 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.

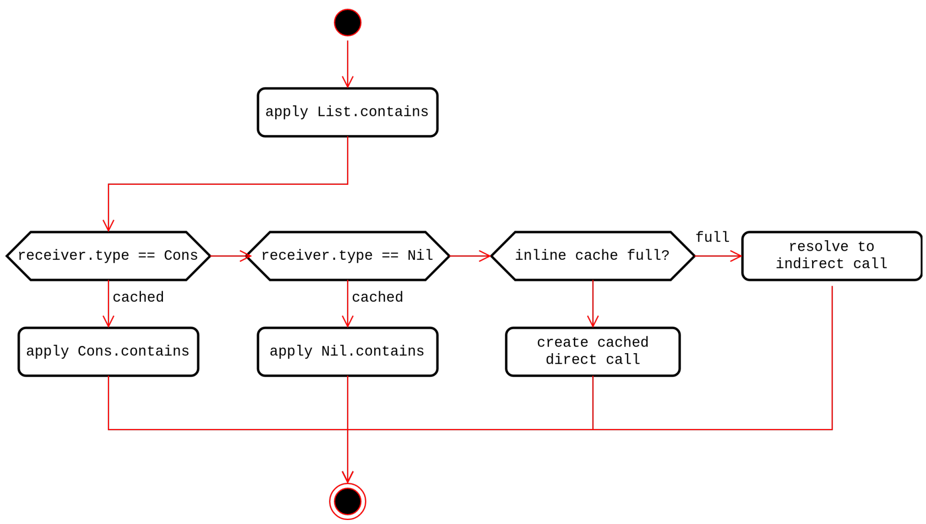

Figure 3.7: A possible polymorphic inline cache for a `List.contains` callsite.

When the polymorphic inline cache is applied to a monomorphic call site, it simplifies to a single element inline cache[5]. 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 primtive types and have an underlying implementation on the JVM[15], 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 semantcs as primitive operations are frequently used and simple to optimize as instrinsic implementations exist on many Java virtual machines.[?]

3.2 Specialization

3.3 Specializing Classes

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

Figure 3.8: A placeholder node for polymorphic code in TastyTruffle

3.3.1 Specializing Terms

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

Polymorphic local access

Polymorphic field access

Polymorphic method call

Polymorphic instantiation

3.3.2 Specializing Methods

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

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

Method Parameters

Typed Dispatch Chains

Dispatch chains[?]

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

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

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

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

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

Figure 3.9: Simplified implementation of generic dispatch node based on reified type arguments.

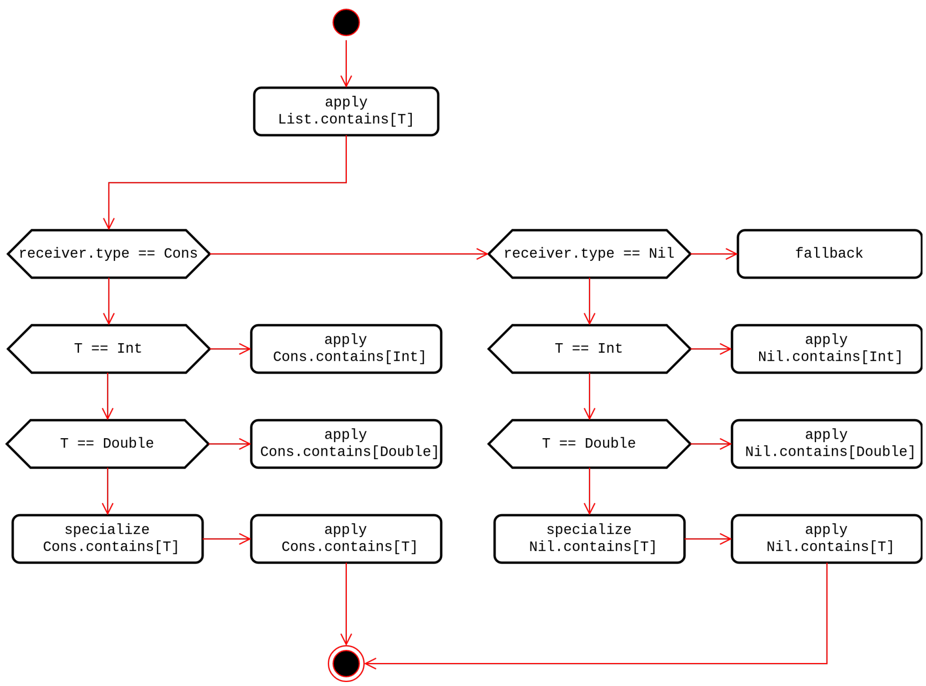

Figure 3.10: The typed dispatch chain for a `List.contains` call site

Code Duplication

Partial Evaluation

Chapter 4

Evaluation

Chapter 5

Related Work

Chapter 6

Future Work

Chapter 7

Conclusions

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

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

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

[20] 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.

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

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

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

APPENDICES

Appendix A

Scala 3 Compiler Phases

```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 v
        new ElimPackagePrefixes,    // Eliminate references to package prefixes in Select nodes
        new CookComments,           // Cook the comments: expand variables, doc, etc.
```

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