Miniphases: Compilation using Modular and Efficient Tree Transformations

Dmitry Petrashko
EPFL, Switzerland
dmitry.petrashko@gmail.com

Ondřej Lhoták
University of Waterloo, Canada
olhotak@uwaterloo.ca

Martin Odersky
EPFL, Switzerland
martin.odersky@epfl.ch

Abstract

Production compilers commonly perform dozens of transformations on an intermediate representation. Running those transformations in separate passes harms performance. One approach to recover performance is to combine transformations by hand in order to reduce number of passes. Such an approach harms modularity, and thus makes it hard to maintain and evolve a compiler over the long term, and makes reasoning about performance harder. This paper describes a methodology that allows a compiler writer to define multiple transformations separately, but fuse them into a single traversal of the intermediate representation when the compiler runs. This approach has been implemented in the Dotty compiler for the Scala language. Our performance evaluation indicates that this approach reduces the running time of tree transformations by 35% and shows that this is due to improved cache friendliness. At the same time, the approach improves total memory consumption by reducing the object tenuring rate by 50%. This approach enables compiler writers to write transformations that are both modular and fast at the same time.

CCS Concepts • **Software and its engineering** → *Compilers*

Keywords compiler performance, tree traversal fusion, cache locality

1. Introduction

Contemporary compilers are complicated, consisting of thousands to millions of lines of code. The design of a compiler is constrained by multiple competing requirements, and it is challenging to satisfy all of them simultaneously. A compiler needs to be correct, and therefore easy to test. A compiler needs to be maintainable and easy to debug. To serve both of these needs, the design of the compiler should be modular. But a compiler also needs to be fast. Compiling a complicated programming language is computationally expensive, but software developers run their compilers many times during development, and waiting for the compiler hinders their productivity. A good compiler design provides both modularity and performance at the same time.

Balancing modularity and performance has been a difficult and long-running challenge in the compiler for the Scala programming language. Compilation times have been a frequent complaint from users. On many occasions, compiler developers had to make difficult trade-offs between modularity, maintainability, and performance.

Most compilers are composed of a sequence of transformations of some intermediate representation of the program being compiled. Often, a core part of the intermediate representation is an abstract syntax tree.

In this paper, we propose a new design for tree transformations that is both modular and efficient at the same time. This design is adopted in the Dotty compiler for Scala. We present the design to demonstrate its modularity and we empirically evaluate its performance in the Dotty compiler.

For modularity, each transformation of the intermediate representation should be expressed as an independent traversal of the abstract syntax tree. However, the tree is much too large to fit in cache, so each traversal of the whole tree is expensive. Our solution enables the compiler developer to implement, test, and reason about transformations as separate traversals. However, our approach fuses the transformations performed at individual tree nodes so that multiple logical transformation passes (“Miniphases”) are performed in a single traversal of the abstract syntax tree.

The remainder of this paper is organized as follows:

- Section 2 shows the conflict between modularity and performance based on experience with Scala 2.x compilers;
- Section 3 presents performance characteristics that we targeted when designing the Miniphases framework;
- Section 4 introduces proposed design abstractions and describes the implementation inside the Dotty compiler;

PLDI '17, June 19-24, 2017, Barcelona, Spain

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4988-8/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3062341.3062346

- Section 5 presents results of experiments that evaluate the impact of the Miniphases framework on GC object promotion rate and CPU cache misses;
- Section 6 covers limitations of the framework and soundness of fusion;
- Section 7 discusses real-world experience with the framework, such as maintenance cost and the on-boarding process for new contributors;
- Section 8 presents related work;
- Section 9 concludes.

2. Background: Scala Compilers

The current Scala compiler has been the production compiler since version 2.0 of Scala in 2006. The Miniphase approach that we study in this paper is being implemented in Dotty, a next-generation compiler for experimenting with new language features and compiler designs for Scala.

Both compilers share the following common structure. The major internal data structures are trees, which describe the syntax of the program being compiled, and are gradually transformed by the compiler pipeline, and types and symbols, which describe semantic information and the relationships between program entities. The program being compiled is represented as a sequence of compilation units. Every compilation unit is a single source-file which may define multiple top-level classes.

The tree nodes in both compilers are logically immutable and do not have a link to their parent node. This allows to reuse trees in multiple locations, and simplifies debugging as no mutation to trees is possible. When trees are modified, they are rebuilt using copiers. An optimization avoids the copying in the (quite common) case where a transform returns a tree with the same fields as its input.

Symbols are unique identifiers for definitions, including members and local variables, coming both from sources currently being compiled as well as their binary dependencies. Types are used not only to describe the type of an entity, but can also serve as references to program definitions such as methods or variables. In the Dotty compiler, this has been generalized to a point where *all* references to other program parts are embodied in types. This is possible, and convenient, because the Scala type system includes singleton types [18], which guarantee that an expression has the same value as some entity such as a field or variable, and are thus equivalent to references to those fields and variables. Types also encode constants [13] and higher-kinded types.

An execution of the compiler is broadly divided into the front-end, the tree transformation pipeline, and code generator. The front-end parses and type-checks source code, and generates trees annotated with type information. The tree transformations gradually desugar and lower the Scala-like code to a simpler form that is close to Java bytecode. The code generator emits Java bytecode from the lowered trees.

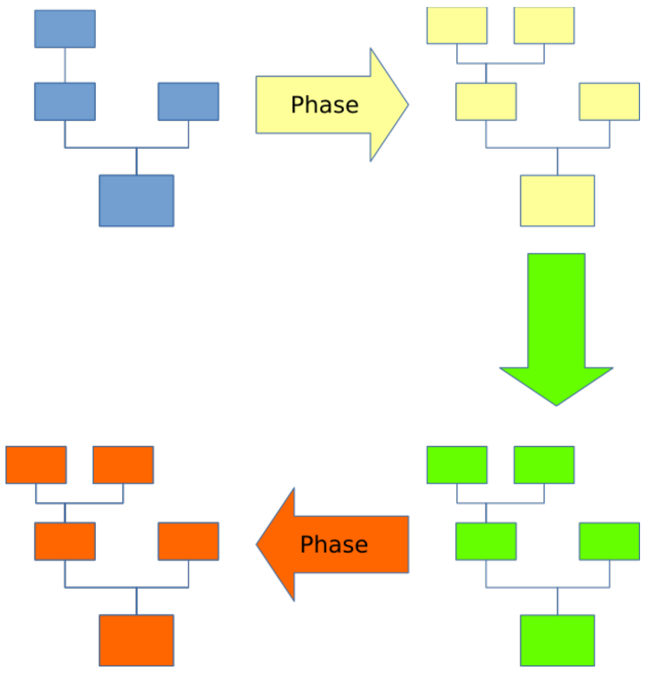


Figure 1: Mega-phase based transformation of a tree

In this paper, our focus is on the middle phases which consitute the tree transformation pipeline.

2.1 Experience with the Scala Compiler

In this section, we review experience from the past ten years of developing the Scala compiler, focusing especially on modularity and performance.

The compiler that has been used for Scala versions 2.0 to 2.12 is organized as a sequence of phases. Each phase is a function that takes the tree of a compilation unit as input and returns a transformed tree as output. The implementation of each phase can be arbitrary Scala code, and there are no restrictions on how it, for example, traverses the tree. This *Megaphase* approach is illustrated in Figure 1. In the compiler for Scala version 2.12.0, there are 24 such phases, listed in Table 1.

The Megaphase approach was originally intended to be modular in that each phase is an independent transformation of the tree.

A drawback is that each phase that implements a specific language feature must traverse the entire tree to find uses of that feature. When a use of the feature is found, the phase transforms the relevant tree node. All ancestor nodes are also rebuilt because the tree is immutable. For example, the program in Listing 1 uses pattern matching, lazy vals, and mixins. To compile this program, at least five transformations are needed to implement the three language features, to create a constructor for the class `Increment`, and to normalize the method `interfaceMethod` to take an empty list of arguments. When implemented as independent Megaphases, each of these transformations must traverse the entire tree.

phase name	id	description
parser	1	parse source into ASTs, perform simple desugaring
namer	2	resolve names, attach symbols to named trees
packageobjects	3	load package objects
typer	4	the meat and potatoes: type the trees
patmat	5	translate match expressions
superaccessors	6	add super accessors in traits and nested classes
extmethods	7	add extension methods for inline classes
pickler	8	serialize symbol tables
refchecks	9	reference/override checking, translate nested objects
uncurry	10	uncurry, translate function values to anonymous classes
fields	11	synthesize accessors and fields, including bitmaps for lazy vals
tailcalls	12	replace tail calls by jumps
specialize	13	@specialized-driven class and method specialization
explicitouter	14	this refs to outer pointers
erasure	15	erase types, add interfaces for traits
posterasure	16	clean up erased inline classes
lambdalift	17	move nested functions to top level
constructors	18	move field definitions into constructors
flatten	19	eliminate inner classes
mixin	20	mixin composition
cleanup	21	platform-specific cleanups, generate reflective calls
delambdafy	22	remove lambdas
jvm	23	generate JVM bytecode
terminal	24	the last phase during a compilation run

Table 1: Phases in Scala 2.12.0

```scala
trait Interface {
  def interfaceMethod = 1
  lazy val interfaceField = 2
}

class Increment(by: Int) extends Interface {
  def incOrZero(b: Any) = b match {
    case b: Int => b + by
    case _ => 0
  }
}
```

Listing 1: Sample Scala program

In this example, each of the phases changes only a single node in the tree, yet five traversals are needed to change five nodes.

To improve performance, consecutive phases have been joined at the source level by hand, making the resulting phase contain code to perform multiple transformations at once. Even though the Megaphase design was intended to be modular, performance considerations pressured the developers to mix unrelated transformations in individual phases. This reduction in the number of phases makes the compiler faster, at a cost of hard-to-predict interactions between different transformations. Over the years, this has led to a code-base that is hard to maintain and evolve.

For example, Scala supports method definitions with multiple argument lists. The phase called `uncurry` was originally written to flatten the argument lists in such definitions into a single list of arguments. For the sake of performance, several unrelated transformations were added to this phase. In particular, this phase also finds **`try`** blocks used as subexpressions of some expression and lifts them into separate methods. This transformation is necessary because Java try blocks are statements, not expressions, so the JVM implementation of exception handlers does not provide a way to communicate an expression context from the try block to the exception handler. This transformation is completely unrelated to the original purpose of the `uncurry` phase. In the Dotty compiler, this transformation is done in its own Miniphase called LiftTry.

As another example, the Scala compiler contains a phase called `refchecks`, originally written to check that overriding methods conform to the types of the superclass methods that they override. Originally, the phase was intended to only inspect but not modify the tree. However, the current implementation of this phase performs multiple transformations of the tree. In particular, it replaces local (singleton) object definitions by local variables containing the object, it replaces calls to factory methods with calls to class constructors, and it eliminates conditional branches when their condition is statically known. None of these transformations

phase name	id	description
FrontEnd	1	Compiler frontend: scanner, parser, namer, typer
sbt.ExtractDependencies	2	Sends information on classes' dependencies to sbt via callbacks
PostTyper	3	Additional checks and cleanups after type checking
sbt.ExtractAPI	4	Sends a representation of the API of classes to sbt via callbacks
Pickler	5	Generate TASTY info
FirstTransform	6	Some transformations to put trees into a canonical form
CheckReentrant	7	Internal use only: Check that compiled program has no data races involving global vars
RefChecks*	8	Various checks mostly related to abstract members and overriding
CheckStatic*	9	Check restrictions that apply to @static members
ElimRepeated*	10	Rewrite vararg parameters and arguments
NormalizeFlags*	11	Rewrite some definition flags
ExtensionMethods*	12	Expand methods of value classes with extension methods
ExpandSAMs*	13	Expand single abstract method closures to anonymous classes
TailRec*	14	Rewrite tail recursion to loops
LiftTry*	15	Put try expressions that might execute on non-empty stacks into their own methods
ClassOf*	16	Expand 'Predef.classOf' calls.
TryCatchPatterns*	17	Compile cases in try/catch
PatternMatcher*	18	Compile pattern matches
ExplicitOuter*	19	Add accessors to outer classes from nested ones.
ExplicitSelf*	20	Make references to non-trivial self types explicit as casts
CrossCastAnd*	21	Normalize selections involving intersection types.
Splitter*	22	Expand selections involving union types into conditionals
VCInlineMethods*	23	Inlines calls to value class methods
IsInstanceOfEvaluator*	24	Issues warnings when unreachable statements are present in match/if expressions
SeqLiterals*	25	Express vararg arguments as arrays
InterceptedMethods*	26	Special handling of '==', '!=', 'getClass' methods
Getters*	27	Replace non-private vals and vars with getter defs (fields are added later)
ElimByName*	28	Expand by-name parameters and arguments
AugmentScala2Traits*	29	Expand traits defined in Scala 2.11 to simulate old-style rewritings
ResolveSuper*	30	Implement super accessors and add forwarders to trait methods
ArrayConstructors*	31	Intercept creation of (non-generic) arrays and intrinsify.
Erasure	32	Rewrite types to JVM model, erasing all type parameters, abstract types and refinements.
ElimErasedValueType*	33	Expand erased value types to their underlying implmementation types
VCElideAllocations*	34	Peep-hole optimization to eliminate unnecessary value class allocations
Mixin*	35	Expand trait fields and trait initializers
LazyVals*	36	Expand lazy vals
Memoize*	37	Add private fields to getters and setters
LinkScala2ImplClasses*	38	Forward calls to the implementation classes of traits defined by Scala 2.11
NonLocalReturns*	38	Expand non-local returns
CapturedVars*	39	Represent vars captured by closures as heap objects
Constructors*	40	Collect initialization code in primary constructors
FunctionalInterfaces*	41	Rewrites closures to implement @specialized types of Functions.
GetClass*	42	Rewrites getClass calls on primitive types.
LambdaLift*	43	Lifts out nested functions to class scope, storing free variables in environments
ElimStaticThis*	44	Replace 'this' references to static objects by global identifiers
Flatten*	45	Lift all inner classes to package scope
RestoreScopes*	46	Repair scopes rendered invalid by moving definitions in prior phases of the group
ExpandPrivate*	47	Widen private definitions accessed from nested classes
SelectStatic*	48	get rid of selects that would be compiled into GetStatic*
CollectEntryPoints*	49	Find classes with main methods
CollectSuperCalls*	50	Find classes that are called with super
DropInlined*	51	Drop Inlined nodes, since backend has no use for them
MoveStatics*	52	Move static methods to companion classes
LabelDefs*	53	Converts calls to labels to jumps
GenBCode	54	Generate JVM bytecode

Table 2: Phases in Dotty compiler. The horizontal lines indicate blocks of Miniphases(*) that constitute a single transformation.

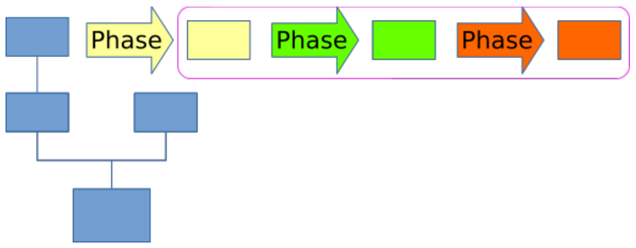


Figure 2: Pipelining of a leaf-node through Miniphases

are related to the original purpose of the `refchecks` phase, or to each other.

In this paper, we propose a framework that removes the need to make this trade-off. The proposed framework allows separate transformations to be defined in separate phases, yet applies the transformations in a common traversal of the tree for performance. Thus, it frees compiler developers from the pressure to combine unrelated transformations in the same phase.

Currently, the code of the Dotty compiler is modularized into 54 phases, listed in Table 2. We expect that the number of phases could increase to around 100 once the compiler is finished.

3. Target Performance Characteristics

While designing the framework, we had approximate performance characteristics in mind.

Based on user feedback on existing versions of the Scala compiler, we would like to be able to compile about 4000 lines per second (on a MacBook Pro 14', 2014). The current `scalac` compiler can compile 1000–2000 lines per second on such a machine, depending on the application being compiled.

The tree transformation pipeline uses about one-third of the compilation time. The rest of the time is spent in the typechecker and the code generator, which are independent of the tree transformation pipeline. Thus, the tree transformations should process 12000 lines of code per second. A typical line of code corresponds to about 12 tree nodes. We estimate that the compiler performs about 100 distinct transformations, each of which justifies a separate phase. We would like the framework to spend no more than 20% of the time traversing the tree, leaving 80% of the time for useful transformations. Thus, a Megaphase approach would need to visit each node in about 14 nanoseconds, or 28 CPU cycles. If we can perform the 100 transformations in only 10 traversals, we can use 140 nanoseconds, or 280 CPU cycles per tree node visit.

4. Design

Listing 2 presents a simplified structure of the tree nodes used in the Dotty compiler. Each tree node has a `withNewChildren` method that creates a new node with a modified list of children.

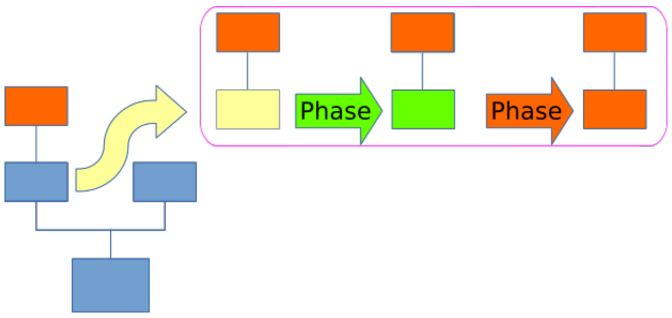


Figure 3: Pipelining of an inner-node through Miniphases

```
abstract sealed class Tree {
  def tpe: Type
  def withNewChildren(list: List[Tree]): Tree
  def children: List[Tree]
}
class Ident(sym: Symbol) extends Tree
class Select(from: Tree, name: String) extends Tree
...
class ValDef(sym: Symbol, rhs: Tree) extends Tree
class DefDef(sym: Symbol, rhs: Tree) extends Tree
class CompilationUnit(trees: List[Tree]) extends
    Tree
```

Listing 2: Tree nodes

```
def compileUnits(units: List[CompilationUnit],
    phases: List[Phase]) = {
  var units1 = units
  for (phase <- phases)
   units1 = units1.map(unit => phase.runPhase(unit)
    )
}
```

Listing 3: Overall traversal

The tree transformation pipeline has the overall structure given in Listing 3. For each phase, and for each compilation unit, the compiler applies the phase to the compilation unit. In the Miniphase approach, this high-level structure remains the same. However, multiple Miniphase transformations are fused together and performed in a single phase.

To support this fusion, all Miniphases must traverse the tree in a consistent order. A Miniphase is therefore implemented as a phase whose `runPhase` does a postorder traversal over the tree, as shown in Listing 4. When visiting each node, it calls the `transform` method, which dispatches to a specific node transformation function depending on the type of the tree node. By default, the node transformations are all identity methods. An implementation of a specific transformation is expected to override the transformation methods of the types of node relevant to the transformation.

The advantage of imposing a uniform postorder traversal is that multiple Miniphases can now be fused together, after being combined by functions presented in Listing 5. The

```scala
class Phase {
  def runPhase(t: Tree): Tree

  val runsAfter: Set[MiniPhase] = Set.empty
  def checkPostCondition(t: Tree): Boolean = true
}

class MiniPhase extends Phase {
  val valDefTransform: ValDef => Tree = id
  val defDefTransform: DefDef => Tree = id
  val identTransform: Ident => Tree = id
  ...
  val selectTransform: Select => Tree = id

  final def transform(t: Tree) = t match {
    case a: ValDef => valDefTransform(a)
    case a: DefDef => defDefTransform(a)
    ...
    case a: Select => selectTransform(a)
  }

  final def runPhase(t: Tree): Tree = {
    val newChildren =
      t.children.map(sub => runPhase(sub))
    val reconstructed = t.withNewChildren(newChildren)
    transform(reconstructed)
  }
}
```

Listing 4: Definition of a Miniphase

```scala
private def chainMiniPhases(first: MiniPhase, second: MiniPhase) = {
  new MiniPhase {
    val valDefTransform =  { x: ValDef =>
      val newTree = first.valDefTransform(x)
      second.transform(newTree)
    }

    ... // similar to valDefTransform for all node kinds

    val runsAfter: Set[MiniPhase] =
      second.runsAfter -- first ++ first.runsAfter

    def checkPostCondition(t: Tree) =
      first.checkPostCondition(t)  &&
      second.checkPostCondition(t)
  }
}

def combine(a: Array[MiniPhase]): MiniPhase =
  a.reduceRight((phase, acc) =>
    chainMiniPhases(phase, acc)
  )
```

Listing 5: Fusion algorithm for Miniphases

fused Miniphase traverses the tree only once. While visiting each tree node, it applies the transformations implemented by all of its constituent Miniphases. The `valDefTransform` method applies the `valDefTransform` method of the first Miniphase (and similarly for other node types), but for subsequent Miniphases, it must call the general `transform` method, because the first Miniphase might have changed the type of the node. This is illustrated in Figures 2 and 3. In Figure 2, the blue leaf node is transformed by three Miniphases (yellow, green, orange), yielding an orange node, before any of the other blue nodes are processed. In the next step, in Figure 3, the parent of the now orange node is processed by the same three Miniphases.

A set of fused Miniphases has the following properties, which must be taken into account by implementors:

- The `transform` method is called on all nodes of the compilation unit in a post-order traversal order.
- When the `transform` method of Miniphase m is called on a tree node t, t has already been transformed by all Miniphases that come before m, and the children of t have been transformed by all Miniphases that have been fused with m, including ones that come both before and after m. In Figure 3, the yellow and green Miniphases process a node whose child is already orange, even though the orange Miniphase comes after the green one. Though it is surprising that Miniphase m "sees the future" in its child subtrees, we have found that this rarely creates any problems, since most phases simplify the trees and introduce new invariants and rarely break existing ones.

We will discuss in Section 6 the criteria that developers of transformation phases must consider in deciding whether a phase can be fused with other phases.

Two important optimizations can be applied to the basic fusion technique. Both these optimizations are shown in the modified version of the Miniphase fusion implementation given in Listing 6.

First, since most Miniphases transform only a small subset of the types of tree nodes, the fusion code explicitly checks (Line 81, Listing 6) if the transformation in one of the Miniphases is the identity, and if so, the transformation in that Miniphase is skipped.

Second, since most transformations do not change the type of the tree node, a fast path that explicitly checks for this case was added that avoids the dispatch in the `transform` method, and instead calls the node transformation method for the relevant node type directly.

4.1 Prepares

The Miniphase framework presented so far is sufficiently general to implement all but 4 Miniphases present in the Dotty compiler. The remaining 4 phases, however, perform transformations that depend on the ancestors of the current

```
private def chainMiniphases(first: Miniphase, second
    : Miniphase) = {
  new Miniphase {
    val valDefTransform =
      if (first.valDefTransform == id)
        second.valDefTransform
      else if (second.valDefTransform == id)
        first.valDefTransform
      else { x: ValDef =>
        val newX = phase.valDefTransform(x)
        newX match {
          case newX: ValDef =>
            second.valDefTransform(x)
          case other: Tree =>
            second.transform(other)
        }
    ...  // similar changes form all AST nodes
    }
}
```

Listing 6: Optimization for identity transforms and for nodes that keep the same node type

```
class MiniPhase extends Phase {
  ... //members introduced in previous listings
  val valDefPrepare: ValDef => Unit = empty
  val defDefPrepare: DefDef => Unit = empty
  val identPrepare: Ident => Unit = empty
  ...
  val selectPrepare: Select => Unit = empty
}
```

Listing 7: MiniPhase extended with prepares

tree node, so it may seem that a post-order traversal is not ideal.

One example is the LiftTry transformation which was described in Section 2.1. This transformation lifts **try** blocks within an expression into independent methods. When it encounters a try block, this phase needs to know whether the block is part of a larger expression, and thus it needs information about its ancestors in the tree.

In order to accommodate such phases without abandoning the consistent post-order traversal that enables phase fusion, `prepare` methods have been added to the framework that mutate the internal state of a phase when entering a given type of subtree. Specifically, the LiftTry phase maintains a boolean state which is an over-approximation of whether the current subtree is inside an expression that requires try blocks to be lifted into methods. Before processing a tree node using the `transform` method, the `runPhase` method first calls the corresponding `prepare` method to update the state of the Miniphase.

The `chainMiniPhases` method now also needs to chain `prepare`s, as shown in Listing 8.

In the current implementation, there is a separate prepare method for each type of tree node, just as there are node-

```
private def chainMiniPhases(first: MiniPhase, second
    : MiniPhase) = {
  new MiniPhase {
    val valDefTransform = ... // as before

    ... // as before

    val runsAfter: Set[MiniPhase] = ... // as before

    def checkPostCondition(t: Tree) = ... // as
     before

    val valDefPrepare =
      if (first.valDefPrepare == empty)
        second.valDefPrepare
      else { t: ValDef =>
        first.valDefPrepare(t)
        second.valDefPrepare(t)
      }
     ... // similar to valDefPrepare for all AST
     nodes
  }
}
```

Listing 8: Fusion with prepares

specific transform methods. Only very few phases have non-empty prepare methods, and those that do need to prepare for most kinds of tree node types. Therefore, it may have been sufficient (and simpler) to only have a single prepare method that is executed for every node regardless of its type.

4.2 Initialization and Finalization of Phases

Later, during development, we have found it helpful to extend Miniphases with the ability to `prepare` for a compilation unit and `transform` a compilation unit. `compilationUnitPrepare` is the proper place to initialize the initial internal state of the phase, such as populating global references used by the phase, while `compilationUnitTransorm` is a natural place to clean the internal state to avoid high memory footprint and memory leaks.

5. Evaluation

We have performed an empirical evaluation of the performance benefits of the Miniphase approach. We compared the current version of the Dotty compiler, which uses Miniphases, with a modified version in which the groups of Miniphases were split up, so that each Miniphase performed a separate tree traversal, like in the Megaphase approach. We ran both versions of the compiler on two significant input programs: the Scala standard library (34 000 LOC) and the Dotty compiler itself (50 000 LOC). In addition to the overall running time, we compared data from the JVM garbage collector, specifically the number of objects allocated and promoted to the old generation, and data collected using low-level CPU counters to explain cache behavior.

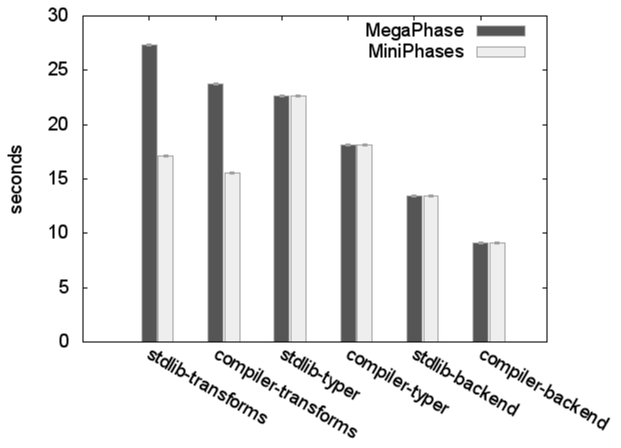


Figure 4: Execution time of tree transformation passes, typechecker, and code generation backend in Miniphase and Megaphase versions of the Dotty compiler.

The benchmarks were executed on a server with two Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz CPUs, running on a fixed frequency of 2.4Ghz with HyperThreading disabled. This CPU has a 25MB L3 cache. Every one of the 10 cores in this CPU additionally has a 256KB L2 cache and 32KB L1-icache and L1-dcache. In this architecture, the L2 cache is not inclusive and the L3 cache is inclusive on all levels above it: data contained in the core caches must also reside in the last level cache [6].

This server has 64Gb of 4-channel memory and runs 64-bit Ubuntu Linux with kernel version 4.4.0-45-generic. We have used the Oracle Hotspot Java VM version 1.8.0_111, build 25.111-b14. In order to ensure consistency between the runs and reduce variance due to disk seeks, all data needed for compilation is stored in `tmpfs`, a Linux filesystem that is an in-memory store.

5.1 Overall Time

Figure 4 shows the overall running time of the frontend, tree transformation pipeline, and backend. The tree transformations use a significant amount of the overall compilation time: in the Megaphase approach, they take more time than either the frontend or the backend. The graph also shows that Miniphases decrease the time taken by the tree transformations by 37% when compiling the standard library and 34% when compiling the Dotty compiler. Overall, the total compilation time (including the frontend and backend) decreases by 15% and 16%, respectively. In the following sections, we look in more detail at the likely reasons for this improvement.

5.2 GC Object Allocation and Promotion

In this section, we investigate the performance of the garbage collector. The reported values were obtained by parsing the GC logs obtained by passing `-XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to the Oracle Hotspot Java VM. The entire compiler pipeline was executed 50 times from a cold start, which represents a common setup of batch compilation in a big project.

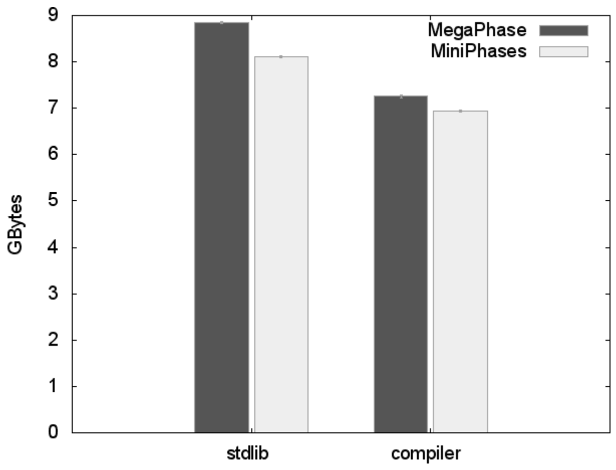


Figure 5: Total size of GC object allocated, GBytes

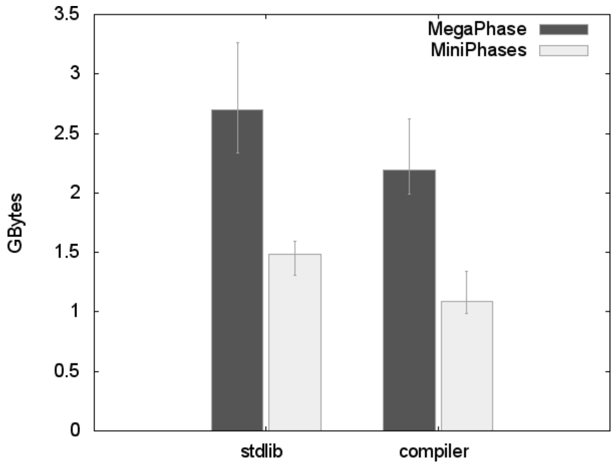


Figure 6: Total size of GC object tenured, GBytes

We measured how many managed objects are allocated and then promoted to the old generation by garbage collection. We performed our measurements during the compilation of the compiler itself and the standard library.

Figure 5 shows the total size of the objects allocated in the tree transformation pipeline. Miniphases reduce the amount of memory allocated by 5% during compilation of the Dotty compiler itself and 9% during compilation of the Scala standard library. This is explained by the fact that we need to recreate a path from the modified part of the tree to the root less frequently. It is important to note that the absolute amount of memory allocated is high, between 7 to 9 GB, so even a decrease of 9% is a lot of memory. Note that this is the total size of objects allocated during the entire execution of the compiler, not the total consumed amount of memory at any particular point in time.

The decrease in the objects promoted to the old generation is much more significant, even in a relative sense, as shown in Figure 6. The reduction thanks to Miniphases is a

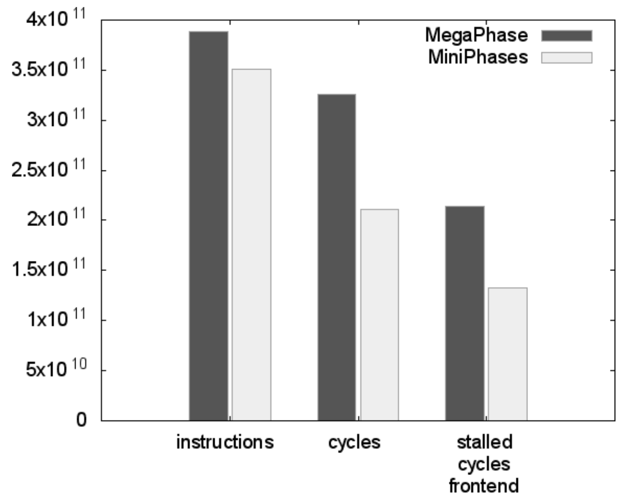


Figure 7: Instructions and cycle counters

full 49% and 55% for the standard library and Dotty compiler, respectively. In absolute terms, Miniphases reduce the promoted objects by over 1 GB in both cases. Many tree nodes that are created in a Miniphase are replaced by subsequent Miniphases in the same traversal, so they die young. In contrast, in the Megaphase approach, a node created in one phase is not replaced until the next traversal of the whole tree, and by that time, the node may already have been promoted to the old generation.

5.3 CPU Performance Counters

Focusing now on CPU behaviour, we used the `perf` utility that is shipped with Ubuntu Linux 16.04 with Linux kernel 4.4.0-45-generic to measure low-level CPU counters. This measurement approach is less intrusive than tracing or sampling profiling and allows to explain details of how the code was executed by the CPU.

To isolate the tree transformation pipeline from the front end and the code generator, we made two modified versions of the Dotty compiler: one stops execution after the front end, and the other stops execution after the tree transformations. The data collected during 50 executions of each of these versions was very consistent, with a variability less than 0.5% across runs. We subtracted the counts of the two versions to approximate the effect of the tree transformations on the performance counters.

Figure 7 shows the number of instructions executed, the number of clock cycles taken, and the number of stalled cycles during the execution of the tree transformations. The total number of instructions decreased by 10%, but the number of cycles used to execute those instructions decreased by a much larger 35%.

This is explained by Figure 8a, which shows that Miniphases decreased the cache miss rate by 47%, 17% and 40% for L1 cache loads, L1 cache stores and last level cache loads, respectively. Figure 8b indicates that the total number of cache accesses decreased by only 10%. Figure 8c shows that the total number of accesses that miss all on-chip caches and access main memory decreased by 47%, from 512 million to 278 million accesses.

Figure 8d presents the L1-instruction cache miss count, which decreased by 24%. We believe that this is explained by the fact that CPU caches are inclusive and eviction from last level cache would also trigger eviction from lower level caches. By improving the hit rate in data caches, Miniphases also indirectly reduce evictions from the L1-instruction cache.

We conclude that the main reason for the performance improvements of the Miniphase approach compared to the Megaphase approach is that the Miniphase approach makes more effective use of the CPU caches.

5.4 Comparison with Existing Production Compiler

To put the running times of the Dotty compiler with Miniphases in perspective, Figure 9 compares its performance to the existing Scala production compiler, `scalac`, which implements the Megaphase approach. It must be noted that they are different compilers, so confounding factors other than Miniphases also influence differences in their performance. Nevertheless, we observe that Dotty spends only 42% and 39% as much time in tree transformations as `scalac` when compiling the standard library and Dotty, respectively. Dotty's type checker is also faster than that of `scalac`, though this is unrelated to Miniphases, and the performance of the backends is about the same. Overall, Dotty compiles the standard library and itself in only 51% and 58% of the time taken by `scalac`.

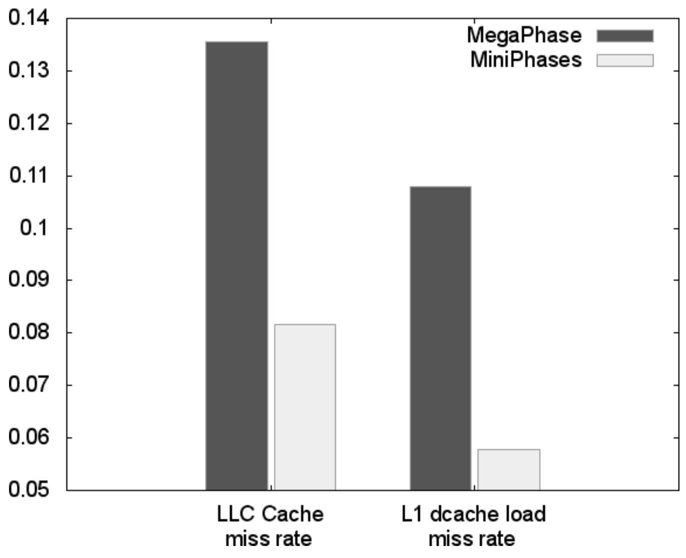
6. Soundness and Limitations of Phase Fusion

6.1 Fusion Criteria

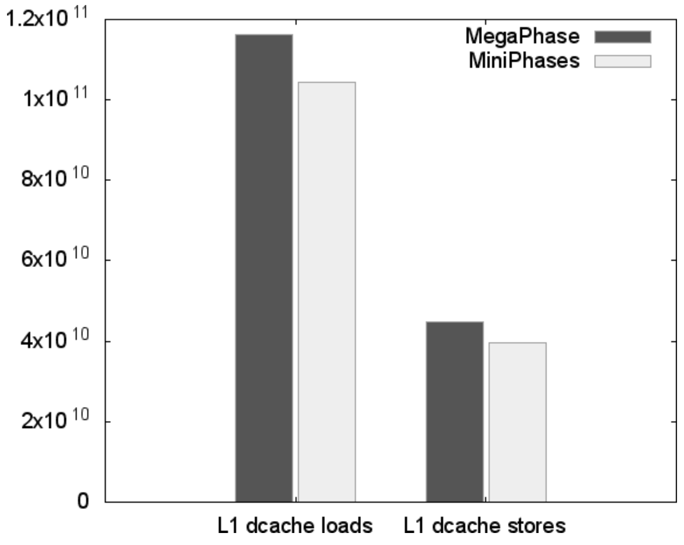
We do not formally define criteria that would give soundness guarantees that fusing phases does not change their behaviour. To be sound, any such formal criteria would have be conservative. They can give guarantees for simple programs in which tree traversals affect a small number of well-behaved data structures, but they would be too conservative to apply to the setting of a complex production compiler in which the tree traversals indirectly interact with files, tools external to the compiler itself and other kinds of global mutable state.

Instead, we provide high-level criteria that must be interpreted with an understanding of the overall design of the compiler and the high-level relationships between the major global data structures. The following requirements are sufficient for a Miniphase to be fusible into a block:

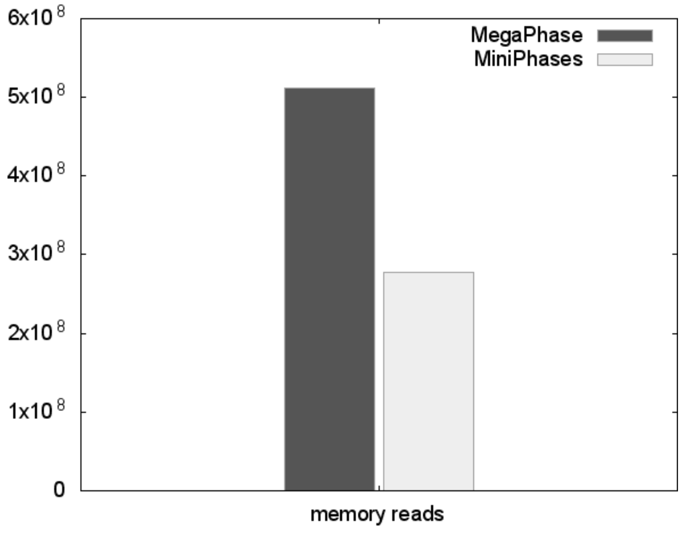
1. A phase does not break invariants registered by previous phases in the same block.



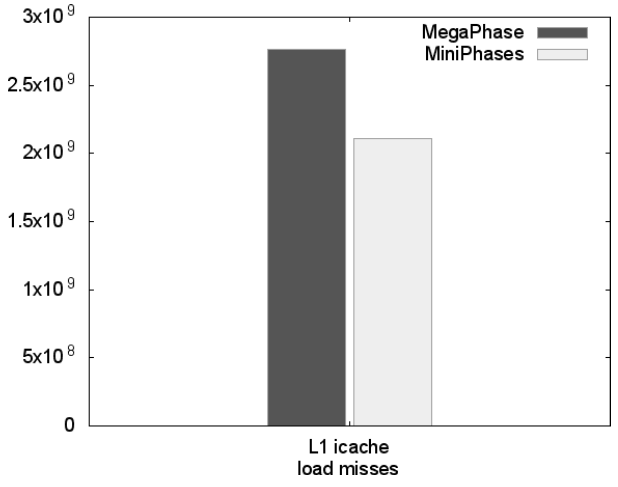
(a) Cache miss rates



(b) L1 cache access counters



(c) Memory accesses that missed cache



(d) L1-icache-load misses

Figure 8: Cache access counters.

2. A phase can successfully transform trees whose children have already been transformed by future phases in the same block.
3. A phase does not require that previous phases in the same block have finished transforming the entire compilation unit. Usually, when this is required, it is due to global data structures outside of the tree being transformed, such as the symbol table.

We have built a system for expressing phase invariants and postconditions that are enforced by dynamic checkers during testing. In our experience, these checkers are able to catch cases when these three requirements for phase fusion are violated. We will discuss these checkers in Section 6.3, but first, we examine examples of phases that are not fused because they violate the fusion criteria.

6.2 Example Violations of Fusion Criteria

Ideally, all the Miniphases in the compiler would be fused into a single traversal of the tree. In practice, our compiler has 6 separate blocks of Miniphases, marked with (*) in Table 2. Miniphases in the same block are fused together, but each block requires a separate traversal of the tree. Here, we describe some of the reasons that prevented us from fusing all Miniphases.

We have found that phases that violate of rule 1 are uncommon. While we did have phases that relax some invariants of previous phases, we were able to implement them in a more maintainable way following rule 1.

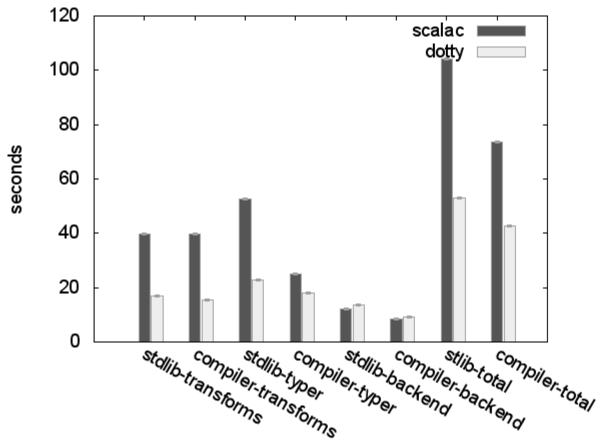


Figure 9: Execution time of stages of the Dotty and `scalac` compilers when compiling the standard library and Dotty.

6.2.1 Rule 2 Example: Pattern Matching

The Scala language has a very expressive pattern matching construct. A pattern matching phase translates this construct into complicated code with many branches and instructions similar to gotos. This phase also introduces a split between groups of Miniphases because it makes major changes to the structure of the trees, and because it would be difficult for other phases to handle both the high-level pattern matching constructs and the low-level control flow generated by this phase. One example of such a conflicting phase is tail recursion elimination, which transforms self-recursive methods with tail-calls into loops within the method (which do not grow the stack). Since both the pattern matching phase and the tail recursion elimination phase make non-local changes in the control flow, it would be very difficult to design them so that they can both execute in a single tree traversal. Following rule 2, pattern matching introduces a split between Miniphases in the phase-plan.

6.2.2 Rules 2 and 3 Example: Erasure

Since Java bytecode does not have generic types, a Scala compiler needs to erase type arguments from generic types. The phase that performs type erasure modifies the types of many trees. Since types are the main carriers of semantic information, it would be difficult to write other transformation phases that work on trees with both unerased and erased versions of types, violating rule 2.

At the same time, erasure has some global assumptions about trees that it sees. In particular it assumes absence of member selections on union types [19]. Union types are eliminated by the splitter phase, which is required to transform the entire compilation unit to eliminate all of them. Therefore, the type erasure phase introduces a split between groups of Miniphases because it violates both rules 2 and 3.

6.3 Phase Preconditions and Postconditions

Since the criteria from Section 6.1 are not verified statically, the Miniphase framework uses a system of dynamic assertions exercised by a large test suite to ensure correctness, and to localize a bug to a specific phase.

Each Miniphase defines postconditions that must hold about the tree nodes after the phase has transformed them. Runtime tests of the postconditions are implemented in the `checkPostcondition` method (Listing 4) of the Miniphase. The intended meaning of the postconditions is that if one Miniphase establishes a postcondition, all later Miniphases must also preserve it.

During testing, a checker pass is inserted between phases. A simplified version of its implementation is shown in Listing 9. The pass first checks various global invariants that are expected to always hold between any phase. For example, the checker removes all types from the tree and reconstructs them bottom-up, and checks that the reconstructed types are the same as the types that were associated with the tree. After checking global invariants, the checker pass runs the postcondition checks of not only the last executed Miniphase, but also of all the Miniphases that executed before it. This ensures not only that each Miniphase has established its postconditions, but also that no other Miniphases have invalidated them. In practice, we have found this mechanism to be very effective in the localizing bugs to a given Miniphase. In particular, bugs that involve interactions between different Miniphases would be difficult to track down without these checks. But if a postcondition of phase X fails after executing phase Y, we know immediately that phase Y breaks the invariant that phase X is intended to establish. For example, if a phase reintroduces a tree that contains pattern matching after the phase that eliminates pattern matching, we know immediately which phase to blame.

Miniphases also define preconditions by reference to the postconditions of other Miniphases. That is, a Miniphase specifies which other Miniphases must execute before it. For example, the phase that removes pattern matching requires the tail recursion elimination phase to finish processing all the trees before it can finish executing. Any preconditions specific to a Miniphase are usually the postconditions of some earlier Miniphase. To specify preconditions, a Miniphase defines two methods. The `runsAfter` method returns a set of Miniphases that must precede the current Miniphase. The `runsAfterGroupsOf` method returns a set of Miniphases that must strictly precede the fused Megaphase containing the current Miniphase. In other words, a Miniphase in `runsAfterGroupsOf` must completely finish transforming the tree before the current Miniphase can run. These two methods are used to specify the ordering criteria between Miniphases, in particular rule 2 from Section 6.1. If Miniphase X requires the postcondition of Miniphase Y to hold for only the node that X is immediately processing, X includes Y in `runsAfter`. If X requires the postcondi-

```
class TreeChecker(previousPhases: List[Phase], typer
     : Typer) extends Phase {
 def runPhase(t: Tree): Tree = {
   t.forAllSubtrees{subt =>
     val reTyped = typer.typeCheck(subt.stripTypes)

     reTyped.hasSameTypes(subt) &&
     checkNoDoubleDefinitions(subt) &&
     checkValidJVMNames(subt) &&
     checkcheckNoOrphanTypes(subt) &&
     /* other non-phase-specific sanity checks*/
     previousPhases.forAll { phase =>
       phase.checkPostCondition(subt)
     }
   }
 }
 ... // implementations of hellper methods such as
     checkNoDoubleDefinitions
}
```

Listing 9: Simplified version of TreeChecker

tion of Y to hold for all nodes of the tree, in particular for the children of the node that X is immediately processing, X includes Y in `runsAfterGroupsOf`. The phase ordering requirements specified by these two methods are checked when the Dotty compiler runs, not when it is compiled, but they are checked as soon as the compiler starts up, so any violations are caught immediately, independent of any test input.

The runtime overhead of the dynamic checks depends significantly on the specific code being compiled, but the approximate slowdown in the running time of the compiler is about 1.5x. The dynamic checks are enabled on every run of the test suite. The Dotty compiler has an extensive test suite that includes the tests from the test suite of the current production `scalac` compiler.

A similar dynamic invariant checking pass was initially implemented in the current production `scalac` compiler. However, in practice, it has not been maintained in a passing state: some Megaphases invalidate the postconditions of other Megaphases. For example, the pattern matching elimination phase creates references to symbols that are created only later, by a later phase. In general, because each Megaphase does multiple unrelated things, and because related transformations need to be split into different Megaphases, it has proven infeasible in practice to allocate to specific Megaphases the postconditions that should logically belong to the individual transformations.

7. Discussion

In this section, we discuss further experience with the Miniphase framework, including the onboarding process, code readability and maintenance, and common patterns that work well together with Miniphases.

7.1 Readability

The Scala and Dotty compilers are developed by several disconnected teams and open-source contributors. Most open-source contributors contribute their time voluntarily, and wish to start contributing quickly, without spending a lot of time to just get started. Most contributors want to solve the specific problem that bothers them. With the Miniphase framework, contributors find the phases easier to understand for two reasons:

First, each Miniphase is smaller and does a single transformation. A new developer needs to initially understand only one small phase, rather than a large Megaphase in which multiple different transformations are interleaved. This leads to less coupling and easier understanding.

Second, the Miniphase framework insists on a specific uniform structure of phases. While this makes it harder to write the initial implementation in this framework, it helps over the long term by making phases have similar structure and be easier to understand and maintain.

This is a very substantial improvement over the situation in the Scala 2.0-2.12 compiler, where fusing multiple complex phases together by hand made it very hard to keep track of what every phase does and how.

7.2 Predictable Performance Characteristics

The Miniphase approach imposes a specific structure that makes it easy for external contributors to join and reason about performance of a Miniphase. In most cases, the obvious solution that is suggested by the framework is the most efficient. This is very helpful in the presence of open-source contributors, since it reduces the number of iterations needed to polish the performance of contributed code.

7.3 Onboarding Process

Open-source contributors frequently ask how they can get involved and learn about internals of the compiler. A good way for new contributors to start working on the compiler is by extending either the tree checkers or phase postconditions. The new contributor learns which properties can be relied on in which phases, and can check her assumptions in test executions of the compiler. At the same time, the contributor improves the compiler with stronger checkers that make it possible to catch bugs earlier and simplify development and debugging. Moreover, the added postcondition checkers can serve as documentation of invariants for other new contributors.

7.4 Experience with Contributors

When a new phase is being developed, we need to decide where the phase should be run in the pipeline. Deciding whether two phases should be fused is a complex question that depends on how much high-level information the phase needs and whether it can co-exist in the same phase block. The former is commonly trivial while the latter is covered by the rules presented in Section 6.

Based on our experience, most people who contribute to the compiler are on the extremes: either they are experts who have been working on the compiler for long and know the entire pipeline, or they come to make a small contribution once in a while. While the first group doesn't need any guidance on knowing where to place a phase, the second group commonly starts by discussing the idea of a phase in a mailing list, online chat, or personal communication. In this discussion, experts suggest how the phase should be written and where it should be in the pipeline.

After an initial implementation is written, it is contributed as a pull request to a github repository and goes through review by experts maintaining the repository. At the same time, continuous integration systems run tests that verify that pre- and post-conditions hold for the entire testsuite, which includes the compiler itself, the standard library, and several thousands of programs contributed by the community.

8. Related Work

8.1 Deforestation and Stream Fusion

The original inspiration for the Miniphase approach was prior work on "deforestation" [3, 5, 24]. These approaches compose multiple functions that transform lists or trees without explicitly constructing the intermediate data structures between the composed functions. A limitation of these general approaches is that the functions to be composed must be in so-called treeless form. In the specific case of a Scala compiler, this condition is violated because the tree transformations inspect nodes nested inside subtrees and construct new subtrees consumed by subsequent phases. Thus, the general deforestation technique cannot be applied because it would change the semantics of the transformations.

8.2 Sound Fusion in Tree Traversal Languages

In this section, we describe several domain-specific tree traversal languages and frameworks that are more general than the functions that can be fused by deforestation, but still sufficiently restricted to enable static analysis of the patterns of data accesses in a traversal. This enables automatic sound reordering of the node visits in multiple traversals.

Attribute Grammar Scheduling Attribute grammars [12] are a formalism that defines computation on trees as evaluation of a set of pure functions for each node that may depend on the attribute values computed for other nodes. The formalism has been applied in many practical compiler implementations over the decades. As an example, JastAdd [4] is a recent attribute grammar framework that continues to be actively maintained, developed, and extended. A key problem is to find an order in which to evaluate the attributes of tree nodes that respects the dependencies between the attribute functions. For a particular parse tree, it suffices to topologically sort the pairs of tree nodes and their attributes, since the dependencies are explicit in the attribute evaluation functions. Various restricted classes of attribute grammars have been defined for which an evaluation order can be pre-computed ahead of time, independently of a particular parse tree. Some of these classes can be evaluated in a single pass over the parse tree, with a single visit of each node [10, 11, 15]. More general classes of attribute grammars require multiple passes, and algorithms have been proposed for finding evaluation orders that minimize the number of passes [1, 22]. These techniques have been extended to evaluation of attributes of multiple tree nodes in parallel [9]. Meyerovich et al. [16] combines parallel attribute scheduling techniques with programmer input in the form of sketches to synthesize GPU and multicore CPU implementations of tree manipulating programs.

Locality in Tree Traversals Techniques have been proposed to rewrite recursive programs that traverse trees to enhance data locality [7, 8, 25]. Jo and Kulkarni [7] proposed point blocking, a transformation similar to loop interchange, in which an outer loop of multiple tree traversals is interchanged with the traversal of the tree nodes, yielding a single traversal that executes the previously outer loop at each node that it visits. The transformation is applicable when the outer loop is parallelizable. Jo and Kulkarni [8] extended the idea of point blocking into a similar but more sophisticated technique, traversal splicing, that improves locality of irregular tree traversals that traverse only a subset of the nodes of the tree. Weijiang et al. [25] defined a static dependence test for a domain specific language for tree traversals. The dependence test analyzes tree access path expressions in the code that visits each tree node to determine which visits of which nodes can be reordered. The dependence test makes it possible to soundly apply point blocking, traversal splicing, and parallelization to a larger set of tree traversal algorithms.

MADNESS Passes Rajbhandari et al. [20, 21] propose and prove correct a technique that is able to compose recursive operators that are implemented using a set of primitive recursive operators. They demonstrate significant speedup obtained by fusion. Their approach is able to find an optimal schedule for fusion, while in our case the schedule is predefined. Compared to the dependence test of Weijiang et al. [25], the MADNESS system is more general in that it applies to both pre-order and post-order traversals.

The main benefit of the techniques described in this section is that they identify cases when soundness of fusion can be proven automatically. There are two reasons why they cannot be applied in the Dotty compiler. First, Dotty transformations modify the tree and construct new subtrees. Second, the implementations of Miniphase transformations are not purely functional: they manipulate non-local mutable data structures such as symbol tables, and they even cause additional files to be parsed and type-checked and transformed when they are referenced.

8.3 Other Pass Fusion Approaches

ASM [2] is Java bytecode instrumentation and emission library based on the visitor design pattern. A visitor transforms instructions in a sequence of bytecode instructions. ASM allows multiple visitors to be fused, so that part of the bytecode sequence is processed by all of them before continuing with the rest of the sequence. The obvious difference is that ASM transforms sequences, while Miniphases transform trees. For sequences, there is one obvious traversal order, while for trees, various traversal orders are possible. Miniphases impose a post-order traversal but provide the mechanism of prepares, discussed in Section 4.1, to implement transformations that would otherwise require different traversal orders. Another difference is that in Dotty, the meaning of a tree often depends significantly on its subtrees, so the issue of a phase observing children that have already been transformed by other trees is more important. In contrast, the meaning of a bytecode instruction usually does not depend on preceding instructions, at least not directly. Instead, it depends strongly on context, such as the state of the JVM operand stack, which ASM transformers usually maintain in additional data structures, not as part of the instructions themselves. In contrast, in the tree-based representation of Dotty, information about the operands of an expression node is associated with its child nodes. In general, both the input and the output of an ASM pass is JVM bytecode. In contrast, the purpose of the transformations in Dotty is to translate an intermediate representation similar to Scala source code to one similar to Java bytecode, so the types of nodes that appear in the tree gradually change as the tree passes through the sequence of transformations.

Lepper [14] proposes to optimize a sequence of traversals of trees by multiple visitors by detecting which visitors are interested in processing which nodes of the tree. This is done by using reflection to identify visitors that do not override the default visit methods for certain types of tree nodes. The optimized traversal can then skip traversing entire subtrees whose types ensure that none of the visitors are interested in visiting any of their nodes. A key difference is that these optimized visitors only traverse the tree, but do not generate different trees to pass from one visitor phase to the next.

8.4 Compilers Based on Tree Transformation Passes

The Nanopass Framework [23] is a compiler intended for teaching courses on compiler construction. In the framework, each individual transformation is done in a separate pass. Fusing the phases is suggested as possible future work. Due to practical considerations when compiling a complex language such as Scala, we need to have additional `prepare` passes, which the Nanopass Framework does not have.

Like Dotty, the Polyglot compiler [17] is structured as a sequence of passes that successively transform trees, in this case from various extensions of Java to Java itself. Like in Dotty, tree nodes are immutable, so each pass that replaces a tree node with a new one rebuilds the spine of the tree up to the root. The Miniphase approach of fusing tree transformations could also be used to improve the performance of Polyglot.

9. Conclusion and Future Work

The Miniphase approach removes the need to choose between modularity and efficiency in the implementation of tree transformations in a compiler. The resulting compiler is thus more modular and more efficient than using the Megaphase approach. This methodology simplifies both development and maintenance. Our evaluation indicates that using fused Miniphases allows speedups for tree transformations up to 1.6x that we demonstrated on real code bases with a real-world Scala compiler. Our detailed evaluation shows that the biggest contributing factor is improved cache friendliness, which leads to better CPU utilization.

Our approach is applicable not only to trees, but can be extended to directed acyclic graphs. We are also interested in using Miniphase-based approaches for executing independent compiler phases in parallel.

While our work was primarily focused on a compiler for Scala, we believe that the approach is general enough to be used in other compilers which share the same internal representation for considerable parts of their pipelines.

Acknowledgments

We want to thank Iulian Dragos for sharing his experience based on 12 years work on Scala compilers, starting before the time of Scala 2.0, even before the Scala compiler had bootstrapped itself. His knowledge was very helpful in understanding the evolution of the Scala 2.0-2.12 codebase.

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

We are grateful to other researchers and students that use and base their work on the Dotty compiler and to the anonymous reviewers for the valuable feedback and helpful comments.

References

[1] H. Alblas. Attribute evaluation methods. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 48–113, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-38490-8. doi: 10.1007/3-540-54572-7_3. URL `http://dx.doi.org/10.1007/3-540-54572-7_3`.

[2] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002.

[3] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming,*

ICFP 2007, Freiburg, Germany, October 1-3, 2007, pages 315–326. ACM, 2007. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291199. URL http://doi.acm.org/10.1145/1291151.1291199.

[4] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1):14–26, 2007.

[5] A. J. Gill. *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, UK, 1996. URL http://theses.gla.ac.uk/4817/.

[6] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2016. URL http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[7] Y. Jo and M. Kulkarni. Enhancing locality for recursive traversals of recursive structures. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 463–482, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048104. URL http://doi.acm.org/10.1145/2048066.2048104.

[8] Y. Jo and M. Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. In G. T. Leavens and M. B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 355–374. ACM, 2012. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384643. URL http://doi.acm.org/10.1145/2384616.2384643.

[9] M. Jourdan. A survey of parallel attribute evaluation methods. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 234–255, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-38490-8. doi: 10.1007/3-540-54572-7_9. URL http://dx.doi.org/10.1007/3-540-54572-7_9.

[10] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980. ISSN 1432-0525. doi: 10.1007/BF00288644. URL http://dx.doi.org/10.1007/BF00288644.

[11] U. Kastens. Implementation of visit-oriented attribute evaluators. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 114–139, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-38490-8. doi: 10.1007/3-540-54572-7_4. URL http://dx.doi.org/10.1007/3-540-54572-7_4.

[12] D. E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968. ISSN 1433-0490. doi: 10.1007/BF01692511. URL http://dx.doi.org/10.1007/BF01692511.

[13] G. Leontiev, E. Burmako, J. Zaugg, A. Moors, and P. Phillips. Sip-23 - literal-based singleton types. https://github.com/scala/scala/pull/4706, 2016. Accessed: 2016-10-24.

[14] M. Lepper and B. Trancón y Widemann. Optimization of visitor performance by reflection-based analysis. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations: 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pages 15–30, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-21732-6. doi: 10.1007/978-3-642-21732-6_2. URL http://dx.doi.org/10.1007/978-3-642-21732-6_2.

[15] P. Lewis, D. Rosenkrantz, and R. Stearns. Attributed translations. *Journal of Computer and System Sciences*, 9(3):279 – 307, 1974. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/S0022-0000(74)80045-0. URL http://www.sciencedirect.com/science/article/pii/S0022000074800450.

[16] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 187–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442535. URL http://doi.acm.org/10.1145/2442516.2442535.

[17] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In G. Hedin, editor, *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*, pages 138–152, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36579-2. doi: 10.1007/3-540-36579-6_11. URL http://dx.doi.org/10.1007/3-540-36579-6_11.

[18] M. Odersky. The Scala language specification v 2.11, 2016. URL https://web.archive.org/web/20160702192746/http://www.scala-lang.org/files/archive/spec/2.11/.

[19] B. C. Pierce. Programming with intersection types, union types. Technical report, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[20] S. Rajbhandari, J. Kim, S. Krishnamoorthy, L. Pouchet, F. Rastello, R. J. Harrison, and P. Sadayappan. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In J. West and C. M. Pancake, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 40:1–40:12. ACM, 2016. ISBN 978-1-4673-8815-3. URL http://dl.acm.org/citation.cfm?id=3014958.

[21] S. Rajbhandari, J. Kim, S. Krishnamoorthy, L.-N. Pouchet, F. Rastello, R. J. Harrison, and P. Sadayappan. On fusing recursive traversals of K-d trees. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 152–162. ACM, 2016.

[22] H. Riis Nielson. Computation sequences: A way to characterize classes of attribute grammars. *Acta Informatica*, 19(3): 255–268, 1983. ISSN 1432-0525. doi: 10.1007/BF00265558. URL http://dx.doi.org/10.1007/BF00265558.

[23] D. Sarkar, O. Waddell, and R. K. Dybvig. Educational pearl: A nanopass framework for compiler education. *J. Funct.*

Program., 15(5):653–667, 2005. URL `http://dx.doi.org/10.1017/S0956796805005605`.

[24] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[25] Y. Weijiang, S. Balakrishna, J. Liu, and M. Kulkarni. Tree dependence analysis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 314–325, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737972. URL `http://doi.acm.org/10.1145/2737924.2737972`.