Design and implementation of an optimizing type-centric compiler for a high-level language

THÈSE N^{O} 7979 (2017)

PRÉSENTÉE LE 11 DÉCEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Dmytro PETRASHKO

acceptée sur proposition du jury:

Prof. M. C. Gastpar, président du jury
Prof. M. Odersky, directeur de thèse
Prof. O. Lhoták, rapporteur
Dr C. Click, rapporteur
Prof. V. Kunčak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2017

A fact is the most stubborn thing in the world.
— Voland, The Master and Margarita,
Mikhail Bulgakov

To my parents, Igor and Tatyana, who have put me on this enjoyable path.

Acknowledgements

First of all, I want to thank my advisor, Martin Odersky, for letting me be a part of Scala team at EPFL. I was blessed with the opportunity to join you in developing a new compiler and it was a great experience. I have learned a lot from you, especially about design and soft skills. This was nothing short of awesome.

Second, I want to thank Ondřej Lhoták, who helped me in developing communication skills that allowed me to share my knowledge more accessibly. Thank you for asking all the questions which unveiled a lot of possible misinterpretations of what was said and written by me. Thank you for showing how to rephrase some ideas so that they become clearer and more accessible without loosing the mathematical rigor and precision of the original formulation. I also want to thank you for sharing your bright mood and inspiration. It was a pleasure to work with you and I hope we will stay in touch.

I would also like to thank Brian Goetz, especially for our discussions during the JVM Language Summit in 2016 and PLDI in 2017. Our discussions on design constraints of the Java programming language have broadened my views on both the long-term evolution of programming languages and the needs of industry. It was interesting to see how you design language features for Java under constraints which are different from the ones that apply to Scala.

I am thankful to my colleagues in the LAMP laboratory, whom I have had a chance to collaborate with over the years: Eugene Burmako, Denis Shabalin, Nicolas Stucki, Felix Mulder, Olivier Blanvillain, Vojin Jovanovich, Nada Amin, Sébastien Doeraene, Manohar Jonnalagedda, Sandro Stucki, Heather Miller and Samuel Grütter. I especially want to thank our secretaries, Danielle Chamberlain and Natascha Fontana, for taking care of the administrative matters relating to our research, and our system manager Fabien Salvi for a technical support.

Special thanks for my first experience in collaborating with the Scala team go to Alexandar Prokopec. ScalaBlitz, the project we developed together, has taught me how to design good libraries in Scala. It was also a great experience to beat Intel Thread Building blocks in benchmarks in the middle of the night at the EPFL and walk back together, as the metro wasn't working that late.

Acknowledgements

Over those four years I have supervised several exceptional semester and a Master's projects. I want to thank Guillaume Martres for multiple Dotty related projects we have done together. You are a remarkable student and I believe you'll do great in your Ph.D. at LAMP. Also, I want to thank Allan Renucci for his work on AutoCollections: you were the first to use the call graph construction algorithm in a novel use-case and the first to find the tricky bugs hiding within it. I want to thank Alexandre Sikiaridis for the preliminary exploration of specialization techniques in Dotty. TreeTypeMaps that have been developed for your project have become an important building block in Dotty. Also I want to thank Angel Axel, who contributed lazy collection operations to Scala Blitz, the first project of mine at LAMP. I have had a great experience working with Alfonso Petersen on Delaying arrays; although we haven't pushed this data structure out yet, I believe it has strong potential.

I appreciate the knowledge that I've learned from the previous generations of the Scala team at EPFL, namely Iulian Dragos, Michel Schinz, Lukas Rytz, and Adriaan Moors. You have shared a lot of insights based on your years of experience developing Scala; these insights saved me from repeating the mistakes of the past. You are the keepers of knowledge in the Scala world and you have always been willing to share this with those who search for it. Thank you.

I cannot write an acknowledgment without mentioning Yakir Michail, who taught me Math. I'm very thankful for you accepting me into your class and teaching me the culture and the artistry of Math. You have placed me and generations of students on the road to success in life by teaching us how to reason elegantly and efficiently. And thanks for sharing your library of the science magazine "Kvant". It is still my favorite magazine.

I want to thank my first programming teacher: Natalya Golubnycha. The mathematical & computer programming problems that you gave during school breaks got me interested in algorithms. While Michail's study of mathematics allowed me to arrive at elegant methods, you have taught me efficient methods in computer science. Unfortunately you had to leave the school after my first year, but the love of algorithms that you shared still lives inside me.

I would also like to thank Natalya Bogomolova for providing me the right material to maintain this love of algorithms and for your limitless kindness. Your classroom was the nicest place I have ever been and I cherish the memories of all the nice tea parties that we had there.

Thanks also go to Kharitonova Ludmila, who showed me how to apply Math not only in theoretical, but in practical areas that handle real properties of existing objects; chemistry in particular. You taught me how to approximate complex Math quickly and this skill is becoming more and more useful in my life.

I thank Alexander Gasnikov, who taught me the beautiful science of probability theory, mathematical statistics and stochastic processes, with further application to traffic jams. The seminar and laboratory organized by you was always a place where I could discover fascinating facts in all areas of applied mathematics. It was a pleasure collaborating with you in making Moscow a better place to drive for everybody.

Vera Petrova helped me both to appreciate the abstract part of science and showed me how to teach it. Your wonderful lists of problems are a unique teaching tool and they show your refined taste in math. Thank you for all the time you took to look through my solutions to all 368 of them, and for your time explaining how they could be solved more elegantly.

I thank Sergei Belyaev for showing me the connection between art and math and showing me that Math can also be emotional. Your drawing of elephants in every detail was a surprising way to explain how mathematical problems should be solved. And also thank you for making me love compass-and-straightedge construction.

I want to thank Philip Adronov, with whom I've shared the first experience of working with big data stores. Our joint work on building indexes for Cassandra let me appreciate the needs of industry. This was instrumental in helping me understand the purpose tools, and in particular compilers, serve and helped me to make my research more influential.

I would like to thank Elena Bunina and Maxim Babenko for organizing the Yandex School of Data analysis, where I studied I/O efficient algorithms and machine learning. It allowed to me both to deepen my knowledge in general of algorithms and to broaden it to compression, machine learning, hashing and information retrieval at scale. This school creates a wonderful unifying environment for people passionate about algorithms and data analysis and I enjoyed my time there.

I want to thank the broad Scala community for building a great ecosystem with lots of useful libraries. Though we don't always agree on the direction we should move towards, we all act to build a bright future for Scala. Through trials and errors we all are developing new methods to build software that works well and is pleasant to deal with. Among the Scala community I want to especially thank Jon Pretty for being a wonderful organizer of events, a great arbiter of conflicts , and a good and reliable friend.

When it comes to personal friends, I want to thank Mark Bochkov. I have ways had great adventures with you, either in the real world or in digital worlds. I hope we will continue our friendship and share many more quests together.

I would also like to thank Alexander Zemtsov for the discussions about professionalism in music that we had. In those discussions I have found strong similarities between computer science and music and it helped me to overcome my laziness.

Acknowledgements

I want to thank Yulia for her endless care and understanding — for helping me to endure hard times of my Ph.D., and for being with me at moments of joy. I am glad that I have met you and I'm looking forward to living long years together, my dear wife.

And finally, I would like to heartily thank my parents Igor and Tanya. You were always making sure that I was doing great. Having you as role models in my life served me well and made me who I am today.

Lausanne, 19 July 2017 D. P.

Abstract

Production compilers for programming languages face multiple requirements. They should be correct, as we rely on them to produce code. They should be fast, in order to provide a good developer experience. They should also be easy to maintain and evolve.

This thesis shows how an expressive high level type system can be used to simplify the development of a compiler. We demonstrate the system on a compiler for Scala.

First, we show how expressive types of high level languages can be used to build internal data structures that provide a statically checked API, ensuring that important properties hold at compile time.

Second, we also show how high level language features can be used to abstract the components of a compiler. We demonstrate this by introducing a type-safe layer on top of the bytecode emission phase. This makes it possible to abstract away the implementation details of the compiler frontend and run the same bytecode emission phase in two different Scala compilers.

Third, we present "MiniPhases", a novel way to organize transformation passes in a compiler. MiniPhases impose constraints on the organization of passes that are beneficial for maintainability, performance, and testability. We include a detailed performance evaluation of MiniPhases which indicates that their speedup is due to improved cache friendliness and to a lower rate of promotions of objects into the old generations of garbage collectors.

Finally, we demonstrate how the expressive type system of the language being compiled can be used for static analysis. We present a novel call graph construction algorithm which uses the typing context for context sensitivity. The resulting algorithm is both substantially faster and more precise than existing alternatives. We demonstrate the applicability of this analysis by extending common subexpression elimination to idempotent expression elimination.

Key words: compiler design, optimizing compiler, compiler performance, tree traversal fusion, cache locality, call graphs, parametric polymorphism, static analysis, Scala

Résumé

Pour pouvoir être utilisé en production, le compilateur d'un langage de programmation doit répondre à de multiples critères. Il doit être correct car le développeur en dépend pour générer du code. Il doit être rapide afin de fournir une bonne expérience utilisateur. Il doit être facile à maintenir et à faire évoluer.

Cette thèse montre comment un système de typage de haut niveau peut être utilisé pour simplifier le développement d'un compilateur. Le principe présenté est illustré dans un compilateur pour Scala.

Nous commençons par montrer comment, à l'aide des types expressifs d'un langage de haut niveau, nous pouvons construire des structures de données internes qui fournissent une interface de programmation (API) vérifiée statiquement, garantissant à la compilation que certaines propriétés importantes sont vérifiées.

Ensuite, nous montrons comment des fonctionnalités d'un langage de haut niveau peuvent être utilisées pour abstraire les composants d'un compilateur. Nous démontrons ceci en introduisant une couche d'abstraction à typage sûr par-dessus la phase de génération de *bytecode*. Ceci permet d'abstraire les détails d'implémentation de la partie avant du compilateur (*frontend*) et d'utiliser la même phase de génération de *bytecode* pour deux compilateurs Scala différents.

Troisièmement, nous présentons les *MiniPhases*, une manière nouvelle d'organiser les passes de transformations d'un compilateur. Les MiniPhases imposent des contraintes sur l'organisation des passes, qui sont bénéfiques à la fois pour la maintenance et les performances du compilateur, ainsi que pour sa capacité à être testé. Notre évaluation détaillée montre que les bonnes performances des MiniPhases sont dues d'une part à une utilisation plus intelligente du cache, et d'autre part à un taux inférieur d'objets promus dans les vieilles générations du ramasse-miettes (*garbage collector*).

Enfin, nous démontrons comment, lors de la compilation d'un langage de haut niveau, son système de typage peut être utilisé pour effectuer de l'analyse statique. Nous présentons un nouvel algorithme de construction de graphe d'appels qui utilise le contexte de typage pour être sensible au contexte. Cet algorithme est à la fois plus rapide et plus précis que les alterna-

tives existantes. Nous montrons, en example de l'intérêt pratique de cette analyse, comment l'utiliser pour étendre l'élimination de sous-expressions communes à l'élimination d'expressions idempotentes.

Mots clefs : conception de compilateur, compilateur optimisant, performances d'un compilateur, fusion de parcours d'arbres, graphes d'appels, polymorphisme paramétrique, analyse statique, Scala

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

There is no greatness where there
is no simplicity, goodness and truth.

— Leo Tolstoy

Compilers for a real world programming language face multiple requirements:

- Big compilers are developed by big groups of people. For example, the current Scala compiler has over 300 contributors, and among them over 40 have contributed more than 10,000 lines of code each. Collaboration on such a big codebase requires clear structure and a clear separation of concerns in order to be maintainable.
- At the same time, a compiler is frequently invoked by users during their every-day development. Every key stroke in the IDE fires up a compiler to parse, typecheck and validate the correctness of the current state of the code in the IDE. It is vital for the developer's productivity to keep response times after every key stroke short. This requires a compiler to be fast measured both by throughput, as the file being edited can be big, and by latency, to also respond quickly for small files.

It is commonly thought that the requirements indicated in the preceding two paragraphs — modularity and performance — are mutually exclusive. Modularity comes in the form of abstractions and abstractions have an inherent cost.

At the same time, an often-considered way to get performance is to side-step abstractions and use inventive ways to work around existing infrastructure to speed up the application. Thus it is usually thought that fast code is unnatural in a modular system as it may be inconsistent with modularity.

This is a hard choice to make. Choosing maintainability and modularity over performance is likely to make the compiler slow and its users unhappy due to long compilation times. You'll have a nicely organized compiler that is rarely used and thus is not well tested.

Choosing performance at the expense of maintainability introduces a huge burden on future development of the compiler. Developing new features as well as fixing bugs becomes hard in

a code-base that uses ad-hoc escape hatches to side-step internal APIs which are considered slow. At the same time, this provides the best user experience in the early days of the compiler, as users get good performance right away, at the cost of future work for compiler developers.

This thesis shows that this is not a mutually exclusive choice — with a careful architecture one can get both maintainability and good performance by design using:

- an expressive type system to guide the implementation of code in the compiler towards correctness. This type system will make the code inside the compiler more uniform and natural;
- high-level abstractions that are friendly both to contemporary CPUs with multi level caches and to developers, by providing a convenient API that promotes a natural notion of modularity in this system.

This work has been performed in the context of Scala, a functional object oriented language with multiple trait inheritance that compiles into Java bytecode and runs on the Java Virtual Machine. While there are proposals to add static dispatch to Scala [Petrashko et al., 2011], as of this writing all calls to non-private methods in Scala are virtual.

1.1 Contributions

The Work performed in this thesis was targeted at improving techniques used to build compilers and was demonstrated on a Scala compiler. It shows that expressive type systems and high level abstractions of the host language can be used to build compilers which are maintainable, modular, and fast.

This thesis claims to make the following contributions:

- a case study that shows that usage of expressive types of high level languages can be used to build compiler components for a real-world programming language that is easy to maintain and develop;
- MiniPhases, a practical way to organize phases in a production compiler that allows the building of both a pipeline that is both maintainable and performant. This methodology allows a compiler writer to define multiple transformations separately, but fuse them into a single traversal of the intermediate representation when the compiler runs. The evaluation shows that the proposed scheme behaves faster than expected and explains that this performance is due to better cache locality;
- a callgraph analysis using the precision of the underlying type system of the language, that is both more precise and faster than existing alternatives. The idea is applicable to languages with parametric polymorphism. This analysis is context sensitive and

uses the typing environment as context. The use of static types from the caller as context is effective because it allows more precise dispatch of call sites inside the callee. The context-sensitive analysis runs two times faster than a context-insensitive one and discovers 20% more monomorphic call sites at the same time. This analysis has been designed with the intention to include it in the mainline compiler to power whole-program optimizations.

These contributions have been validated by implementing an experimental compiler for Scala called "Dotty". This experiment has been proven successful. Practical evidence suggests that developing new language features is considerably simpler in this compiler as most new Scala features are first implemented in Dotty. A compiler based on Dotty is slated to become the main compiler for the release of Scala 3.0.

1.2 Overview

This thesis is organized in the following way:

- Chapter 2 describes the high level organization of the Dotty compiler as well as data structures used to represent the information necessary for program compilation.
- Chapter 3 demonstrates how high level abstractions can be used to separate components of the compiler by presenting the abstractions used by BackendInterface in Dotty.
- Chapter 4 contains a detailed presentation and an evaluation of MiniPhases, a technique that was used to build a maintainable and fast compiler.
- Chapter 5 presents a call-graph construction algorithm that is both more precise and faster than existing alternatives, making it practical for inclusion in a production compiler, and demonstrates its use to perform class and method specialization.
- Chapter 6 provides an example of an application of the call-graph analysis to perform global idempotence inference which permits the extension of common subexpression elimination to more complex expressions.
- Chapter 7 covers local optimizations that we use to speed up the compilation and generate better code, even in the absence of whole-world analysis.

2 Structure of the Compiler

It is common for compilers to use multiple different representations to store the program currently being compiled. Scala compilers are distinct in that they use the same data structures during the entire compilation. In this section we will describe the motivation behind core entities in the Dotty compiler and demonstrate how they work together. Dotty compiler is compiling Scala programs and is itself is written in Scala.

Special attention is given to API decisions that had an impact on the coding style used inside the Dotty compiler. These decisions made a substantial improvement in either type safety or efficiency.

Attribution

The work presented in this chapter has been performed by Martin Odersky and is included to serve as a background for other chapters of this thesis. While author of this thesis was the first to use this API, provide the feedback and helped to fix bugs, the authorship of ideas, implementation and terminology presented in this chapter is attributed to Martin Odersky. The author of this thesis only claims the authorship of benchmarks and measurements presented in Section 2.11.

2.1 Names

In Scala, a term and a type may share the same name:

```
1 trait A {
2   val member = 0
3   type member
4 }
```

Listing 2.1 – Types and terms can share the same name

While both the term `member` and the type `member` are named the same, they behave differently,

```
5  abstract class Name  { self =>
6
7   /** A type for names of the same kind as this name */
8    type ThisName <: Name { type ThisName = self.ThisName }
9   /** Is this name a type name? */
10   def isTypeName: Boolean
11
12   /** Is this name a term name? */
13   def isTermName: Boolean
14
15   def ++ (other: Name): ThisName = ...
16   ...
17 }
18
19 class TermName(chrs: Array[Char], start: Int, length: Int) extends Name {
20   type ThisName = TermName
21   def isTypeName: Boolean = false
22   def isTermName: Boolean = true
23   ...
24 }
25
26 class TypeName(val toTermName: TermName) extends Name {
27   type ThisName = TypeName
28
29   def isTypeName = true
30   def isTermName = false
31   ...
32 }
```

Listing 2.2 – Term and type names

one being a type, the other — a term.

In order to disambiguate names of terms and names of types, Dotty uses classes to represent different kinds of names (see 2.2).

As such, even though they *textually* have the same name, they are semantically distinguishable.

2.1.1 Tracking kinds of names

It is very common to transform names while keeping their kinds; that is why we have introduced a type-safe way to do it. We use a type member `ThisName` to be able to define a method that is statically known to return the same kind of name as the receiver of the call. This substantially improves type-safety of names and reduces the number of casts needed in the code base. This code pattern is quite common inside the Dotty codebase.

2.1.2 Names are cached

Common names of identifiers such as `i` and `apply` are used in multiple scopes. Storing these names multiple times would be wasteful, even if they do not refer to the same variable, field or type. In order to lower the memory footprint, we reuse the same underlying character array `chrs` in the entire compiler.

This is also beneficial as name comparison is a very frequent operation. In order to speed it up, we also introduce an additional guarantee that two names are equal if, and only if, they are referentially equal.

This allows us to reduce memory usage and optimize name comparisons at the cost of a more complex procedure when allocating new names. Allocation of a new name now requires determining if the same name has ever been allocated, which may require comparison with all names allocated before. In order to reduce the number of comparisons, we hash all allocated names and form linked lists of names that have the same hash:

```
33  /** Hashtable for finding term names quickly. */
34  private var table = new Array[SimpleName](InitialHashSize)
35
36  /** The number of defined names. */
37  private var size = 1
38
39  /** Create a term name from the characters in cs[offset..offset+len-1].
40   *  Assume they are already encoded.
41   */
42  def termName(cs: Array[Char], offset: Int, len: Int): SimpleName = synchronized {
43    util.Stats.record("termName")
44    val h = hashValue(cs, offset, len) & (table.size - 1)
45
46    /** Make sure the capacity of the character array is at least `n` */
47    def ensureCapacity(n: Int) =
48      if (n > chrs.length) {
49        val newchrs = new Array[Char](chrs.length * 2)
50        chrs.copyToArray(newchrs)
51        chrs = newchrs
52      }
53
54    /** Enter characters into chrs array. */
55    def enterChars(): Unit = {
56      ensureCapacity(nc + len)
57      var i = 0
58      while (i < len) {
59        chrs(nc + i) = cs(offset + i)
60        i += 1
61      }
62      nc += len
63    }
```

```
64
65      /** Rehash chain of names */
66      def rehash(name: SimpleName): Unit =
67        if (name != null) {
68          val oldNext = name.next
69          val h = hashValue(chrs, name.start, name.length) & (table.size - 1)
70          name.next = table(h)
71          table(h) = name
72          rehash(oldNext)
73        }
74
75      /** Make sure the hash table is large enough for the given load factor */
76      def incTableSize() = {
77        size += 1
78        if (size.toDouble / table.size > fillFactor) {
79          val oldTable = table
80          table = new Array[SimpleName](table.size * 2)
81          for (i <- 0 until oldTable.size) rehash(oldTable(i))
82        }
83      }
84
85      val next = table(h)
86      var name = next
87      while (name ne null) {
88        if (name.length == len && equals(name.start, cs, offset, len))
89          return name
90        name = name.next
91      }
92      name = new SimpleName(nc, len, next)
93      enterChars()
94      table(h) = name
95      incTableSize()
96      name
97    }
```

Listing 2.3 – Caching of names

This strategy is similar to the string interning performed by Java virtual machines, although we use our own tables. This allows us to side-step efficiency problems related to active use of interned strings [Shipilev, 2011].

2.2 Signatures

In Scala, multiple methods in the same class are allowed to have identical names, as long as they have different signatures:

```
98  class Foo {
99    def foo(a: Int): Int
100   def foo(a: Short): Int
101 }
```

Listing 2.4 – Method overloading example

In this example, both `foo` methods may exist at the same time, as we can distinguish them by their *signature*. A signature is a string that represents erased classes of arguments and return type. There is a terminology clash with the JVM Specification [Lindholm and Yellin, 1999], as our signatures are called "Method Descriptors" in the JVM specification, while what is called a "signature" in the JVM Specification is referred to simply as "type" in our compiler.

```
102 case class Signature(paramsSig: List[TypeName], resSig: TypeName) {
103   ...
104 }
```

Listing 2.5 – Signatures in Dotty

There are two important details in how the Dotty compiler deals with signatures:

- Signatures are created by the Typer, a phase that does type inference and typechecking, during early overload resolution. This is an interesting situation, as this requires being able to erase types when we have not finished typing the compilation unit. DenotTransformers (2.10) are instrumental in making this possible.
- Signatures are used to keep overload resolution stable until Erasure. This introduces a limitation on what DenotTransformers (2.10) can do before erasure: changing the signature of a denotation will break all the links to it from TermRefs. After erasure, all links are symbolic and such changes are fine.

2.3 Trees

Trees, or, more formally, abstract syntax trees, represent the application currently being compiled. Trees are first created by Parser, which processes the code written by the user.

Later, these trees are typed by Typer, which attributes every tree with types that it infers and removes syntactic sugar. These trees are later transformed by the compiler.

2.3.1 Trees are immutable

In order to simplify the API of the compiler, trees are immutable. This means that they do not contain links to parent tree nodes, as otherwise there will have to be mutated to set it. This allows trees to be reused in multiple places. In particular, all references to the same entity can be potentially represented using the very same object. [1]

[1] ↑This is only true for synthetic trees however, as non-synthetic trees have to contain source positions that are used by IDEs

```
105  /** Checks whether predicate 'p' is true for all result parts of this expression,
106   *  where we zoom into Ifs, Matches, and Blocks.
107   */
108  def forallResults(tree: Tree, p: Tree => Boolean): Boolean = tree match {
109    case If(_, thenp, elsep) => forallResults(thenp, p) && forallResults(elsep, p)
110    case Match(_, cases) => cases forall (c => forallResults(c.body, p))
111    case Block(_, expr) => forallResults(expr, p)
112    case _ => p(tree)
113  }
```

Listing 2.6 – Utility function that works both for typed and untyped trees

2.3.2 Type-safe usage of typed and untyped trees

As Dotty uses the same trees both after Parser and after Typer, trees can exist in both typed and untyped variants.

Sometimes it is useful to distinguish between typed and untyped trees. For example:

- we want to ensure that untyped trees can never be contained inside typed trees;
- we want to allow typed trees to be contained inside untyped ones;
- we want to be able to write utility methods, such as shown in Listing 2.6, that operate on both typed and untyped trees. It would be wasteful to implement them twice.

In Dotty , we use Scala's expressive type system to use the same runtime data structure to represent both typed and untyped trees, while relying on the compile-time type system to guarantee that untyped trees do not escape to places where only typed trees are expected.

This is achieved by having a generic class `Tree` (see Listing 2.7) that takes a type-argument. Two instantiations of this class are provided, with type arguments `Untyped` and `Type` respectively.

Using this technique, it is possible to indicate if a method is able to work only on typed trees, or on both. More importantly, it is possible to write a single method that, given a typed tree, will return a typed tree, and given an untyped tree will return an untyped tree, as shown in the method `findSubTree` presented in Listing 2.8.

2.3.3 Type-safe tracking of the kind of a typed tree

It is also quite common that a utility method should return the same kind of AST node that it was given. Consider a method `withType`(Listing 2.9) that assigns a type to a tree node: it will return the same kind of node, but the new node will be known to be typed. This is the same idiom as the one presented in Section 2.1.1.

```
object Trees{
  abstract class Tree[-T >: Untyped] {
    def tpe: Type
    ...
  }

  case class Ident[-T >: Untyped](name: Name) extends RefTree[T]

  abstract class Instance[T >: Untyped <: Type] {
    type Tree = Trees.Tree[T]
    type Ident = Trees.Ident[T]
    type Select = Trees.Select[T]
    type ValDef = Trees.ValDef[T]
    ...
  }
}

object tpd extends Trees.Instance[Type] {
  ...
}

object untpd extends Trees.Instance[Untyped] {
  ...
}
```

Listing 2.7 – Trees

```
def isPureExpr(tree: tpd.Tree): Boolean = ...

def findSubTree[T >: Untyped](pred: Tree[T] => Boolean)(inTree: Tree[T]): Tree[T]= ...
```

Listing 2.8 – Abstracting over the typedness of a tree in methods

```
object Trees {
  abstract class Tree[-T >: Untyped] {
    def withType(tpe: Type)(implicit ctx: Context): ThisTree[Type] = {
      val tree =
        if (myTpe == null || (myTpe eq tpe)) this
        else clone
     tree.asInstanceOf[Tree[Type]].overwriteType(tpe)
     tree.asInstanceOf[ThisTree[Type]]
    }

    type ThisTree[T >: Untyped] <: Tree[T]
    ...
  }
  case class Ident[-T >: Untyped](name: Name) extends RefTree[T]{
    type ThisTree[-T >: Untyped] = Ident[T]
  }
}
```

Listing 2.9 – Illustation on generic tracking of the kind of a tree

2.3.4 Tree copiers

While trees are allocated very frequently inside the compiler, most transformations that are performed on trees will, in practice, return the same tree unchanged. In order to reduce the number of trees allocated during transformation, we developed `TreeCopiers` which checks if the previous version of the tree can be used instead of allocating a new one.

The previous version of the tree is used in case:

- the updated tree has all the subtrees and attributes unchanged, or
- the type of the tree is assigned for the first time.

Systematic use of `TreeCopiers` allows the reuse of entire subtrees, reducing pressure on the allocator and garbage collector and improving memory locality by reducing the size of the working set.

2.4 Types

Types represent the semantic meaning of a tree. Here are several examples of the surface syntax for types in Dotty:

```
158 val a: Int = ...                    // a has type Int
159 val b: a.type = ...                 // b has type which inciates that b stores value a
160 val c: Int | Double = ...           // c is either an Int or a Double
161 val d: Serializable & Product = ... // d is both a Serializable and a Product
162 val e: List[Int] = ...               // e has a type List{T} & {T = Int}
163 val f: Int @unchecked = ...         // f is an annotated type
164 def g: Int = ...                    // g is an expression
165 def h(): Int = ...                  // h is a parameterless method
166 def k[T](): Int                     // k is a poly-method that returns a method
167
168 type A = [B] =>  (B, B)             // type A has type type lambda
169 type C >: Int <: Any                // type C has type typebounds
170 type D = Int                        // type D has type typebounds where both lower and
171                                     // upper bound are the same
```

Listing 2.10 – Surface syntax for types in Dotty

At the same time, there are some types that developers will never encounter but which are still needed for correct compilation of Dotty sources, for example *MethodType* and *LazyType*.

2.4.1 Classification of types

In order to keep track of such a big variety of types, we have introduced several different dimensions used to classify them.

```
176 /** A marker trait for type proxies.
177  *  Each implementation is expected to redefine the `underlying` method.
178  */
179 abstract class TypeProxy extends Type {
180
181   /** The type to which this proxy forwards operations. */
182   def underlying(implicit ctx: Context): Type
183
184 }
185
186 case class AnnotatedType(tpe: Type, annot: Annotation)
187       extends UncachedProxyType with ValueType {...}
188
189 abstract case class RefinedType(parent: Type, refinedName: Name, refinedInfo: Type)
190 extends CachedProxyType with ValueType {...}
```

Listing 2.12 – Proxy types

TypeTypes and TermTypes

First of all, we introduce a distinction between *TypeType*s and *TermType*s.

TypeTypes can only apply to definitions of types defined in the program: classes, traits, type members and type arguments.

```
172 class Example[A] {
173   type B = Int
174   def foo[C] = 1
175 }
```

Listing 2.11 – A, B, C and Example will have types that are TypeTypes

TermTypes apply to terms: variables, methods and fields. They are by far the most common types.

Proxy Types and Ground Types

We introduce a distinction between *ground types*, which are proper new types, and proxy types, which somehow add information to already existing types. Examples of Proxy types include AnnotatedType and RefinedType, see Listing 2.12.

An AnnotatedType indicates that the already existing type has been annotated, such as `val f: Int @unchecked`. Here, the underlying type would be `Int`.

A refined type is used to refine a value of the already existing type member. The straightforward way would be to refine a member directly, as in the following example:

```
trait A {
 type T
}
val d: A { type T = Int }
```

Listing 2.13 – Refined types example

Dotty also uses refinement types to implement types of higher kind. For example: `List[Int]` would be encoded as `List {type List$T = Int }`. For more details about this encoding see [Odersky et al., 2016].

Cached Types and Uncached types

It would be wasteful if we had a new type allocated for each user-defined variable of type `Int`. Instead, we cache a lot of types. This not only improves memory consumption, but additionally allows us to speed up sub-typing checking through the usage of reference quality.

Caching is done through per-compilation hashmaps. Types are hashed and grouped by the hashcode. A special hash code value is used to indicate that a type has a component that is not hashed. Hashes are computed lazily and memoized; therefore an additional value is needed to indicate that the hash has not been computed yet.

```
object Hashable {

  /** A hash value indicating that the underlying type is not
   *  cached in uniques.
   */
  final val NotCached = 0

  /** An alternative value returned from `hash` if the
   *  computed hashCode would be `NotCached`.
   */
  private[core] final val NotCachedAlt = Int.MinValue

  /** A value that indicates that the hash code is unknown
   */
  private[core] final val HashUnknown = 1234

  /** An alternative value if computeHash would otherwise yield HashUnknown
   */
  private[core] final val HashUnknownAlt = 4321
}
```

Listing 2.14 – Special values of hashes

Due to the fact that there are several special values in the caching scheme, we should be very careful to ensure that when mixing hashes from components of a type, we do not inadvertently generate a special value. That is why, if a type hashes into a special value, we will put it into a pre-defined alternative bucket. This means that two buckets — that would otherwise be

hashed to special values — are empty, and their entries are moved to alternative buckets. Those alternative buckets will have, on average, twice as many elements as other buckets.

```
215 private def avoidSpecialHashes(h: Int) =
216     if (h == NotCached) NotCachedAlt
217     else if (h == HashUnknown) HashUnknownAlt
218     else h
```

Listing 2.15 – Avoiding special hashes

SingletonTypes

SingletonTypes are known to contain only a single non-null inhabitant. Though they aren't very common in the Scala language itself, they are very common inside compiler data structures, as *TermRef*s, *ThisType*s and *SuperType*s are all singleton types. *TermRefs* represents the majority of types allocated in Dotty (see Section 2.11.2).

NamedTypes: TermRefs and TypeRefs

Named types are the core abstraction in Dotty and are closely linked to Denotations, which will be described later. They represent a reference to a named selection from a prefix. The prefix is also represented by a type. A special prefix *NoPrefix* is used to indicate that a selection is taken from a local scope.

ValueTypes

ValueTypes are types that can be the types of values. For example, a value can have a type *Int*, but it can not have a type that is a MethodType.

ProtoTypes

Prototypes are not user-facing and describe an expected type that is used in Typer. A good illustration would be *SelectionProto*, which indicates that the expression being typed is in a location where we expect this tree to have a member with the name `name` whose type matches `memberProto`.

```
219 abstract case class SelectionProto(name: Name, memberProto: Type, compat:
220 Compatibility, privateOK: Boolean)
221   extends CachedProxyType with ProtoType with ValueTypeOrProto {...}
```

Listing 2.16 – Examples of ProtoTypes

LazyTypes and Completers

Lazy types are assigned to symbols that have not yet been provided a type. A lazy type is a suspended computation that will populate the type of a symbol on invocation. They are

stored as temporary types and will be invoked when this type is needed. Lazy types are used in Dotty to achieve two goals:

- avoiding loading classes and methods that are not necessary for the compilation;
- discovering and breaking false cycles during typechecking.

2.5 Symbols

Trees provide the information about the classes and methods that are currently being compiled. These methods may refer to classes and methods that have been compiled before in a separate compilation, preceding the current one. We will have neither the trees nor the source for those methods, but we still need a way to uniquely refer to their definitions, and that creates the need for Symbols. These classes and methods are commonly loaded from the bytecode and may come from other JVM languages, such as Java.

A Symbol uniquely identifies a definition. These definitions may be:

- classes, either top-level or inner or local;
- methods, either members of a class or local methods;
- fields, either mutable, or immutable or lazy;
- local variables;
- method parameters;
- type members of classes, including type arguments;
- temporary skolem symbols synthesized in subtyping checks.

At the same time, symbols generally exist only in a single run. Symbols do not store much information(see Listing 2.17), but we do track if the type level of a symbol identifies a term or a type. Among type symbols, we differentiate `ClassSymbols` that define a class or a trait and additionally track which file this class came from. All the data describing the semantic meaning of the symbol is stored inside the *Denotation* which this symbol refers to (see Section 2.10).

2.6 Flags

The most commonly used information about a symbol is stored in a way that is compact, fast to access and operate: as the bits of a 64-bit integer.

```
222 class Symbol {
223   type ThisName <: Name
224
225   /** The last denotation of this symbol */
226   private[this] var lastDenot: SymDenotation = _
227
228   def denot: Denotation = ...
229
230   final def isTerm(implicit ctx: Context): Boolean =
231     denot.isTerm
232   final def asTerm(implicit ctx: Context): TermSymbol = {
233     assert(isTerm, s"asTerm called on not-a-Term $this" );
234     this.asInstanceOf[TermSymbol]
235   }
236
237   final def isType(implicit ctx: Context): Boolean =
238     denot.isType
239   final def asType(implicit ctx: Context): TypeSymbol = {
240     assert(isType, s"isType called on not-a-Type $this");
241     this.asInstanceOf[TypeSymbol]
242   }
243
244   final def isClass: Boolean = isInstanceOf[ClassSymbol]
245   final def asClass: ClassSymbol = asInstanceOf[ClassSymbol]
246   ...
247 }
248
249 type TermSymbol = Symbol { type ThisName = TermName }
250 type TypeSymbol = Symbol { type ThisName = TypeName }
251
252 class ClassSymbol(val assocFile: AbstractFile) extends Symbol {
253   type ThisName = TypeName
254   ....
255 }
```

Listing 2.17 – Symbols

Some flags, such as the one indicating if a symbol is mutable, are only applicable to terms. Other flags, such as the one indicating if a type is contra or co-variant, are only applicable to types. There are also flags that are applicable to both terms and types such as the privateness of a symbol.

Because of this, the first two bits of a FlagSet are reserved to indicate if this FlagSet is applicable to types, terms, or both (see Listing 2.18).

2.7 Runs

A single Dotty compiler can be used for multiple compilations by creating different runs. Knowledge from previous compilations, such as information from the Java standard library, will be carried over between runs, speeding up subsequent compilations.

2.8 Phases and Periods

The compiler is split in multiple traversals over the Trees, which represent files being compiled. These traversals are called phases, and every phase is assigned a single period. Periods may span multiple phases, but are always in the same run (see Listing 2.19).

2.9 Compiler pipeline and laziness

The compiler definitely needs to read and analyze the entire codebase currently being compiled. At the same time, it is very uncommon for an application to refer to all classes and methods available on the classpath. Loading and computing all the information about all the classes available on the classpath is impractical; instead, definitions originating from the classpath are loaded and transformed lazily.

This creates the need for two different pipelines:

- a pipeline of Tree transformations, which eagerly transforms the codebase that is currently being compiled. This is the main compilation pipeline — it drives the compilation.
- a pipeline of Denotation transformations, that lazily transforms the meaning of types and symbols. This pipeline is invoked lazily when the main pipeline requires semantic information that has not yet been computed.

The other motivation for the need of several pipelines was presented in Section 2.2: we need to erase types in Typer to resolve overloads. This creates two possibilities:

- the denotation pipeline for a symbol or type can be *behind* the global tree transforma-

```
/** A FlagSet represents a set of flags. Flags are encoded as follows:
   *  The first two bits indicate whether a flagset applies to terms,
   *  to types, or to both.  Bits 2..63 are available for properties
   *  and can be doubly used for terms and types.
   *  Combining two FlagSets with `|` will give a FlagSet
   *  that has the intersection of the applicability to terms/types
   *  of the two flag sets. We check that this intersection is not empty.
   */
case class FlagSet(val bits: Long) extends AnyVal {
  /** The union of this flag set and the given flag set
     */
  def | (that: FlagSet): FlagSet =
    if (bits == 0) that
    else if (that.bits == 0) this
    else {
      val tbits = bits & that.bits & KINDFLAGS
      assert(tbits != 0, s"illegal flagset combination: $this and $that")
      FlagSet(tbits | ((this.bits | that.bits) & ~KINDFLAGS))
    }

  /** The intersection of this flag set and the given flag set */
  def & (that: FlagSet) = FlagSet(bits & that.bits)

  /** The intersection of this flag set with the complement of the given flag set */
  def &~ (that: FlagSet) = {
    val tbits = bits & KINDFLAGS
    if ((tbits & that.bits) == 0) this
    else FlagSet(tbits | ((this.bits & ~that.bits) & ~KINDFLAGS))
  }

  /** Does this flag set have a non-empty intersection with the given flag set?
   *  This means that both the kind flags and the carrier bits have a non-empty
   intersection.
   */
  def is(flags: FlagSet): Boolean = {
    val fs = bits & flags.bits
    (fs & KINDFLAGS) != 0 && (fs & ~KINDFLAGS) != 0
  }
}
```

Listing 2.18 – FlagSets in Dotty

```
294   /** A period is a contiguous sequence of phase ids in some run.
295    *  It is coded as follows:
296    *
297    *     sign, always 0        1 bit
298    *     runid                17 bits
299    *     last phase id:        7 bits
300    *     #phases before last:  7 bits
301    *
302    */
303   class Period(val code: Int) extends AnyVal {
304
305     /** The run identifier of this period. */
306     def runId: RunId = code >>> (PhaseWidth * 2)
307
308     /** The phase identifier of this single-phase period. */
309     def phaseId: PhaseId = (code >>> PhaseWidth) & PhaseMask
310
311     /** The last phase of this period */
312     def lastPhaseId: PhaseId =
313       (code >>> PhaseWidth) & PhaseMask
314   ...
315 }
```

Listing 2.19 – Periods

tion pipeline if we have not needed information about this symbol yet;

- the denotation pipeline for a symbol can be *ahead of* of the global tree transformation pipeline if we have needed to see the future type of this symbol, e.g. its erased type.

2.10 Denotations and Denotation Transformers

A denotation is the result of resolving a name during a given period. A denotation carries all the semantic information for a symbol:

- name;
- type or completer;
- signature;
- flags;
- annotations;
- privateWithin, which defines a package within which this member is private;
- denotation validity period.

```
316 abstract class Denotation(val symbol: Symbol) {
317   /** The type info of the denotation, exists only for non-overloaded denotations */
318   def info(implicit ctx: Context): Type
319
320   /** The type info, or, if this is a SymDenotation where the symbol
321    *  is not yet completed, the completer
322    */
323   def infoOrCompleter: Type
324
325   /** The period during which this denotation is valid. */
326   def validFor: Period
327
328   /** Is this a reference to a type symbol? */
329   def isType: Boolean
330
331   /** Is this a reference to a term symbol? */
332   def isTerm: Boolean = !isType
333
334   /** Is this denotation overloaded? */
335   final def isOverloaded = isInstanceOf[MultiDenotation]
336
337   /** The signature of the denotation. */
338   def signature(implicit ctx: Context): Signature
339 }
```

Listing 2.20 – Denotations in Dotty

Denotations contain a symbol, in case there is a single one that can identify all names that the denotation resolves to. In case there is no such symbol, a sentinel *NoSymbol* is used. In the snippet below, the denotation of the call to `r.f` will have *symbol=NoSymbol*.

```
340 class Foo { def baz: Int }
341 class Bar { def baz: Int }
342 val r: A | B =
343   if (random())
344     new Foo
345   else
346     new Bar
347
348 r.f
```

Listing 2.21 – Example of denotation with symbol=NoSymbol

A Denotation is either a SingleDenotation or a MultiDenotation. SingleDenotations store all semantic information about a single member. A MultiDenotation indicates that there are multiple entities with the same name (e.g., overloaded methods).

```
349 abstract class SingleDenotation(symbol: Symbol) extends Denotation(symbol) {
350   /** The next SingleDenotation in this run, with wrap-around from last to first. */
351   protected var nextInRun: SingleDenotation = this
352
353   /** Produce a denotation that is valid for the period of the given context */
354   def current(implicit ctx: Context): SingleDenotation = ...
355
356   ...
357 }
358
359 case class MultiDenotation(denot1: Denotation, denot2: Denotation) extends Denotation(
      NoSymbol) {
360   ...
361 }
```

Listing 2.22 – SingleDenotations and MultiDenotations in Dotty

SingleDenotations create circular linked lists, where every succeeding entry is the meaning of the previous one in the next period and the last meaning is followed by the first one. Consider the example below:

```
362 class C {
363   def id[T](t: T) = t
364 }
```

In this example, the type and the signature of the method *id* will be changed by erasure; the denotation cycle is illustrated by Figure 2.1.

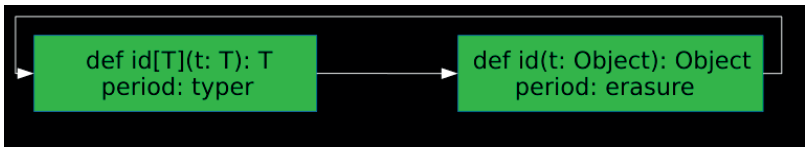


Figure 2.1 – Denotation cycle for id

Denotations for the entire compilation unit are illustrated in Figure 2.2

2.11 Measurements

This section contains various measurements that are helpful when reasoning about the performance of the compiler.

2.11.1 Frequency of trees

Figure 2.3 presents allocation statistics for different kinds of trees during an entire compilation run of Dotty compiling itself. This graph is instrumental in understanding the frequencies of

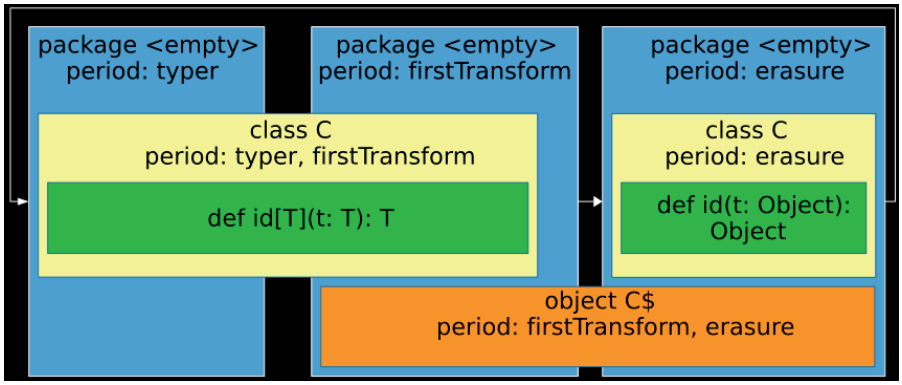


Figure 2.2 – Denotation cycle for class C

different tree kinds. *Ident*, *Apply* and *Select* are the most frequent nodes and together cover 53.8% of trees.

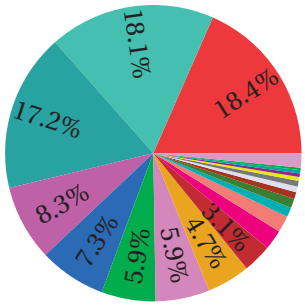
2.11.2 Frequency of types

Figure 2.4 presents allocation statistics for different kinds of types during an entire compilation run of Dotty compiling itself. As can be seen from the graph, term references are the most frequent kind of type, accounting for more than 60% of all allocated types.

2.11.3 Phase running time

Figure 2.5 shows the distribution of compilation times during the compilation of the Dotty compiler itself. This distribution is very characteristic of how Dotty compiles most common code bases. As can be seen, the frontend, which includes Parser and Typer, accounts for around 40% of the entire compilation run, while bytecode generation takes around 18%. Erasure takes around 8%. The remaining 24% are split among mini-phase blocks.

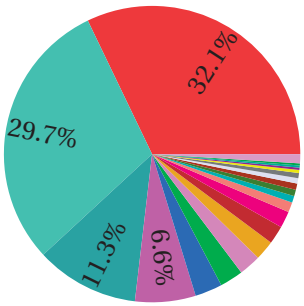
Figure 2.6 shows a similar distribution for compilation of the standard library. The standard library contains many complex inheritance hierarchies. Checking the correctness of overriding as well as generating bridges takes more time for such code. As can be seen in the graph, blocks which include mixin and refchecks take substantially larger portions of compilation time. This is because the complexity transformations implemented by those phases, namely overriding checks and trait composition is proportional to number of super classes in the inheritance hierarchy. Standard library contains classes with uncommonly large number super classes and thus represents an irregular codebase.



	Tree Class	count
	Ident	762635
	Apply	749869
	Select	713928
	TypeTree	342121
	Block	302485
	ValDef	244627
	DefDef	244374
	This	194221
	TypeApply	129870
	If	89533
	Literal	75250
	CaseDef	43026
	New	41523
	Template	30801
	Typed	26023
	TypeDef	25750
	Match	17628
	Assign	15779
	Closure	15238
	Bind	10581
	Others	64963

Tree Class	count
UnApply	10292
PackageDef	9009
Thicket	9002
Super	7472
JavaSeqLiteral	6303
AppliedTypeTree	5496
Import	4544
SeqLiteral	4212
Try	1655
Inlined	1204
TypeBoundsTree	1136
NamedArg	1091
Annotated	1076
SingletonTypeTree	956
Return	642
Alternative	552
ByNameTypeTree	165
LambdaTypeTree	110
BackquotedIdent	23
RefinedTypeTree	18
AndTypeTree	5

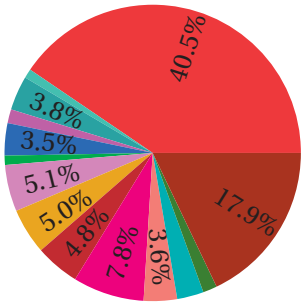
Figure 2.3 – Tree allocation counts when compiling Dotty



Type Class	count
TermRefWithFixedSym	1040708
TermRefWithSignature	962673
CachedMethodType	364890
CachedWildcardType	213971
ImplicitMethodType	96569
CachedSelectionProto	82369
PolyType	77698
CachedExprType	69218
FunProto	64416
RealTypeBounds	58440
TypeRefWithFixedSym	35247
CachedConstantType	23558
CachedClassInfo	22470
CachedViewProto	22240
CachedThisType	19007
TypeVar	18079
HKTypeLambda	10870
CachedAndType	7976
CachedSuperType	7704
CachedJavaArrayType	7351
Other	32294

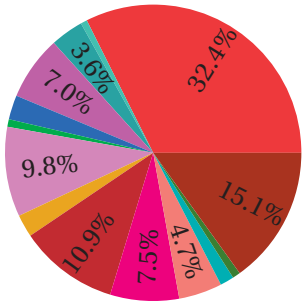
Type Class	count
CachedHKApply	6304
UnapplySelectionProto	5588
UnapplyFunProto	5588
JavaMethodType	3262
CachedOrType	3059
FunProtoTyped	2871
ErrorType	2759
TempClassInfo	1877
LambdaParam	977
RecType	9

Figure 2.4 – Type allocation counts when compiling Dotty



	Phases	Time
	frontend	18650 ms
	sbt-deps	461 ms
	posttyper	1727 ms
	sbt-api	697 ms
	pickler	1622 ms
	firstTransform, checkReentrant, elimJavaPackages	467 ms
	checkStatic, checkPhantomCast, elimRepeated, refchecks, normalizeFlags, extmethods, expandSAMs, tailrec, byNameClosures, liftTry, hoistSuperArgs, classOf	2341 ms
	tryCatchPatterns, patternMatcher, explicitOuter, explicitSelf, shortcutImplicits, crossCast, splitter	2311 ms
	vcInlineMethods, isInstanceOfEvaluator, seqLiterals, intercepted, getters, elimByName, augmentScala2Traits, resolveSuper, simplify, primitiveForwarders, functionXXLForwarders, arrayConstructors	2222 ms
	erasure	3585 ms
	elimErasedValueType, vcElideAllocations, mixin, LazyVals, memoize, nonLocalReturns, capturedVars, constructors, functionalInterfaces, getClass, simplify	1676 ms
	linkScala2Impls, lambdaLift, elimStaticThis, flatten, restoreScopes	1335 ms
	transformWildcards, moveStatic, expandPrivate, selectStatic, collectEntryPoints, collectSuperCalls, dropInlined, labelDef	708 ms
	genBCode	8245 ms

Figure 2.5 – Dotty compilation time per phase



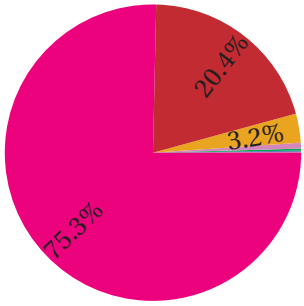
	Phases	Time
	frontend	12357 ms
	sbt-deps	270 ms
	posttyper	1372 ms
	sbt-api	2673 ms
	pickler	1011 ms
	firstTransform, checkReentrant, elimJavaPackages	313 ms
	checkStatic, checkPhantomCast, elimRepeated, refchecks, normalizeFlags, extmethods, expandSAMs, tailrec, byNameClosures, liftTry, hoistSuperArgs, classOf	3738 ms
	tryCatchPatterns, patternMatcher, explicitOuter, explicitSelf, shortcutImplicits, crossCast, splitter	939 ms
	vcInlineMethods, isInstanceOfEvaluator, seqLiterals, intercepted, getters, elimByName, augmentScala2Traits, resolveSuper, simplify, primitiveForwarders, functionXXLForwarders, arrayConstructors	4144 ms
	erasure	2866 ms
	elimErasedValueType, vcElideAllocations, mixin, LazyVals, memoize, nonLocalReturns, capturedVars, constructors, functionalInterfaces, getClass, simplify	1805 ms
	linkScala2Impls, lambdaLift, elimStaticThis, flatten, restoreScopes	577 ms
	transformWildcards, moveStatic, expandPrivate, selectStatic, collectEntryPoints, collectSuperCalls, dropInlined, labelDef	325 ms
	genBCode	5751 ms

Figure 2.6 – Stdlib compilation time per phase

2.11.4 Denotation cycle length

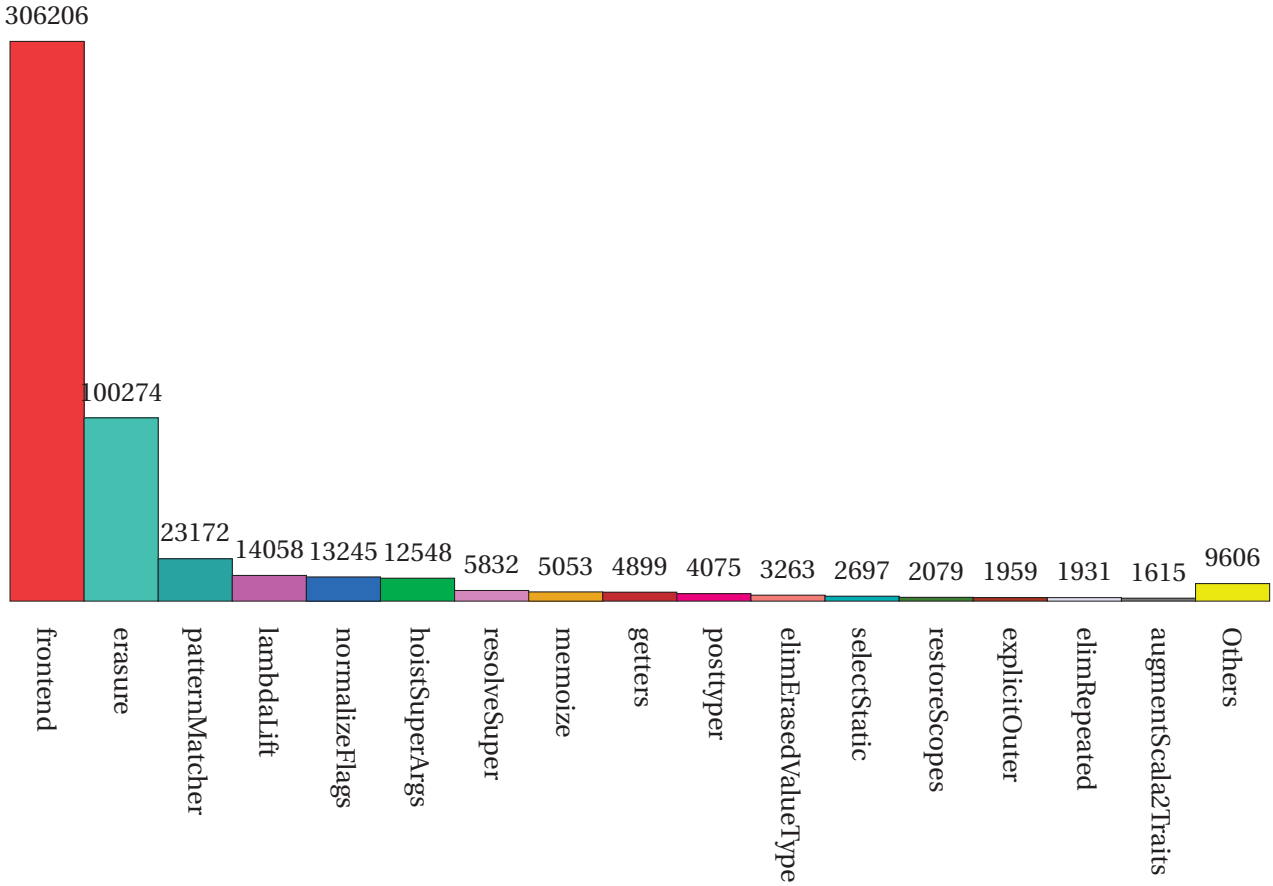
Figure 2.7 presents statistics of the length of denotation lists. As can be seen, most denotations have length 1. This is because these denotations represent methods and fields of classes that were loaded during classpath parsing but were not necessary for compilation and their denotation does not change during the compilation run.

Figure 2.8 shows which phases create new denotations. As can be seen, a small number of phases, namely Frontend, Erasure and PatternMatcher, account for most allocated denotations.



Cycle length	Count
10	1
9	11
8	80
7	379
6	406
5	851
4	2492
3	12415
2	79903
1	295090

Figure 2.7 – Distribution of Denotation cycle length during the compilation of Dotty



DenotationTransformer	denotations
frontend	306206
erasure	100274
patternMatcher	23172
lambdaLift	14058
normalizeFlags	13245
hoistSuperArgs	12548
resolveSuper	5832
memoize	5053
getters	4899
posttyper	4075
elimErasedValueType	3263
selectStatic	2697
restoreScopes	2079
explicitOuter	1959
elimRepeated	1931
augmentScala2Traits	1615
Others	9606

DenotationTransformer	denotations
byNameClosures	1596
flatten	1507
LazyVals	1421
mixin	1303
extmethods	915
vcInlineMethods	843
constructors	553
expandPrivate	489
firstTransform	316
capturedVars	298
elimByName	183
vcElideAllocations	127
moveStatic	28
liftTry	18

Figure 2.8 – Number of denotations created by each denotation transformer

3 Shared Backend Interface

Careful choice of abstractions allows us to create maintainable code that reads nicely and rarely needs to be modified.

In this chapter we will present the Backend interface. This abstraction was introduced during the early days of Dotty with the goal of reusing the bytecode emission from scalac, the current Scala compiler. It allows us to run the Backend in either Scalac or Dotty, as both these compilers provide an implementations for BackendInterface.

We have chosen to use implementations of BackendInterface for demonstrating an advantageous use of Scala abstractions:

- BackendInterface is probably the most abstract part of the Dotty compiler: it permits us to use the same backend efficiently for both scalac and Dotty, although those two compilers have different representations for the AST and classtable and do not cooperate.
- BackendInterface has to describe both low-level bytecode specific notions as well as Scala specific notions. It covers multiple layers that are well separated.
- This is one of the oldest parts of Dotty; its original design is unchanged since the very early days.
- In order to be easier to maintain, BackendInterface uses high-level types to introduce a post-hoc structure on both compilers.
- Over the course of Dotty development, as Dotty was implementing different design decisions, BackendInterface demonstrated the effectiveness of this approach.

All the language features necessary to implement such an API are available both in Scala 2 and in Dotty.

3.1 Abstracting over AST classes

Both scalac and Dotty use a tree-based representations for compilation units. They do not use the same classes and BackedInterface should be able to work with classes provided by both. For this purpose, it defines abstract types representing different AST nodes:

```
/* Interface to abstract over frontend inside backend.
 * Intended to be implemented by both scalac and dotc
 */
abstract class BackendInterface {
  type Flags      = Long

  type Constant   >: Null <: AnyRef
  type Symbol     >: Null <: AnyRef
  type Type       >: Null <: AnyRef
  type Annotation >: Null <: AnyRef
  type Tree       >: Null <: AnyRef
  type Modifiers  >: Null <: AnyRef
  type TypeDef    >: Null <: Tree
  type Apply      >: Null <: Tree
  type Select     >: Null <: Tree
  type TypeApply  >: Null <: Tree
  type ClassDef   >: Null <: Tree
  type Try        >: Null <: Tree
  type If         >: Null <: Tree
  type LabelDef   >: Null <: Tree
  type ValDef     >: Null <: Tree
  type Throw      >: Null <: Tree
  type Return     >: Null <: Tree
  ... // other trees
}
```

Listing 3.1 – AST node kinds in BackendInterface

Listing 3.1 introduces an API based on abstract types which are checked by the compiler but are all erased to *java.lang.Object*.

This is very handy as the runtime classes used to represent ASTs are different in Dotty and scalac. Those classes share no common base classes nor even interfaces.

We also need to create a way to use those classes uniformly inside backend. This requires:

- providing a way to pattern match over those abstract types, despite them being completely erased at runtime;
- providing a way to invoke methods on those abstract types, despite them sharing no common interfaces;
- providing a way to deconstruct those classes, despite them having different data layouts

and underlying representations

3.2 Pattern Matching on Abstract Types

```
tree match {
  ...
  case t: TypeApply =>
    generatedType = genLoadIf(t, expectedType)
  case _ =>
    abort(s"Unexpected tree in genLoad: $tree/${tree.getClass} at: ${tree.pos}")
}
```

Listing 3.2 – Example of pattern matching code from Backend

Backend uses dispatch code similar to Listing 3.2 to handle different kinds of trees. If the abstract type that defines *TypeApply* is erased, how do we support pattern matching over this type?

Both Dotty and Scalac support allow us to provide an implicit *ClassTag* that would be used during runtime to perform a type test:

```
abstract class BackendInterface {
  ....
  implicit val TypeApplyTag: ClassTag[TypeApply]
  implicit val ClassDefTag: ClassTag[ClassDef]
  implicit val TryTag: ClassTag[Try]
  implicit val AssignTag: ClassTag[Assign]
  implicit val IdentTag: ClassTag[Ident]
  implicit val IfTag: ClassTag[If]
  implicit val LabelDefTag: ClassTag[LabelDef]
  implicit val ValDefTag: ClassTag[ValDef]
  implicit val ThrowTag: ClassTag[Throw]
  implicit val ReturnTag: ClassTag[Return]
  ... // other class tags
}
```

Listing 3.3 – AST TypeTags in BackendInterface

The *unapply* method of those ClassTags will be invoked instead of a type test during runtime:

```
  var151_83 = interface().TypeApplyTag().unapply(var4_4);
  if (var151_83.isEmpty() || var151_83.get() == null || !true)
    throw interface().abort(new StringContext((Seq)Predef..MODULE$.wrapRefArray((Object
    [])new String[]{"Unexpected tree in genLoad: ", "/", " at: ", ""})).s((Seq)Predef..
    MODULE$.genericWrapArray((Object)new Object[]{tree, tree.getClass(), interface().
    treeHelper(tree).pos()})));
  else
    generatedType = this.genTypeApply(var4_4);
```

Listing 3.4 – Decompiled version the of snippet above with type test

3.3 Providing Methods on Abstract Types

You may have noticed that on Line 395 there is a call to method *pos* on an abstract type *Tree* that did not define a method *pos*.

The way this works is that those methods are added by an implicit decorator.

```
416 implicit def treeHelper(a: Tree): TreeHelper
417
418   abstract class TreeHelper{
419     def symbol: Symbol
420     def tpe: Type
421     def isEmpty: Boolean
422     def pos: Position
423     def exists(pred: Tree => Boolean): Boolean
424   }
```

This makes it possible to provide an API for an abstract type that itself is left abstract without requiring all implementations to collaborate by subclassing a common class.

The call compiles to the code below:

```
425   var151_83 = interface().TypeApplyTag().unapply(var4_4);
426   if (var151_83.isEmpty() || var151_83.get() == null || !true)
427     throw interface().abort(new StringContext((Seq)Predef..MODULE$.wrapRefArray((Object
      [])new String[]{"Unexpected tree in genLoad: ", "/", " at: ", ""})).s((Seq)Predef..
      MODULE$.genericWrapArray((Object)new Object[]{tree, tree.getClass(), interface().
      treeHelper(tree).pos()})));
428   else
429     generatedType = this.genTypeApply(var4_4);
```

Listing 3.5 – Decompiled version of the snippet above with decorated tree

3.4 Deconstructing Abstract Classes with Pattern Matching

The simple pattern matching presented in Listing 3.2 is not the common case. The common case includes pattern matching on structurally nested parts of the tree such as in the example below:

```
tree match {
  ...
  case app @ Closure(env, call, functionalInterface) =>
    val (fun, args) = call match {
      case Apply(fun, args) => (fun, args)
      case t @ Select(_, _) => (t, Nil)
      case t @ Ident(_) => (t, Nil)
    }
    ...
}
```

Listing 3.6 – Example of deconstructing in pattern matching code from Backend

In order to support this kind of pattern matching, we create deconstructors(Listing 3.7) that support name-based pattern matching [Dotty, 2015]. Note that this feature was only documented in Dotty, though scalac also supports a variant of it as well [Phillips, 2013].

This also serves as a way to access fields, such as in the snippet below:

```
val ArrayValue(tpt, elems) = av
```

Listing 3.8 – Accessing a field of an abstract class

3.5 Symbol interface

Dotty and scalac have vastly different representations for internal datastructures. The biggest disparity comes from Symbols: scalac symbols contain complete semantic information indicating their origins, while in Dotty, all information is encapsulated inside a Denotation.

BackendInterface provides a high level common API for Symbols that encapsulates intentions instead of low-level implementation details. For example, both Dotty and Scalac carefully pack information about a class into flags, but exposing those flags would be very tricky. High level methods, such `isPublic: Boolean`, are provided instead, that will be implemented using low level operations on flags (see Listing 3.9). A similar approach is taken for other semantic information, such as type, name and members of the symbol: the API conceals differences in internal representations between compilers.

A similar approach has been taken for Types, Positions, Names and Annotates: through a decorator we provide a high level API that hides internal representation details.

3.6 Case study: removing Throw tree

After two years of Dotty development we have decided to represent *throw* with a call to an intrinsified method instead of having a separate tree kind for it. This was a convenient opportunity to see if BackendInterface provides the right level of abstraction. In Listing 3.10 can find the entire patch needed to migrate from a separate tree to a kind of apply node:

```
440  val Closure: ClosureDeconstructor
441  ..// other Deconstructors
442  val Select: SelectDeconstructor
443  val Apply: ApplyDeconstructor
444
445  abstract class DeconstructorCommon[T >: Null <: AnyRef] {
446    var field: T = null
447    def get: this.type = this
448    def isEmpty: Boolean = field eq null
449    def isDefined = !isEmpty
450    def unapply(s: T): this.type ={
451      field = s
452      this
453    }
454  }
455
456  abstract class ClosureDeconstructor extends DeconstructorCommon[Closure]{
457      def _1: List[Tree] // environment
458      def _2: Tree // meth
459      def _3: Symbol // functionalInterface
460    }
461
462  abstract class SelectDeconstructor extends DeconstructorCommon[Select]{
463      def _1: Tree // qual
464      def _2: Name // name
465    }
466
467  abstract class ApplyDeconstructor extends DeconstructorCommon[Apply] {
468    def _1: Tree  // fun
469    def _2: List[Tree] // args
470  }
```

Listing 3.7 – Abstract type deconstructors

```
472 implicit def symHelper(sym: Symbol): SymbolHelper
473
474 abstract class SymbolHelper {
475     // names
476     def fullName(sep: Char): String
477     def fullName: String
478     def javaSimpleName: String
479     def javaBinaryName: String
480     ... // other name methods
481
482     // types
483     def info: Type
484     def thisType: Type
485
486     // tests
487     def isClass: Boolean
488     def isType: Boolean
489     def isAnonymousClass: Boolean
490     def isConstructor: Boolean
491     def isAnonymousFunction: Boolean
492     def isMethod: Boolean
493     def isPublic: Boolean
494     def isSynthetic: Boolean
495     ... // other tests
496
497      // members
498      def primaryConstructor: Symbol
499      def nestedClasses: List[Symbol]
500      def memberClasses: List[Symbol]
501      def annotations: List[Annotation]
502      ... // other kinds of members
503 }
```

Listing 3.9 – Symbol API in the BackendInterface

```
@@ -48,7 +48,7 @@ class DottyBackendInterface()(implicit ctx: Context) extends
    BackendInterface{
   type Ident           = tpd.Ident
   type If              = tpd.If
   type ValDef          = tpd.ValDef
-  type Throw           = tpd.Throw
+  type Throw           = tpd.Apply
   type Return          = tpd.Return
   type Block           = tpd.Block
   type Typed           = tpd.Typed
@@ -713,7 +713,16 @@ class DottyBackendInterface()(implicit ctx: Context) extends
    BackendInterface{
   }

   object Throw extends ThrowDeconstructor {
-    def get = field.expr
+    def get = field.args.head
+
+    override def unapply(s: Throw): DottyBackendInterface.this.Throw.type = {
+      if (s.fun.symbol eq defn.throwMethod) {
+        field = s
+      } else {
+        field = null
+      }
+      this
+    }
   }
```

Listing 3.10 – Changes performend to BackendInterfance implementation due to replacing Throw node with synthetic Apply

As can be seen from Listing 3.10, changing the underlying representation is very easy in such a design. The only necessary changes were to 1) indicate that a different class is used at runtime to represent nodes that have a semantic meaning of a *Throw* node; and 2) implement the right technique to test if the Apply node represents a `throw` statement.

3.7 Deconstructors & Decorators: choice between singletons and fresh objects

Consider the code presented in Listing 3.11, which is a simplified version of the working of BackendInterface:

The Line 559 shows how the code is written against such an API, while Line 563 shows the desugared versions of the same code.

Note that the call to `Try.unapply` on Line 566 stores the object `a` to the field of a globally accessible singleton on Line 539. This is done to save allocation, but comes at the cost of thread safety. A potential alternative implementation could have allocated an object per call to the unapply. We benchmark both implementations.

3.8 Performance impact

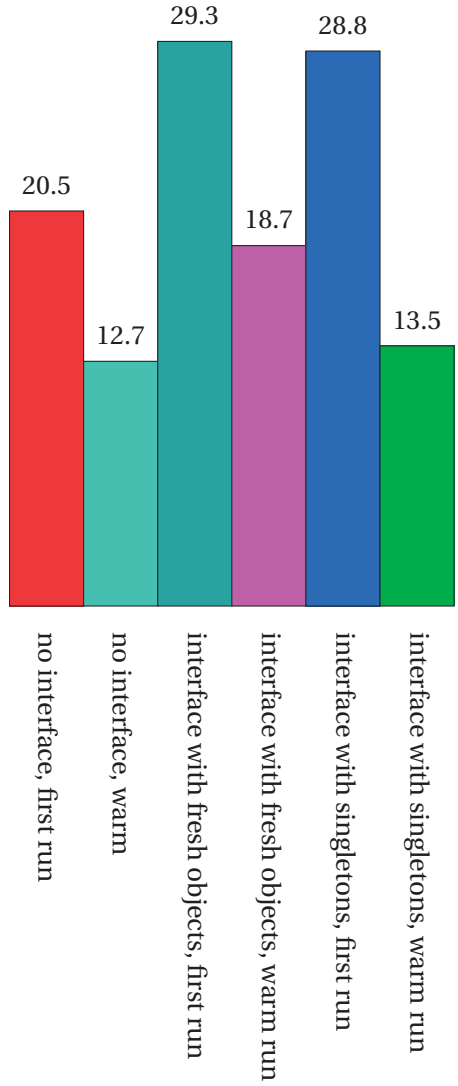
In oder to see what performance impact those additional abstractions have, we have implemented a BackendInterface implementation for scalac. We compared this implementation against the original bytecode emission phase that uses the scalac-specific API directly. We have benchmarked both the version that allocates a new object for every call and the version that uses global singletons.

As can be seen from Figure 3.1, both implementations of BackendInterface incur a substantial overhead on the first run. The overhead becomes substantially lower after the warmup. The likely explanation is that indirection through BackendInterface introduces a substantial slowdown for interpreted code, while higher tier compilers are able to eliminate and inline away most of it. This optimisation is able to trigger because in the runtime only a single subclass of BackendInterface is ever instantiated.

Unfortunately, the thread safe version with fresh objects performs 39% worse than the version that uses globally accessible singletons to store intermediate values. That is why the version used in Dotty is not thread safe.

```
trait Interface {
  type Try;
  val Try: TryDeconstructor
  implicit val TryTag: ClassTag[Try]

  abstract class DeconstructorCommon[T >: Null <: AnyRef] {
    var field: T = null
    def get: this.type = this
    def isEmpty: Boolean = field eq null
    def unapply(s: T): this.type ={
      field = s
      this
    }
  }
   abstract class TryDeconstructor extends DeconstructorCommon[Try]{
      def _1: Tree // expr
      def _2: List[Tree] // handlers
      def _3: Tree // finalizer
    }
}

object Implementation extends Interface {
  type Try = tpd.Try
  implicit val TryTag: ClassTag[Try] = ClassTag[Try](classOf[Try])
  object Try extends TryDeconstructor {
    def _1: Tree = field.expr
    def _2: List[Tree] = field.cases
    def _3: Tree = field.finalizer
  }

  def foo(a: Object) = a match {
    case Try(exp, cases, fin) => <body>
  }

  def foo_desugared(a: Object) = {
    val synth1:Option[Try] = TryTag.unapply(a)
    if (synth1.isEmpty) throw ...
    val synth2: Try = Try.unapply(synth.get())
    if (synth2.isEmpty) throw ...
    val synth3: Try = synth2.get()
    val exp = synth3._1
    val cases = synth3._2
    val fin = synth3._2
    <body>
  }
}
```

Listing 3.11 – Singleton based implementation



	Implementation	running time %
	no interface, first run	20.5
	no interface, warm	12.7
	interface with fresh objects, first run	29.3
	interface with fresh objects, warm run	18.7
	interface with singletons, first run	28.8
	interface with singletons, warm run	13.5

Figure 3.1 – Performance impact of BackendInterface

3.9 Related work

3.9.1 Scala Reflect

Scala Reflect[Coppel, 2008] is an API layer above Scalac trees that is used for meta-programming. Similarly to the BackendAPI, the intention was to provide a high level API that would be used to decouple code from the existing implementation. In the case of Scala Reflect, this was done to discourage meta-programmers from using functions that were not intended as part of the public API of the compiler.

The substantial difference is that Scala trees know about Scala Reflect, and in fact, they directly inherit them, implementing the API directly. Given that there is a single implementation, this was easier to achieve. In our case, two kinds of trees evolve separately and use slightly different guidelines for the API design. Agreeing on a common interface to inherit between two compilers is harder. This is true in particular because virtually all methods in Dotty take an instance of Context that contains global information. In scalac, almost all classes are inner classes of Global cake and don't need a reference to it.

3.9.2 Project Amber

Project Amber[Goetz and Rose, 2017] explores a possible direction for supporting pattern matching in the Java Language. One of the issues addressed in this project is how to extract subpatterns without boxing.

The project proposes a compilation scheme based on method handles that would solve the problem of multiple values returned by an inner patter, without introducing boxing. If this project is successful, the techniques proposed there will become an alternative to currently available approaches discussed in Section 3.7.

4 Miniphases: Compilation using Modular and Efficient Tree Transformations

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 the 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 section 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.

Attribution

The work presented in this chapter has been performed in collaboration with Martin Odersky and Ondřej Lhoták. The author of this thesis has proposed the idea of mini-phases as well its initial implementation — one that is close to the simplified version presented in this chapter. Professor Odersky has computed performance goals presented in Section 4.3 and together with the author of this thesis has developed a version that is currently in use in the Dotty compiler. This version uses reflection to pre-compute the transformation plan, rather than the function composition approach that was presented in this chapter. Ondřej Lhoták has helped considerably during discussions to find corner cases and work out an accessible forms of presentation.

This work has been published and presented at the 2017 ACM SIGPLAN International Conference on Programming Language Design and Implementation[Petrashko et al., 2017].

4.1 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 must provide 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 chapter, 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 chapter is organized as follows:

- Section 4.2 shows the conflict between modularity and performance requirements based on experience with Scala 2.x compilers;
- Section 4.3 presents target performance characteristics that we had in mind when designing the Miniphases framework;
- Section 4.4 introduces proposed design abstractions and describes the implementation inside the Dotty compiler;

- Section 4.5 presents the results of experiments that evaluate the impact of the Miniphases framework on GC object promotion rate and CPU cache misses;
- Section 4.6 covers limitations of the framework and soundness of fusion;
- Section 4.7 discusses real-world experience with the framework, such as maintenance cost and the on-boarding process for new contributors;
- Section 4.8 presents related work;
- Section 4.9 concludes.

4.2 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 chapter 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 us to reuse trees in multiple locations, and simplifies debugging since no mutation to trees is possible. When trees are modified, they are rebuilt using copiers. An optimization avoids this 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 [Odersky, 2014], 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 [Leontiev et al., 2016] and with higher kinds.

The execution of the compiler can be broadly divided into the front-end, the tree transformation pipeline, and the code generator. The front-end parses and type-checks source code, and generates trees annotated with type information. The tree transformations gradually

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 4.1 – Phases in Scala 2.12.0

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. In this chapter, our focus is on the middle phases, which constitute the tree transformation pipeline.

4.2.1 Experience with the Scala Compiler

In this section, we review the accumulated 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 4.1. In the compiler for Scala version 2.12.0, there are 24 such phases, listed in Table 4.1.

The Megaphase approach was originally intended to be modular in that each phase is an

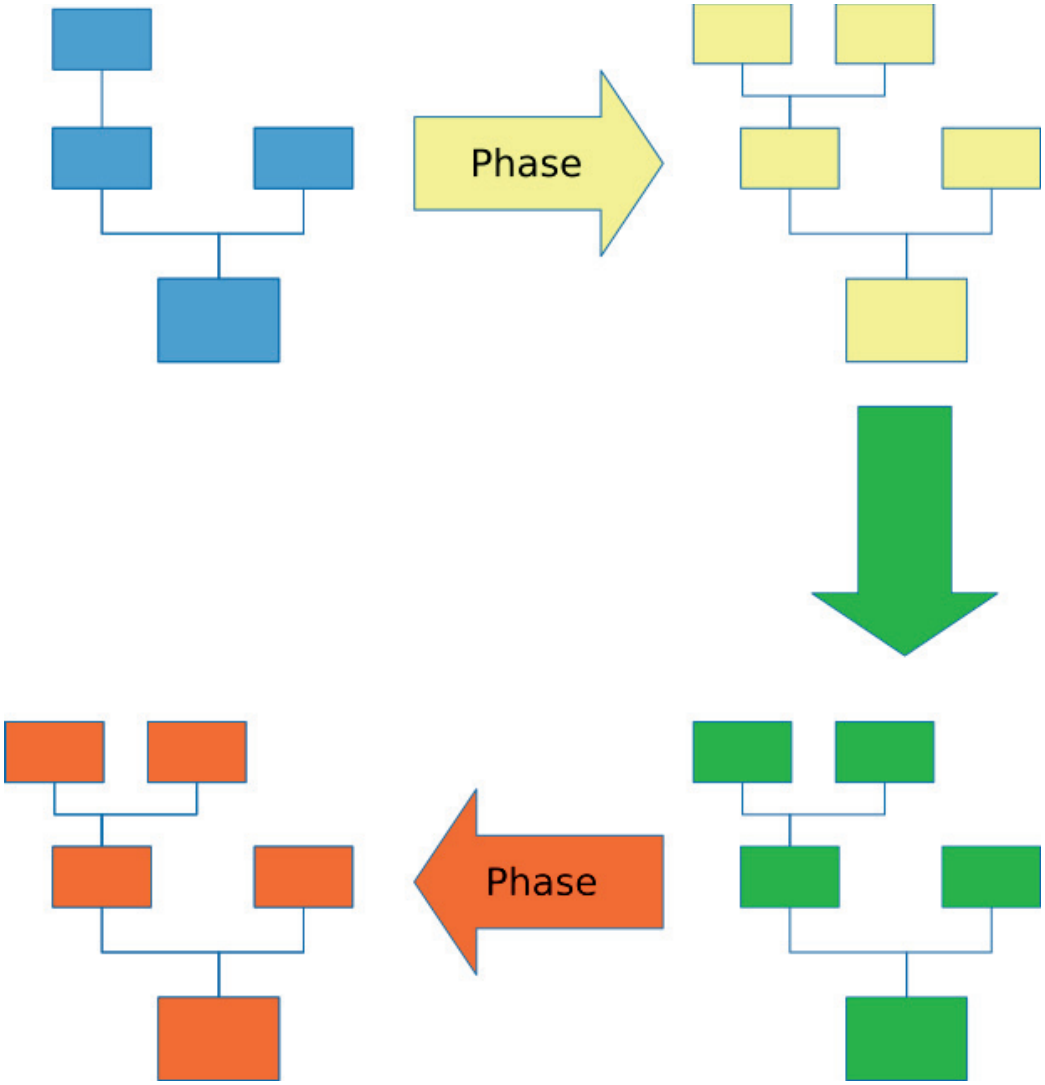


Figure 4.1 – Mega-phase based transformation of a tree

```
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 4.1 – Sample Scala program

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 presented in Listing 4.1 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. 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 simultaneously. 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.

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 absence of data races involving globals
RefChecks*	8	Various checks 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	Lift try expressions that execute on non-empty stacks
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	Issue warnings for unreachable statements in match 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
ElimByName*	28	Expand by-name parameters and arguments
AugmentScala2Traits*	29	Expand traits defined in Scala 2.11 to simulate old mixin
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& etc.
ElimErasedValueType*	33	Expand erased value types to their underlying types
VCElideAllocations*	34	Peep-hole optimization to eliminate 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 Scala 2.11 traits
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, populating 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 broken by 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 4.2 – Phases in Dotty compiler. The horizontal lines indicate blocks of Miniphases(*) that constitute a single transformation.

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 are related to the original purpose of the `refchecks` phase, nor to each other.

In this chapter, 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, for performance, applies the transformations in a common traversal of the tree. 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 4.2. We expect that the number of phases could increase to around 100 once the compiler is finished.

4.3 Target Performance Characteristics

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

Based on user feedback about 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.

4.4 Design

Listing 4.2 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

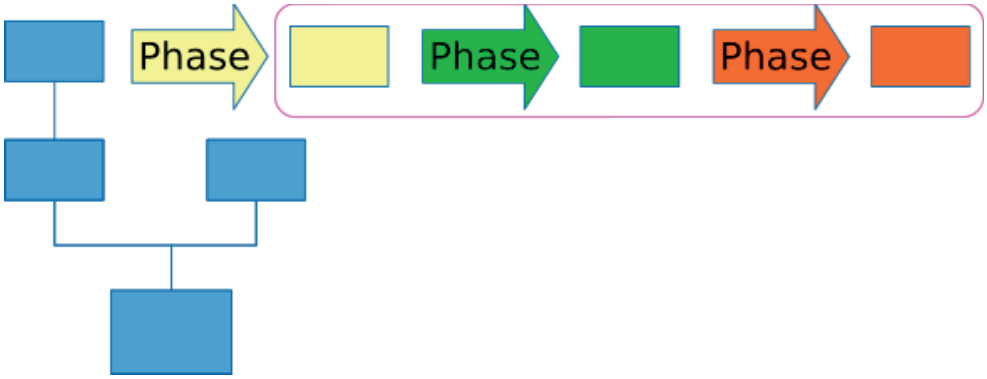


Figure 4.2 – Pipelining of a leaf-node through Miniphases

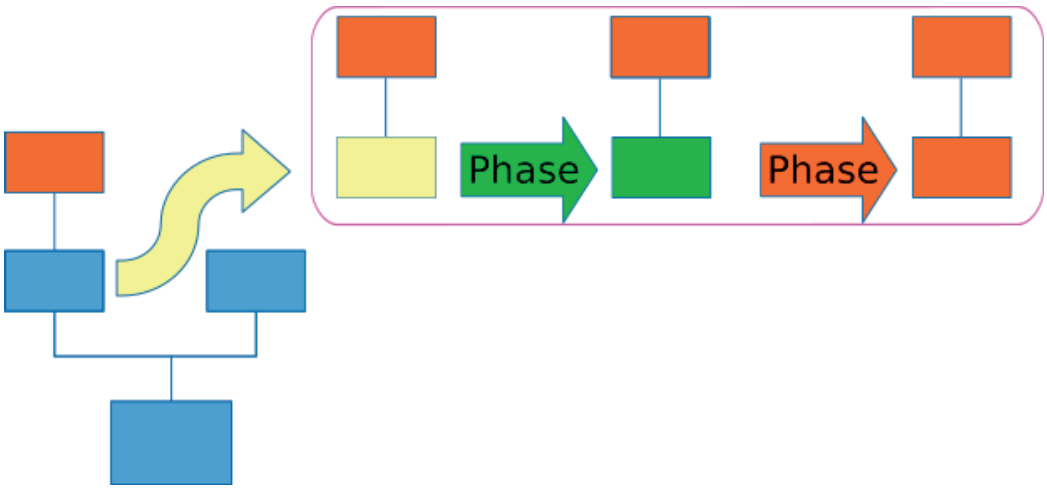


Figure 4.3 – Pipelining of an inner-node through Miniphases

```
586 abstract sealed class Tree {
587   def tpe: Type
588   def withNewChildren(list: List[Tree]): Tree
589   def children: List[Tree]
590 }
591 class Ident(sym: Symbol) extends Tree
592 class Select(from: Tree, name: String) extends Tree
593 ...
594 class ValDef(sym: Symbol, rhs: Tree) extends Tree
595 class DefDef(sym: Symbol, rhs: Tree) extends Tree
596 class CompilationUnit(trees: List[Tree]) extends Tree
```

Listing 4.2 – Tree nodes

```
597 def compileUnits(units: List[CompilationUnit], phases: List[Phase]) = {
598   var units1 = units
599   for (phase <- phases)
600     units1 = units1.map(unit => phase.runPhase(unit))
601 }
```

Listing 4.3 – Overall traversal

children.

The tree transformation pipeline has the overall structure given in Listing 4.3. 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 4.4. 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 4.5. The 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 4.2 and 4.3. In Figure 4.2, 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 4.3, the parent of the now orange node is processed by the same three

```
602 class Phase {
603   def runPhase(t: Tree): Tree
604
605   val runsAfter: Set[MiniPhase] = Set.empty
606   def checkPostCondition(t: Tree): Boolean = true
607 }
608
609 class MiniPhase extends Phase {
610   val valDefTransform: ValDef => Tree = id
611   val defDefTransform: DefDef => Tree = id
612   val identTransform: Ident => Tree = id
613   ...
614   val selectTransform: Select => Tree = id
615
616   final def transform(t: Tree) = t match {
617     case a: ValDef => valDefTransform(a)
618     case a: DefDef => defDefTransform(a)
619     ...
620     case a: Select => selectTransform(a)
621   }
622
623   final def runPhase(t: Tree): Tree = {
624     val newChildren =
625       t.children.map(sub => runPhase(sub))
626     val reconstructed = t.withNewChildren(newChildren)
627     transform(reconstructed)
628   }
629 }
```

Listing 4.4 – Definition of a Miniphase

```
630 private def chainMiniPhases(first: MiniPhase, second: MiniPhase) = {
631   new MiniPhase {
632     val valDefTransform =  { x: ValDef =>
633       val newTree = first.valDefTransform(x)
634       second.transform(newTree)
635     }
636
637     ... // similar to valDefTransform for all node kinds
638
639     val runsAfter: Set[MiniPhase] =
640       second.runsAfter -- first ++ first.runsAfter
641
642     def checkPostCondition(t: Tree) =
643       first.checkPostCondition(t)  &&
644       second.checkPostCondition(t)
645   }
646 }
647
648 def combine(a: Array[MiniPhase]): MiniPhase =
649   a.reduceRight((phase, acc) =>
650     chainMiniPhases(phase, acc)
651   )
```

Listing 4.5 – Fusion algorithm for Miniphases

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 4.3, 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, rarely breaking existing ones.

We will discuss in Section 4.6 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

```
652 private def chainMiniphases(first: Miniphase, second: Miniphase) = {
653   new Miniphase {
654     val valDefTransform =
655       if (first.valDefTransform == id)
656         second.valDefTransform
657       else if (second.valDefTransform == id)
658         first.valDefTransform
659       else { x: ValDef =>
660         val newX = phase.valDefTransform(x)
661         newX match {
662           case newX: ValDef =>
663             second.valDefTransform(x)
664           case other: Tree =>
665             second.transform(other)
666         }
667     ...  // similar changes form all AST nodes
668     }
669 }
```

Listing 4.6 – Optimization for identity transforms and for transformations that keep the same node kind

in Listing 4.6.

First, since most Miniphases transform only a small subset of the types of tree nodes, the fusion code explicitly checks (Section 4.4, Listing 4.6) 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.

4.4.1 Prepares

The Miniphase framework presented so far is sufficiently general to implement all but four Miniphases present in the Dotty compiler. The remaining four phases, however, perform transformations that depend on the ancestors of the current 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 4.2.1. 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

```
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 4.7 – MiniPhase extended with prepares

```
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 4.8 – Fusion with prepares

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 `prepares`, as shown in Listing 4.8.

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

4.4.2 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 a high memory footprint and memory leaks.

4.5 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, as 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. The benchmarks were executed on a server with two Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80 GHz CPUs, running on a fixed frequency of 2.4 Ghz with HyperThreading disabled. This CPU has a 25 MB L3 cache. Every one of the 10 cores in this CPU additionally has a 256 KB L2 cache and 32 KB 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 [Intel Corporation, 2016].

This server has 64 GB 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.

4.5.1 Overall Time

Figure 4.4 shows the overall running time of the frontend, tree transformation pipeline, and backend. The tree transformations use a significant fraction 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.

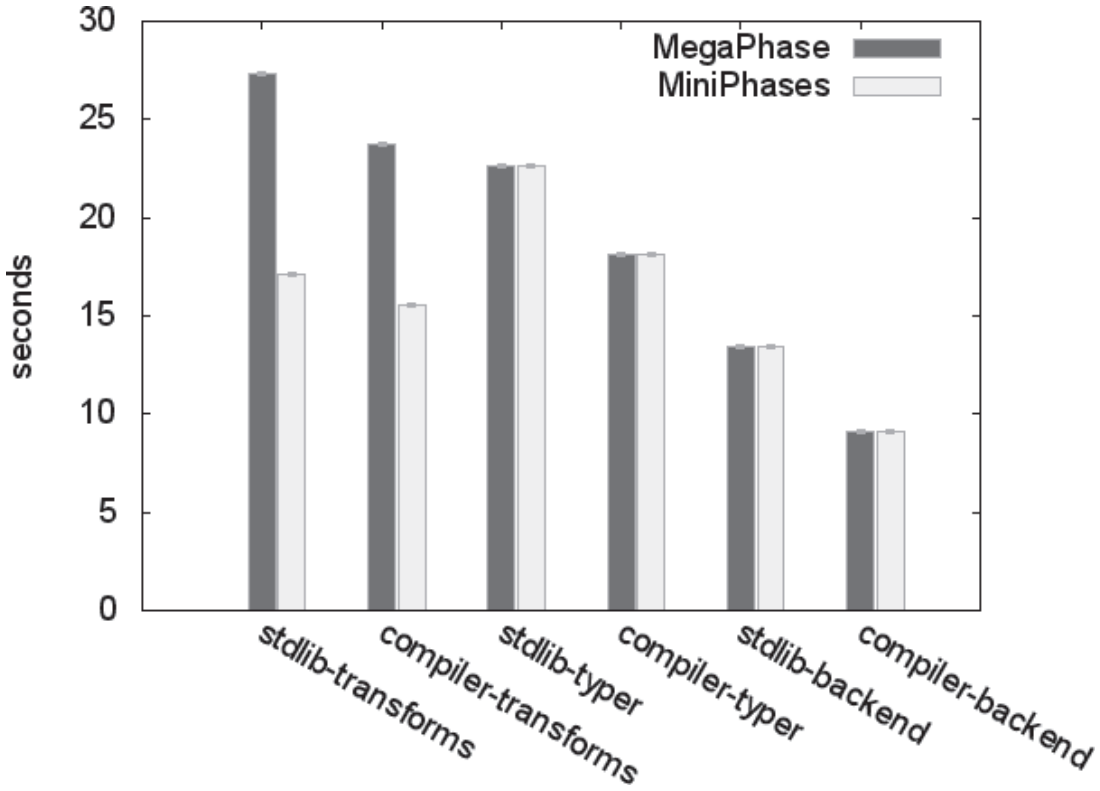


Figure 4.4 – Execution time of tree transformation passes, typechecker, and code generation backend in Miniphase and Megaphase versions of the Dotty compiler.

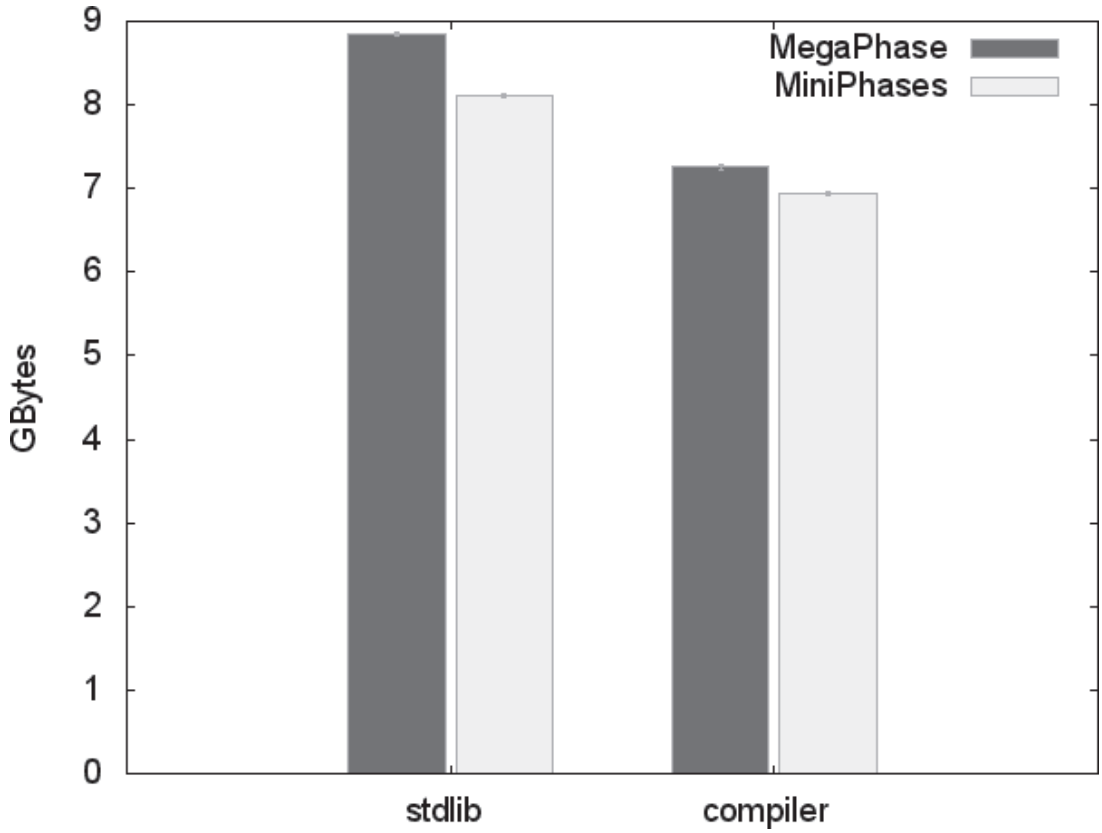


Figure 4.5 – Total size of GC object allocated, GBytes

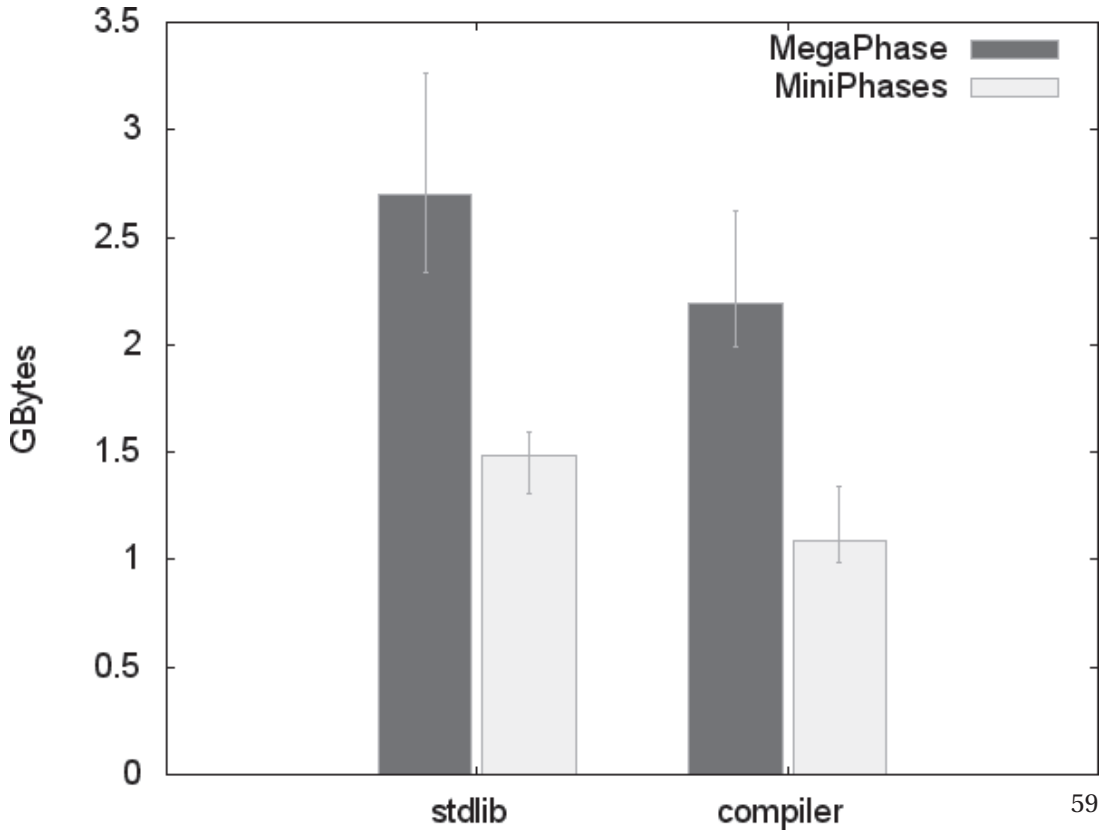


Figure 4.6 – Total size of GC object tenured, GBytes

4.5.2 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 that were 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 for batch compilation in a big project.

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 4.5 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, from 7 to 9 GB, so even a decrease of 9% amounts to a lot of memory. Note that this refers to 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 number of objects promoted to the old generation is much more significant, even in a relative sense, as shown in Figure 4.6. The reduction thanks to Miniphases is a 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.

4.5.3 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.

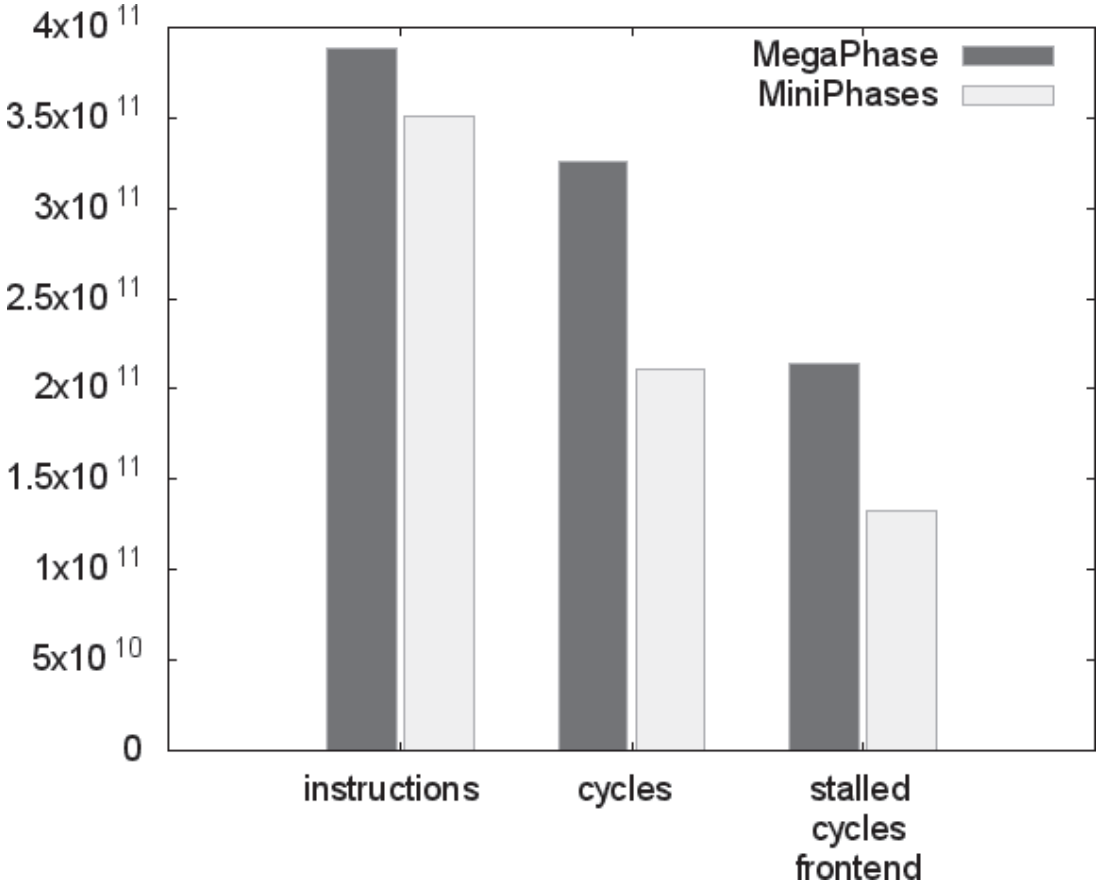


Figure 4.7 – Instructions and cycle counters

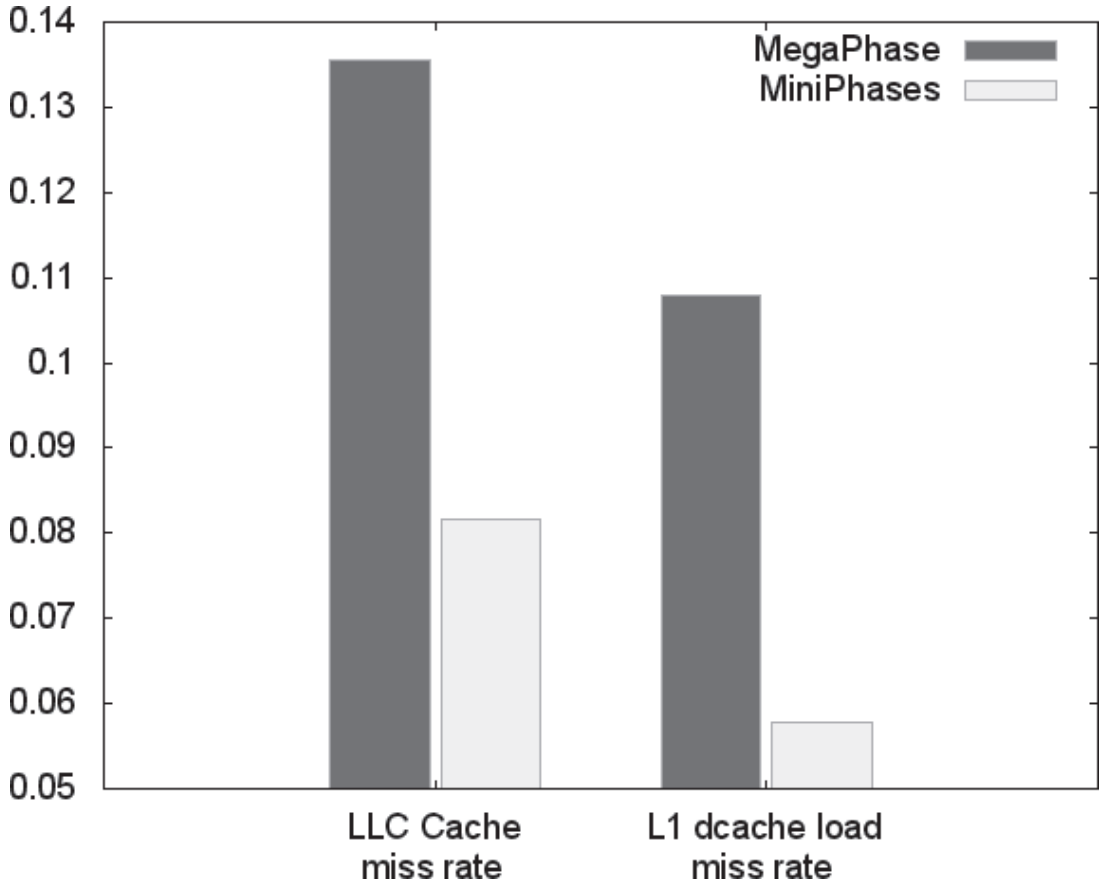


Figure 4.8 – L1 and LLC cache miss rates

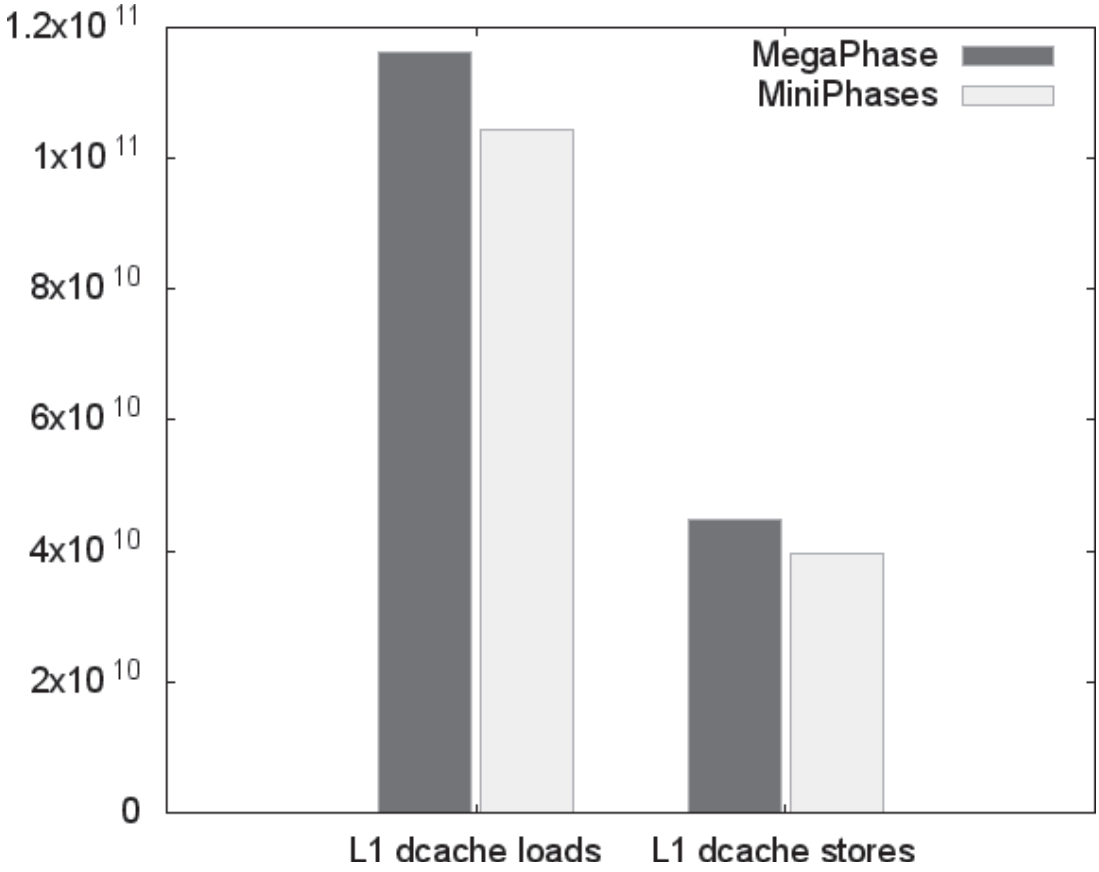


Figure 4.9 – L1 dcache miss rates

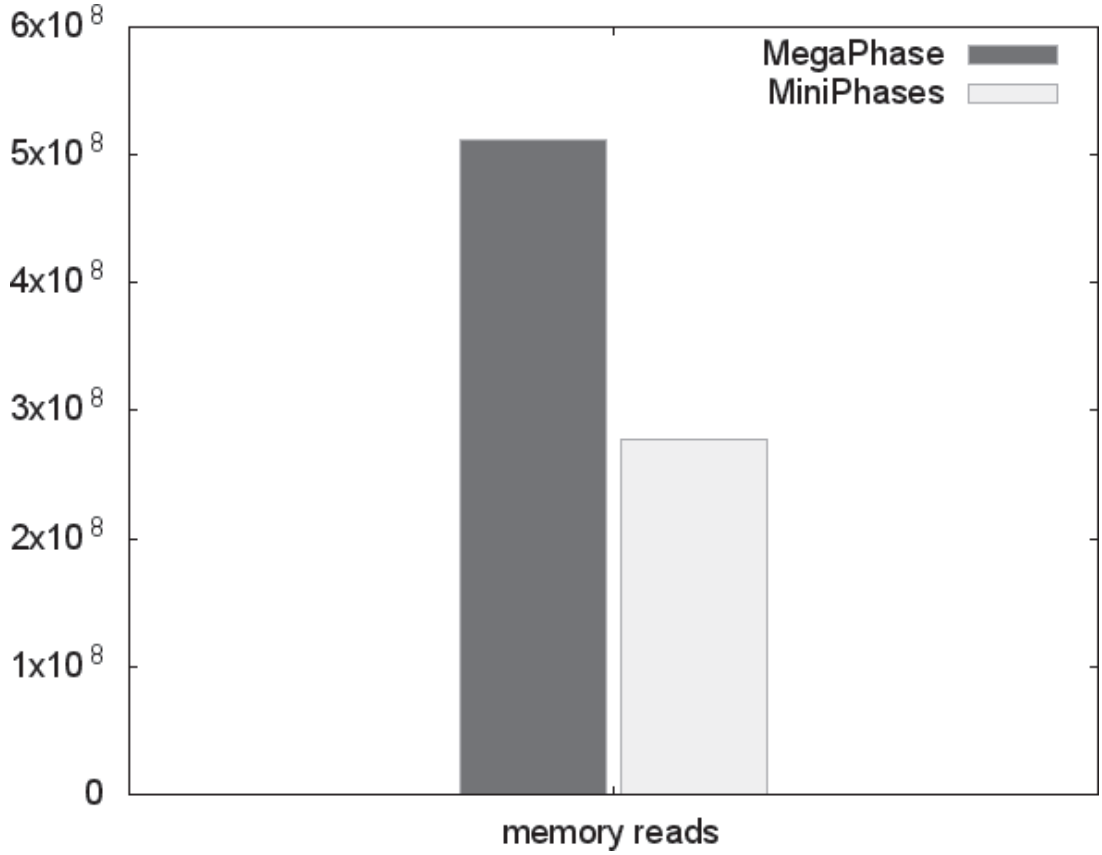


Figure 4.10 – Number of memory reads

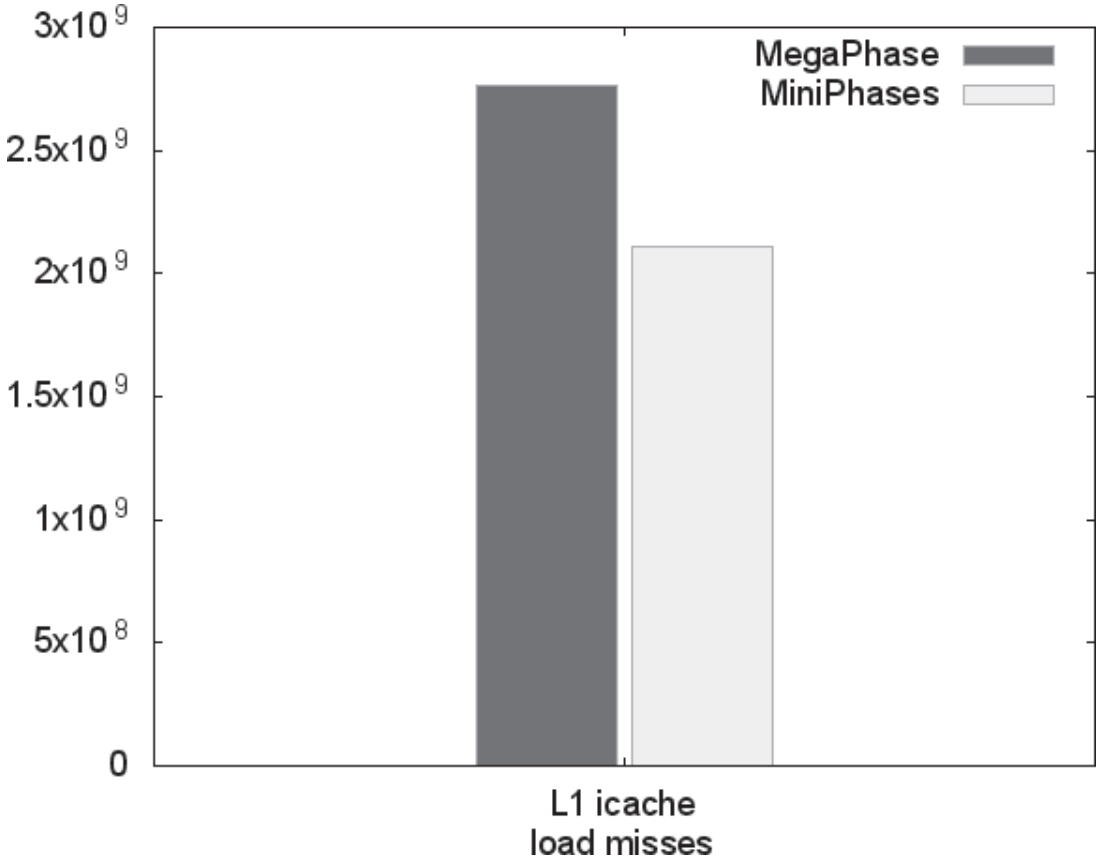


Figure 4.11 – L1 icache miss rate

Figure 4.7 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 Section 4.5.2, 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. Section 4.5.2 indicates that the total number of cache accesses decreased by only 10%. Section 4.5.2 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.

Section 4.5.2 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 the 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.

4.5.4 Comparison with Existing Production Compiler

To put the running times of the Dotty compiler with Miniphases in perspective, Figure 4.12 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`, although 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`, respectively.

4.6 Soundness and Limitations of Phase Fusion

4.6.1 Fusion Criteria

We do not formally define criteria that would give soundness guarantees in the form of a promise that fusing phases does not change their behaviour. To be sound, any such formal criteria would have be conservative. Such criteria can supply guarantees for simple programs in which tree traversals affect a small number of well-behaved data structures. However, these criteria 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.

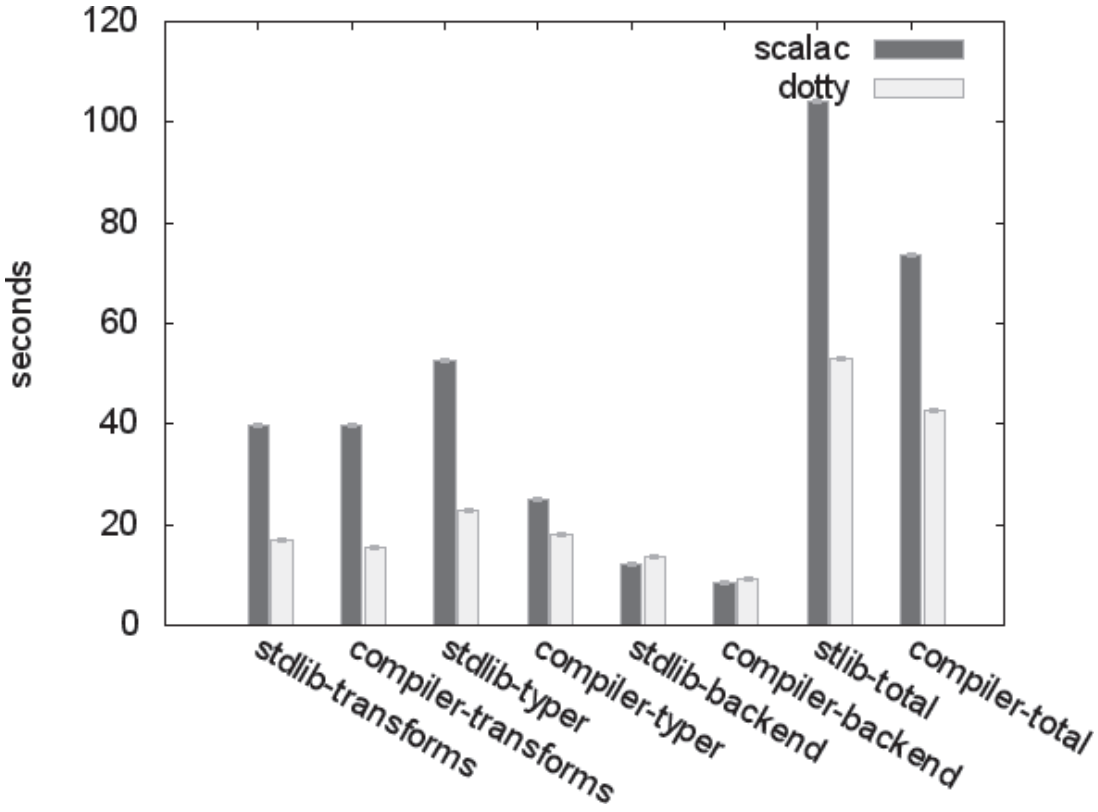


Figure 4.12 – Execution time of stages of the Dotty and `scalac` compilers when compiling the standard library and Dotty .

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 among 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.
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 4.6.3; but first, we examine examples of phases that are not fused because they violate the fusion criteria.

4.6.2 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 six separate blocks of Miniphases, marked with (*) in Table 4.2. 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 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.

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.

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 the absence of member selections on union types [Pierce, 1991]. Union types are eliminated by the splitter phase, which must 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.

4.6.3 Phase Preconditions and Postconditions

Since the criteria from Section 4.6.1 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 any bugs to specific phases.

Each Miniphase defines postconditions that must hold about the tree nodes after the phase has transformed them. The `checkPostcondition` method (Listing 4.4) of the Miniphase implements the runtime tests that enforce postconditions. The intended meaning of the postconditions is that if one Miniphase establishes a postcondition, all later Miniphases must preserve it.

During testing, a checker pass is inserted between phases. A simplified version of its implementation is shown in Listing 4.9. The pass first checks various global invariants that are expected to always hold between any phases. 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 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 match-

ing after the phase that eliminates pattern matching, we know immediately which phase is 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 that the tail recursion elimination phase 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 4.6.1. 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 postcondition 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; however, 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.

4.7 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.

```
698 class TreeChecker(previousPhases: List[Phase], typer: Typer) extends Phase {
699  def runPhase(t: Tree): Tree = {
700    t.forAllSubtrees{subt =>
701      val reTyped = typer.typeCheck(subt.stripTypes)
702
703      reTyped.hasSameTypes(subt) &&
704      checkNoDoubleDefinitions(subt) &&
705      checkValidJVMNames(subt) &&
706      checkcheckNoOrphanTypes(subt) &&
707      /* other non-phase-specific sanity checks*/
708      previousPhases.forAll { phase =>
709        phase.checkPostCondition(subt)
710      }
711    }
712  }
713  ... // implementations of hellper methods such as checkNoDoubleDefinitions
714 }
```

Listing 4.9 – Simplified version of TreeChecker

4.7.1 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 just getting 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 renders them 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 it does it.

4.7.2 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.

4.7.3 Onboarding Process

Open-source contributors frequently ask how they can get involved and learn about the 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.

4.7.4 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 4.6.

Based on our experience, most people who contribute to the compiler lie on one of two extremes: either they are experts who have been working on the compiler for a long time and know the entire pipeline, or they only appear to make a small contribution once in a while. While the first group doesn't need any guidance on 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 will 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 who maintain the repository. At the same time, continuous integration systems run tests that verify that pre- and post-conditions hold for the entire test suite, which includes the compiler itself, the standard library, and several thousands of programs contributed by the community.

4.8 Related Work

4.8.1 Deforestation and Stream Fusion

The original inspiration for the Miniphase approach was prior work on "deforestation" [Coutts et al., 2007; Gill, 1996; Wadler, 1990]. 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 that are consumed by subsequent phases. Thus, the general deforestation technique cannot be applied because it would change the semantics of the transformations.

4.8.2 Sound Fusion in Tree Traversal Languages

In this section, we describe several domain-specific tree traversal languages and frameworks that, while being more general than the functions that can be fused by deforestation, are 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 [Knuth, 1968] 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 [Ekman and Hedin, 2007] 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 precomputed 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 [Kastens, 1980, 1991; Lewis et al., 1974]. More general classes of attribute grammars require multiple passes; algorithms have been proposed for finding evaluation orders that minimize the number of such passes [Alblas, 1991; Riis Nielson, 1983]. These techniques have been extended to evaluation of attributes of multiple tree nodes in parallel [Jourdan, 1991]. Meyerovich et al. [Meyerovich et al., 2013] 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 enhance data locality by rewriting recursive programs that traverse trees [Jo and Kulkarni, 2011, 2012; Weijiang et al., 2015]. Jo and Kulkarni [Jo and Kulkarni, 2011] 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. This yields 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 [Jo and Kulkarni, 2012] extended the idea of point blocking into a similar but more sophisticated technique: traversal splicing. This strategy improves the locality of irregular tree traversals that traverse only a subset of the nodes of the tree. Weijiang et al. [Weijiang et al., 2015] 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. [Rajbhandari et al., 2016a,b] 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 pre-defined. Compared to the dependence test of Weijiang et al. [Weijiang et al., 2015], 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 the 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.

4.8.3 Other Pass Fusion Approaches

ASM [Bruneton et al., 2002] 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 4.4.1, 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 comparatively 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, rather than 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 that is 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 [Lepper and Trancón y Widemann, 2011] 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 the traversal of 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.

4.8.4 Compilers Based on Tree Transformation Passes

The Nanopass Framework [Sarkar et al., 2005] 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 [Nystrom et al., 2003] is structured as a sequence of passes that successively transform trees, in this case from various extensions of Java to Java itself. As 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.

4.9 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. We demonstrated these speedups 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 significant 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.

5 Types as Contexts in Whole Program Analysis

Contemporary object oriented languages provide a natural paradigm, but at the cost of run-time overhead. Method specialization or inlining could reduce this cost, but they require precise call graph analysis.

Existing static call graph analyses do not take advantage of the information provided by the rich type systems of contemporary languages, in particular generic type arguments. Many existing approaches analyze Java bytecode, in which generic types have been erased. This section shows that this discarded information is actually very useful in providing the context for a context-sensitive analysis, where it significantly improves precision and keeps the running time short. Specifically, we propose and evaluate call graph construction algorithms in which the contexts of a method are (i) the type arguments passed to its type parameters; and (ii) the static types of the arguments passed to its term parameters. The use of static types from the caller as context is effective because it allows more precise dispatch of call sites inside the callee.

Our evaluation indicates that the average number of contexts required per method is small. We implement the analysis in the Dotty compiler for Scala, and evaluate it on programs that use the type-parametric Scala collections library and on the Dotty compiler itself. The context-sensitive analysis runs twice as fast as a context-insensitive one and discovers 20% more monomorphic call sites at the same time. When applied to method specialization, the imprecision in a context-insensitive call graph would require the average method to be cloned 22 times, whereas the context-sensitive call graph involves a much more practical 1.00 to 1.50 clones per method.

We applied the proposed analysis to automatically specialize generic methods. The resulting automatic transformation achieves the same performance as state-of-the-art techniques requiring manual annotations, while reducing the size of the generated bytecode by up to five times.

Attribution

The work presented in this section was performed in collaboration with Vlad Ureche and Ondřej Lhoták. Vlad Ureche helped in the comparison with the miniboxing technique. This work is based on previous work by Ondřej Lhoták; his help was instrumental in simplifying the algorithm and the presentation. The actual algorithm was proposed, implemented and evaluated by the author of this thesis.

This work has been published in and was presented at the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications [Petrashko et al., 2016].

5.1 Introduction

Modern programming languages support modularity and scalability using abstraction facilities such as generic methods, interfaces and abstract type members. Unfortunately, these abstractions incur non-negligible performance costs. Optimizing compilers are very good at eliminating performance overheads when they can analyze the whole code fragment. However, abstraction facilities encourage code to be distributed between multiple methods which are called using dynamic dispatch. The call sites are often megamorphic[1]. Most compilers do not try to remove or inline megamorphic dispatch, which prevents other optimization opportunities. To reduce the performance overhead of modern abstraction facilities, a first step is to inline, or at least devirtualize, the method calls in hot code fragments.

For this reason, state-of-the-art JIT compilers perform inlining as one of the first, and crucial, optimization steps. The JIT setting enables precise techniques such as profile-directed and speculative inlining. However, the optimization opportunities necessary to eliminate the performance overhead of abstraction facilities often arise only after many levels of inlining. In many cases, JIT compilers do not inline enough to reach those opportunities [Click, 2011]. JIT compilers also do not have access to the rich type information available at the source code level.

Devirtualization and inlining are possible if the call site is proven to be monomorphic[2]. In order to be sound and computable, a static analysis must be conservative: in some cases, it must overestimate the set of potential dispatch targets. We say that a call graph is more precise than another if it contains fewer spurious dispatch targets that could never be called at run time. One possible approach to improving precision is to construct context-sensitive call graphs by specializing a given call site for the different contexts in which it is executed. A call site that dispatches to a different call target in each different context is megamorphic in a context-insensitive call graph, even though each context-sensitive instance of that call site may be monomorphic. Unfortunately, context sensitive analysis is often costly, and the

[1] ↑have 3 or more potential dispatch locations
[2] ↑has only one possible dispatch location

resulting context-sensitive call graphs are large, making the client analyses that use them costly as well [Lhoták and Hendren, 2008; Smaragdakis et al., 2011].

Analysis of call targets has long benefited from static types. Class hierarchy analysis [Dean et al., 1995] relies entirely on the static types of receivers to determine call targets. In propagation-based points-to analysis for Java (which is used in precise call graph construction algorithms), it has long been recognized that filtering points-to sets using static type information is critical for precision and efficiency [Lhoták and Hendren, 2003].

Existing approaches to call graph construction do not take full advantage of the information provided by the type systems of modern programming languages. Most recent work in the context of object oriented languages targets Java bytecode. When Java programs are compiled to bytecode, generic type parameters and arguments are erased, so they are not available to bytecode-based analyses. In this chapter, however, we show that this discarded type information is actually very useful: it enables us to construct more precise call graphs efficiently to enable devirtualization, and it provides the information necessary for specialization.

An interprocedural analysis is *context-sensitive* if it analyzes each method multiple times in different *contexts*. Ideally, the static contexts are selected so that invocations of the method with dissimilar run-time behaviours are abstracted by different analysis contexts, enabling the analysis to focus precisely on each behaviour. In the specific case of a call graph analysis, it is possible that a call site dispatches to multiple target methods overall, but is monomorphic in each specific analysis context. Unfortunately, in many analyses, the number of contexts often grows very large. As a result, the analysis becomes expensive and its output large, which makes client analyses expensive as well.

Our novel insight is that static type arguments, which have been erased in most previous work, are actually very effective contexts for call graph construction. Often, the static type of the receiver at a call site is a type parameter of the method in which the call site appears, or of the enclosing class of that method. Analyzing the enclosing method separately for each argument type provides static type information that is often precise enough to resolve the call to a single target method (i.e., monomorphically). Moreover, the number of contexts in which the average method needs to be analyzed remains small. At a given call site (in a given context), only *one* static type is passed as the argument for each type parameter, so the number of contexts grows only when a type parameter is actually used with different type arguments in multiple places in the program.

Call graphs contain the information needed for devirtualization, but building them with static types as context also provides the information needed for specialization. One common specialization criterion is to create distinct implementations of polymorphic methods, and of methods in generic classes, for each type argument with which the method or containing class is instantiated. The context-sensitive call graph provides exactly the set of type arguments with which each parameter may be instantiated, and this is the set of specialized methods that need to be generated.

The context-sensitive call information is well suited to devirtualization after specialization has been applied. In particular, the context-sensitive call graph may say that a call site is monomorphic, but only in some specific context. Since the analysis contexts correspond directly to the specialized method implementations, this is exactly the information that is needed to know that a call site in a specific specialized implementation can be devirtualized.

In this Chapter, we propose and evaluate call graph construction algorithms designed for static devirtualization and specialization [Dragos and Odersky, 2009]. The specialization is intended both to enable a static optimizer to perform further performance optimizations using knowledge of high-level language features, as well as to enable a JIT compiler to perform low-level optimizations on the devirtualized and specialized code. Our call graph construction algorithms use the rich type information available at the source code level to define context abstractions that are both effective in supporting devirtualization and keeping the size of the resulting context-sensitive call graphs manageable.

We will present our analysis for Scala [Odersky and Zenger, 2005]. It is possible to apply the proposed techniques to other languages that have abstraction features such as multiple inheritance, generics or type members. With a few exceptions, generic type parameters have been largely ignored in the literature on call graph construction. One reason for this is that, at least in the case of Java, most call graph construction algorithms are studied on Java bytecode, in which type parameters have been erased. Yet our results show that modeling type parameters precisely significantly improves call graph precision. Future work should evaluate to what extent this is also true for other languages, and the practice of analyzing erased bytecode instead of generic Java source code should be reexamined.

Our use of *static* types as contexts is distinct from the *dynamic* type tags used as contexts in the "type-sensitive" analysis of [Smaragdakis et al., 2011, 2014]. That analysis traces the flow of objects (abstracted by their dynamic type tags) from allocation sites along dataflow paths through the program all the way to each call site, and then analyzes the target of the call site in a separate context for each possible dynamic type of the receiver (and optionally of the other arguments [Agesen, 1995]). In contrast, the context that we propose is formed from the *static* types of the receiver and arguments that are available locally at the call site. Unlike dynamic type tags, the static type does not need to be propagated from the allocation site to the call site. Moreover, a given call site may be reached by objects of many different runtime types, which gives rise to many contexts for the target method in the "type-sensitive" analysis. In contrast, only a single static type argument is passed for each type parameter, so the number of contexts in our proposed analysis remains small.

This Chapter makes the following contributions:

— The Chapter proposes two extensions to the Scala call graph construction algorithm of [Ali et al., 2014]. In the first extension, we define the contexts in which a method is analyzed using the actual (but static) type arguments that are substituted for the generic type parameters of the method. In the second extension, we further refine the contexts by replacing the declared

types of the method's term parameters with more precise subtypes. Different combinations of choices of possible subtypes define distinct contexts. This form of context sensitivity is similar to that used in Agesen's Cartesian Product Algorithm [Agesen, 1995]. In the case of type class instances passed using Scala's implicit mechanism, our analysis can often specialize the parameter type to a singleton type that represents one specific instance of the type class.

— The chapter presents experimental results showing that

- the context-sensitive analyses are around two times *faster* than a context-insensitive analysis on substantial programs;
- the context-sensitive analyses discover significantly more monomorphic call sites; and
- the precision due to context-sensitivity reduces the number of times that the average method would have to be specialized from 22 to a much more reasonable 1.00 to 1.50 times.

The rest of the chapter is organized as follows.

- In Section 5.2, we present an example program that motivates the need for specialization and therefore for precise call graphs.
- In Section 5.3, we provide a background discussion of the $\text{TCA}^{\textit{expand-this}}$ analysis of [Ali et al., 2014], on which our extensions are formulated.
- We define our context-sensitive analyses in Section 5.4.
- Section 5.5 presents and discusses our experimental results.
- We discuss related work in Section 5.6, and
- We conclude in Section 5.7.

5.2 Motivation

```
715 implicit def Iterable[T](implicit ord: Ordering[T]): Ordering[Iterable[T]] =
716  new Ordering[Iterable[T]] {
717   def compare(x: Iterable[T], y: Iterable[T]): Int = {
718    val xe = x.iterator
719    val ye = y.iterator
720
721    while (xe.hasNext && ye.hasNext) {
722      val res = ord.compare(xe.next(), ye.next())
723      if (res != 0) return res
724    }
725
726    Boolean.compare(xe.hasNext, ye.hasNext)
727  }
728 }
```

Listing 5.1 – Running example from `scala.math.Ordering`.

We will motivate the need for a more precise call graph abstraction using the example method in Listing 5.1. This method is taken from the `scala.math.Ordering` class in the Scala standard library. Given any ordering `ord` for the type `T`, the method implicitly generates a lexicographic ordering for the type `Iterable[T]`. Since the `compare` method on Line 717 is called many times at run time, in loops, it is beneficial to specialize and inline the call sites within it as much as possible, especially those within the `while` loop on Line 721. In particular, a high-performance code generator should specialize the `compare` method for each value `ord` for which it is generated.

A context-insensitive call graph will contain a path to the `compare` method on Line 717 from the `Arrays.sort` method in the Java standard library. Therefore, for every type `T` that is ever sorted anywhere in the whole program, a sound analysis should find that an object of every such type could reach the parameters `x` and `y` of `compare`. In particular, in a large program, this is likely to include most of the possible subtypes of `Iterable`. In the Scala standard library, the trait `Iterable` has 214 concrete subtypes.

As a result, the calls to `x.iterator` and `y.iterator` on lines 718 and 719 will be highly polymorphic and not inlineable.

As a consequence, the sets of possible types of `xe` and `ye` will be highly imprecise. There are 44 concrete subtypes of `Iterator` in the Scala standard library.

Therefore, the calls to `xe.hasNext` and `ye.hasNext` on Line 721 will also be highly polymorphic and infeasible to inline; this is also true for calls to `xe.next()` and `ye.next()` on Line 722. The bodies of these four methods are usually small, and are called for every element of the iterables; therefore they need to be inlined to achieve good performance.

Finally, the call to `ord.compare` on Line 722 is statically considered to be dispatched to every implementation of `Ordering[T]` that reaches the `ord` parameter. Therefore, this call is also

```
732 def lexicographicSort[T](a: Seq[Iterable[T]])(implicit o: Ordering[T]) = a.sorted
733
734 lexicographicSort[Char](
735   Predef.wrapRefArray[WrappedString](
736     new Array(
737       Predef.wrapString("world"),
738       Predef.wrapString("Hello")
739     )
740   )
741 )(Ordering.Char)
```

Listing 5.3 – Desugared version of example program from Listing 5.2.

highly polymorphic in a context-insensitive call graph.

Let us consider how the static polymorphism could be reduced using context sensitivity (or, equivalently, specialization). We will illustrate this with the example client program in Listing 5.2.

```
729 def lexicographicSort[T](a: Iterable[T]*)(implicit o: Ordering[T]) = a.sorted
730
731 lexicographicSort("world", "Hello")
```

Listing 5.2 – Example program that uses the `compare` method from Listing 5.1.
The snippet defines a generic method `lexicographicSort` that creates a sorted list of values of type `Iterable[T]` by calling the `sorted` method of `SeqLike`. The `*` after the `Iterable[T]` parameter type indicates that the method takes a variable number of parameters, each of type `Iterable[T]`. The `lexicographicSort` method is called with two strings on Line 731.

Type inference and implicit resolution in the early stages of the Scala compiler desugar the program as shown in Listing 5.3.

One of the most serious impediments to good performance of the `compare` method is the need to box and unbox values of primitive Java types such as `char`. The bytecode version of the `Iterator.next` method has a return type of `Object`. This is incompatible with primitive types, so each `char` that it returns must be boxed in a `Character`. Inside the `compare` method of `Ordering.Char`, the `Character` must again be unboxed into a primitive `Char`.

Our first proposed improvement to the call graph is to analyze the entire outer `Iterable` method from Listing 5.1 separately in the context of each possible type argument with which the type parameter `T` is instantiated. In this example, `T` is specialized to `Char`. As a result, the type of `xe` and `ye` becomes `Iterator[Char]`, and the calls to `xe.next()` and `ye.next()` in Line 722 can be redirected to versions of the methods that return a primitive `Char` without boxing. Similarly, the type of `ord` becomes `Ordering[Char]`, so the call of `ord.compare` can be redirected to a version with primitive `Char` parameters that do not need to be unboxed. Thus, all of the boxing and unboxing can be removed from the `while` loop.

Our second proposed improvement is to analyze methods separately in the contexts of the

more precise types of their parameters that are available at the call site. In our running example, we can determine that when `T` is `Char`, the `compare` method is only called with a small number of concrete types of `Iterables`. In particular, we can analyze `compare` specifically in the context in which both of its parameters are of the type `WrappedString`, that is returned by `Predef.wrapString`. The calls to `x.iterator` and `y.iterator` in Lines 718 and 719 become monomorphic, which enables the analysis to give a precise type to `xe` and `ye`. As a result, the calls to `hasNext` and `next()` become monomorphic as well. We can now rewrite the known monomorphic calls to target specific statically known versions of their target methods, which makes it easy for the Java JIT compiler to inline and aggressively optimize them. The resulting optimized code is a simple loop over the arrays underlying the implementations of the strings that are being compared, much like the typical loop that one would write in C to compare two strings.

5.3 Background

The existing state of the art in call graph construction for Scala is the $\text{TCA}^{\textit{expand-this}}$ algorithm of [Ali et al., 2014, 2015]. To enable comparison of our results with previous work, we formulate our improvements as extensions to this existing framework. In this section, we present this baseline framework.

The main inference rules of the formulation are shown in Figure 5.1. The algorithm iterates the rules until a fixed point is reached, using worklists to keep track of new facts and to determine which rules need to be reevaluated. The set R keeps track of the methods reachable from the entry points through the call graph constructed so far. The set $\hat{\Sigma}$ keeps track of the types of objects that may be allocated in these reachable methods. The rule $\text{TCA}^{\textit{expand-this}}_{\text{MAIN}}$ initializes R with the main entry point. The rule $\text{TCA}^{\textit{expand-this}}_{\text{NEW}}$ finds object instantiations in reachable methods and adds the types to $\hat{\Sigma}$. The rule $\text{TCA}^{\textit{expand-this}}_{\text{CALL}}$ resolves a call site $e.m(\ldots)$ using the static type of the receiver e to determine all possible target methods M'. The rule $\text{TCA}^{\textit{expand-this}}_{\text{ABSTRACT-CALL}}$ handles the specific case of a call site at which the static type T of the receiver e is an abstract type. In this case, the $\text{TCA}^{\textit{expand-this}}$ algorithm uses the function *expand*() to determine the possible concrete types with which T could be instantiated. The *expand*() function is computed by additional inference rules that find all the concrete types with which the abstract type T could ever be instantiated. We do not show those rules here; for details, refer to [Ali et al., 2014, 2015]. The rule $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$ is a variation of $\text{TCA}^{\textit{expand-this}}_{\text{CALL}}$ that is more precise in the specific case when the receiver of the call is the `this` pointer in the caller (i.e. the receiver of the callee is the same object as the receiver of the caller). In this case, the rule adds precision by using the additional precondition that the caller M must also be a member of some type C that the callee M' is a member of. The rule $\text{TCA}^{\textit{expand-this}}_{\text{LOCAL-CALL}}$ handles calls to local functions that are nested inside some other function rather than being members of a class. This rule was not given explicitly by [Ali et al., 2014, 2015], but we have added it here for completeness. Calls to such functions do not have a receiver, and they are not dispatched dynamically: the method specified at the call site is the exact method that is executed.

$$\text{TCA}^{\textit{expand-this}}_{\text{MAIN}}\ \frac{}{\textit{main} \in R} \qquad \text{TCA}^{\textit{expand-this}}_{\text{NEW}}\ \frac{\text{“new } C()\text{” occurs in } M \quad M \in R}{C \in \hat{\Sigma}}$$

$$\text{TCA}^{\textit{expand-this}}_{\text{CALL}}\ \frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ C \in \textit{SubTypes}(\textit{StaticType}(e)) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{M' \in R}$$

$$\text{TCA}^{\textit{expand-this}}_{\text{ABSTRACT-CALL}}\ \frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ \textit{StaticType}(e) \text{ is an abstract type } T \\ C \in \textit{SubTypes}(\textit{expand}(\text{T})) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{M' \in R}$$

$$\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}\ \frac{\begin{array}{c}\text{call } D.\textit{this}.m(\ldots) \text{ occurs in method } M \\ D \text{ is the declaring trait of } M \\ C \in \textit{SubTypes}(D) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ \text{method } M \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{M' \in R}$$

$$\text{TCA}^{\textit{expand-this}}_{\text{LOCAL-CALL}}\ \frac{\begin{array}{c}\text{call } M'(\ldots) \text{ occurs in method } M \\ M' \text{ is method nested inside method } M'' \\ M \in R\end{array}}{M' \in R}$$

Figure 5.1 – Inference rules of $\text{TCA}^{\textit{expand-this}}$ from [Ali et al., 2014, 2015]

5.4 Algorithms

5.4.1 TCA^{types}: Propagation of Type Arguments

We now introduce the first extension to the TCA algorithm. The main idea is to construct a context-sensitive call graph in which each context for a given method is a substitution of concrete types for the type parameters of that method. Specifically, the elements of the set R, which were the reachable methods in TCA, now become pairs consisting of a reachable method and a type substitution. The inference rules for the extended algorithm are shown in Figure 5.2. Changes from the original algorithm are shaded.

The rule $\text{TCA}^{types}_{\text{MAIN}}$ pairs the main method with the empty substitution $\emptyset$, since the entry point of the program has no type parameters.

The rule $\text{TCA}^{types}_{\text{NEW}}$ iterates over all reachable method-substitution pairs, ignores the substitution, and adds the types instantiated in each reachable method to $\hat{\Sigma}$, as in the original algorithm.

In the rule $\text{TCA}^{types}_{\text{CALL}}$, for each reachable pair (M, σ), where M is a method and σ is a substitution, σ is applied to the static type of the receiver e. We use the postfix notation $\mathit{StaticType}(e)\sigma$ to denote substitution application. From the actual type arguments passed to the callee M' at the call site, we define the substitution σ' that replaces each type parameter of M' with the argument that is passed for it. In the conclusion of the $\text{TCA}^{types}_{\text{CALL}}$ rule, the caller's context substitution σ is composed with the call site substitution σ'. As a result, if σ' uses one of the type parameters of the caller, it will be replaced, using σ, with the concrete type that it is instantiated with in the specific caller context. We use the notation $\sigma'\sigma$ to denote substitution composition. We restrict the resulting composed substitution to only the type parameters of M', formally $\text{dom}(\sigma')$. The notation $\sigma'\sigma|_{\text{dom}(\sigma')}$ will denote this restriction.

We apply similar modifications to the rules $\text{TCA}^{expand\text{-}this}_{\text{THIS-CALL}}$ and $\text{TCA}^{expand\text{-}this}_{\text{ABSTRACT-CALL}}$ to obtain the new rules $\text{TCA}^{types}_{\text{THIS-CALL}}$ and $\text{TCA}^{types}_{\text{ABSTRACT-CALL}}$.

Because the set of possible types is unbounded, the set of reachable methods paired with type substitutions could grow without bound. In particular, this happens in the case of polymorphic recursion in the following example:

```
742 def foo[A](a: List[A], d: Int): List[_] =
743     if (d == 0) a
744     else foo(a.zip(a), d - 1)
```

The method `foo` in context $[\texttt{A} \mapsto \texttt{Int}]$ calls `foo` in context $[\texttt{A} \mapsto \texttt{(Int, Int)}]$, which later calls `foo` in context $[\texttt{A} \mapsto \texttt{((Int, Int), (Int, Int))}]$, and so on. To ensure the termination of call graph construction, we define a limit for the number of contexts under which each method is considered. If this limit is exceeded, then instead of creating a new context $(M, [\texttt{N}_i \mapsto \texttt{T}_i])$, we loosen

$$\text{TCA}^{types}_{\text{MAIN}}\ \frac{}{(main, \emptyset) \in R} \qquad \text{TCA}^{types}_{\text{NEW}}\ \frac{\text{“new } C()\text{” occurs in } M \quad (M, \ldots) \in R}{C \in \hat{\Sigma}}$$

$$\text{TCA}^{types}_{\text{CALL}}\ \frac{\begin{array}{c}\text{call } e.m\ [\sigma'](\ldots) \text{ occurs in method } M \\ C \in \mathit{SubTypes}(\mathit{StaticType}(\mathrm{e})\sigma) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ (M, \sigma) \in R \qquad C \in \hat{\Sigma}\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}) \in R}$$

$$\text{TCA}^{types}_{\text{ABSTRACT-CALL}}\ \frac{\begin{array}{c}\text{call } e.m\ [\sigma'](\ldots) \text{ occurs in method } M \\ \mathit{StaticType}(\mathrm{e})\sigma \text{ is an abstract type } T \\ C \in \mathit{SubTypes}(\mathit{expand}(\mathrm{T})) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ (M, \sigma) \in R \qquad C \in \hat{\Sigma}\end{array}}{(M, \sigma'\sigma|_{\text{dom}(\sigma')}) \in R}$$

$$\text{TCA}^{types}_{\text{THIS-CALL}}\ \frac{\begin{array}{c}\text{call } D.\mathit{this}.m\ [\sigma'](\ldots) \text{ occurs in method } M \\ D \text{ is the declaring trait of } M \\ C \in \mathit{SubTypes}(D) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ \text{method } M \text{ is a member of type } C \\ (M, \sigma) \in R \qquad C \in \hat{\Sigma}\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}) \in R}$$

$$\text{TCA}^{types}_{\text{LOCAL-CALL}}\ \frac{\begin{array}{c}\text{call } M'\ [\sigma'](\ldots) \text{ occurs in method } M \\ M' \text{ is a method nested inside method } M'' \\ (M, \sigma) \in R\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}) \in R}$$

Figure 5.2 – Propagation of type arguments

the precision of the last created context for the same method $(M, [\mathtt{N}_i \mapsto \mathtt{T}'_i])$ by replacing each type it contains with the least upper bound of the type in the old context and the type in the new context: $(M, [\mathtt{N}_i \mapsto lub(\mathtt{T}_i, \mathtt{T}'_i)])$. The loosened context conservatively over approximates the types in both the old, last created context for the method and the new context that we intended to create.

We did not encounter any cases of such unbounded growth in any of the benchmark programs that we evaluated.

5.4.2 Propagation of Outer Type Parameters

In the previous section, the context of each method substituted concrete types only for the direct type parameters of that method. For even greater precision, we can extend the context with the type parameters of the classes and methods that the method is nested within. Specifically, in our implementation, each element of $\hat{\Sigma}$ is not just an instantiated type C, but a pair (σ, C). Here, σ is a substitution that assigns a concrete type to every type parameter that is in scope at the program location where C is instantiated.

An equivalent method to achieve the same precision is to split the analysis into two phases. The first phase transforms the code using a transformation similar to lambda lifting [Johnsson, 1985], but applied to type parameters. Specifically, whenever a class or method has some type parameter `T` that can be implicitly used in methods nested within it, we add `T` as an explicit type parameter to each of those nested methods, and pass it explicitly at every call site. The second phase is then to perform the simple analysis described in the previous section. For performance reasons, our implementation uses the first approach of associating a substitution with each instantiated type. In the interest of clarity of presentation, our description in this paper follows the second approach, which decouples the issue of instantiating parameters of enclosing classes and methods from the analysis itself.

We illustrate the transformation with the following example program, in which method `bar` is nested in method `foo`, which itself is nested in class `C`:

```
745 class C[T] {
746  def foo[U](t: T, u: U) = {
747   def bar[V](t: T, u: U, v: V) = {...}
748
749   bar[Double](t, u, 1.0)
750  }
751 }
752 (new C[Int]).foo[String](5, "")
```

The above program would be transformed as follows:

```
753 class C[T] {
754  def foo[T2, U](t: T2, u: U) = {
755   def bar[T3, U2, V](t: T3, u: U2, v: V) = {...}
756
757   bar[T2,U,Double](t, u, 1.0)
758  }
759 }
760 (new C[Int]).foo[Int,String](5, "")
```

The type parameter `T` of class `C` has been explicitly added to the methods `foo` and `bar` nested within it as `T2` and `T3`. The type parameter `U` of method `foo` has been explicitly added to the method `bar` that is nested within it as `U2`.

Type parameters need to be passed explicitly when an outer method calls an inner one. When a given type parameter comes from a method in the original program, it is available at the call site as an explicit parameter of the caller method in the transformed program: for example, in the call of `bar` from `foo`, type parameters `T2` and `U` of `foo` are passed as arguments for the parameters `T3` and `U2` of `bar`. When a given type parameter comes from a class in the original program, it is also available at the call site as an argument in the type of the receiver: for example, in the call to `foo`, the type argument `Int` in the type `C[Int]` of the receiver determines the type argument to be passed for the parameter `T2` of `foo`.

Note that the erasure of both the original and the transformed program is the same; therefore the runtime behavior is left unchanged.

In addition to type parameters, we also transform the abstract type members of each class in the same way, turning them into explicit type parameters of all methods nested inside the class. Consider the following program:

```
761 abstract class Buffer {
762   type U
763   type T <: Seq[U]
764   def elements: T
765   def length = elements.length
766 }
767 class Buffer123 extends Buffer {
768   type U = Int
769   type T = List[Int]
770   def elements = List(1, 2, 3)
771 }
772
773 Buffer123.length()
```

The program is transformed to:

```
774 abstract class Buffer {
775   type U
776   type T <: Seq[U]
777   def elements[U2, T2 <: Seq[U2]]: T2
778   def length[U2, T2 <: Seq[U2]] =
779     elements[U2, T2].length
780 }
781 class Buffer123 extends Buffer {
782   type U = Int
783   type T = List[Int]
784   def elements[U2 = Int, T2 = List[U2]]: T2 =
785     List(1,2,3)
786 }
787
788 Buffer123.length[Buffer123.U, Buffer123.T]()
```

A consequence of this transformation is that the body of each method refers only to the type parameters defined on the method itself, and does not refer to any type parameters or type members of outer enclosing classes or methods. As a result, in the transformed program, the substitution context defined in the previous section now provides arguments for all the type parameters of each method. This includes those that came indirectly from outer classes and methods in the original program.

It is now easy to prove inductively that the range of every substitution σ that ever appears in a pair in R consists only of fully instantiated types (which do not contain any type parameters). Suppose that this is true of the substitution context σ of a method M that contains a call site $e.m[\sigma']()$. The only type variables used in the argument substitution σ' are the direct type parameters of M. The context substitution σ provides fully instantiated types for all of these type parameters. Therefore, when σ' and σ are composed, the range of the composed substitution contains only fully instantiated types. It is this composed substitution with fully instantiated types that becomes the new context for the target method called by the call site.

Therefore, the static type of the receiver of a call, $StaticType(e)\sigma$, is never abstract after the caller-context substitution σ has been applied to it. The rule $\text{TCA}^{types}_{\text{ABSTRACT-CALL}}$ is thus never needed and can be removed from the algorithm, together with the rules for computing the *expand*() sets for abstract types.

5.4.3 $\text{TCA}^{types\text{-}terms}$: Propagation of Term Argument Types

It is very common for the receiver at a call site to be one of the (term) parameters of the method containing the call site. The implicit receiver parameter `this` is the most common such receiver, but other parameters are common as well. As an example, consider the following code:

$$\mathrm{TCA}^{\textit{types-terms}}_{\textsc{main}}\ \frac{}{(\mathit{main}, \emptyset, \boxed{\mathit{Array[String]}}) \in R} \qquad \mathrm{TCA}^{\textit{types-terms}}_{\textsc{new}}\ \frac{\text{“new } C()\text{” occurs in } M \quad (M, \ldots, \boxed{\ldots}) \in R}{C \in \hat{\Sigma}}$$

$$\mathrm{TCA}^{\textit{types-terms}}_{\textsc{call}}\ \frac{\begin{array}{c}\text{call } e.m[\sigma'](\boxed{\overline{args}}) \text{ occurs in method } M \\ C \in \mathit{SubTypes}(\mathit{StaticType}(\boxed{\pi}, e)\sigma) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ (M, \sigma, \boxed{\pi}) \in R \qquad C \in \hat{\Sigma} \\ \boxed{\pi' = (e :: \overline{args}).\mathrm{map}(arg \Rightarrow \mathit{StaticType}(\pi, arg)\sigma)}\end{array}}{(M', \sigma'\sigma|_{\mathrm{dom}(\sigma')}, \boxed{\pi'}) \in R}$$

$$\mathrm{TCA}^{\textit{types-terms}}_{\textsc{local-call}}\ \frac{\begin{array}{c}\text{call } M'[\sigma'](\boxed{\overline{args}}) \text{ occurs in method } M \\ M' \text{ is a method nested inside method } M'' \\ (M, \sigma, \boxed{\pi}) \in R \\ \boxed{\pi' = \overline{args}.\mathrm{map}(arg \Rightarrow \mathit{StaticType}(\pi, arg)\sigma)}\end{array}}{(M', \sigma'\sigma|_{\mathrm{dom}(\sigma')}, \boxed{\pi'}) \in R}$$

$$\mathrm{TCA}^{\textit{types-terms}}_{\textsc{this-call}}\ \frac{\begin{array}{c}\text{call } D.\mathit{this}.m[\sigma'](\boxed{\overline{args}}) \text{ occurs in method } M \\ D \text{ is the declaring trait of } M \\ C \in \mathit{SubTypes}(D) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ \text{method } M \text{ is a member of type } C \\ (M, \sigma, \boxed{\pi}) \in R \qquad C \in \hat{\Sigma} \\ \boxed{\pi' = (D.\mathit{this} :: \overline{args}).\mathrm{map}(arg \Rightarrow \mathit{StaticType}(\pi, arg)\sigma)}\end{array}}{(M', \sigma'\sigma|_{\mathrm{dom}(\sigma')}, \boxed{\pi'}) \in R}$$

Figure 5.3 – Propagation of term argument types

```
789 def internalHashCode[T](el: T, nullRep: Object) =
790  if (el != null)
791   el.hashCode
792  else
793   nullRep.hashCode
794
795 internalHashCode[Int](42, "null")
```

The receivers `el` and `nullRep` of the calls to `hashCode` are both parameters of `internalHashCode`. When the type of the receiver is itself a type variable of the caller, the propagation of type arguments that we have described above helps to resolve the call precisely. In the example, the type of `el` is the type parameter `T`, which the context substitution instantiates to `Int`; consequently we know that the target of `el.hashCode` is the implementation of `hashCode` in `Int`. However, in the call `nullRep.hashCode`, we need to assume that the runtime type of the receiver `nullRep` may be any subtype of `Object`. To further improve precision, the analysis can be extended further to propagate the type of the argument from the call site of `internalHashCode`, which is `String`, into the context in which `internalHashCode` is analyzed. As a result, the analysis could then determine that the call `nullRep.hashCode` calls only the `String` implementation of `hashCode`.

To implement this precision improvement in our call graph construction algorithm, we further extend the method contexts contained in the set R. Each element of R becomes a triple that contains a reachable method M and a type parameter substitution σ as before, and, in addition, a list π of more precise types for the term parameters of M (including the implicit `this` receiver parameter).

The inference rules for the extended algorithm are shown in Figure 5.3. Changes from Figure 5.2 are shaded. The *StaticType* function is extended to take a list π of more precise parameter types. If e is a parameter of M, then $\mathit{StaticType}(\pi, e)$ returns the more precise type of e given by π; otherwise it just returns the same static type of e as in the previous analyses. We also extend *StaticType* to map over a sequence of terms and return a sequence of their types. The last premise of the $\text{TCA}^{\mathit{types}}_{\text{CALL}}$ rule uses *StaticType* to get the precise types of the arguments passed at the call site. The substitution σ is applied to these types. These precise types π' are then included in the context that is added to R at the conclusion of the rule.

5.5 Evaluation

We have implemented the $\text{TCA}^{\mathit{expand\text{-}this}}$ analysis of Ali et al. [2014, 2015] and our two extensions $\text{TCA}^{\mathit{types}}$ and $\text{TCA}^{\mathit{types\text{-}terms}}$ on top of the Dotty compiler[3], a new compiler for the future evolution of the Scala language. Although Dotty is not yet finished, it is not a research prototype: it is intended to eventually replace the current `nsc`, becoming the standard production-

[3] ↑https://github.com/lampepfl/dotty

Program	Algorithm	# Instantiated classes	# Classes with reachable method	# Reachable methods	# Reachable contexts	# Maximum contexts per method	# Discovered specializations	Code growth factor
List creation	TCA$^{expand\text{-}this}$	149	64	207	207	1	3469	16.75
	TCA^{types}	117	33	90	90	1	90	1.00
	TCA$^{types\text{-}terms}$	117	31	83	101	2	83	1.002
List & Vector creation	TCA$^{expand\text{-}this}$	152	79	268	268	1	6358	24.73
	TCA^{types}	130	36	95	114	2	114	1.20
	TCA$^{types\text{-}terms}$	130	34	90	138	4	112	1.24
List create and sort	TCA$^{expand\text{-}this}$	157	65	209	209	1	3919	18.75
	TCA^{types}	126	34	92	92	1	92	1.00
	TCA$^{types\text{-}terms}$	126	34	89	147	2	89	1.00
List & Vector create and sort	TCA$^{expand\text{-}this}$	170	83	357	357	1	7725	21.64
	TCA^{types}	142	39	115	140	2	140	1.21
	TCA$^{types\text{-}terms}$	142	37	109	147	5	131	1.20
List create, sort and print	TCA$^{expand\text{-}this}$	171	68	212	212	1	4146	19.56
	TCA^{types}	131	37	95	95	1	95	1.00
	TCA$^{types\text{-}terms}$	131	35	92	206	6	92	1.00
lexicographic Sort	TCA$^{expand\text{-}this}$	182	88	293	293	1	5529	18.87
	TCA^{types}	134	41	102	104	2	104	1.01
	TCA$^{types\text{-}terms}$	134	41	98	231	3	102	1.04
Page rank	TCA$^{expand\text{-}this}$	229	92	341	341	1	12490	36.63
	TCA^{types}	145	50	127	173	3	173	1.36
	TCA$^{types\text{-}terms}$	145	45	118	293	5	165	1.40
Round robin	TCA$^{expand\text{-}this}$	189	76	252	252	1	6272	24.89
	TCA^{types}	147	46	130	174	1	174	1.34
	TCA$^{types\text{-}terms}$	147	44	123	310	3	165	1.34
Dotty type-checker	TCA$^{expand\text{-}this}$	1028	822	10694	10694	1	45278	4.23
	TCA^{types}	832	695	9347	14011	4	14011	1.50
	TCA$^{types\text{-}terms}$	832	629	8992	37992	43	13122	1.46

Table 5.1 – Results of the TCA$^{expand\text{-}this}$, TCA^{types}, and TCA$^{types\text{-}terms}$ analyses on the benchmark programs. The first two columns specify the benchmark program and the analysis algorithm. The next three columns show the number of classes found to be instantiated, including their superclasses, classes that have at least one reachable method, and methods reachable by the analysis. The following two columns show the total number of reachable method contexts and the maximum number of such contexts per method. If every reachable method were specialized for all of the type arguments that the analysis determines may flow to its type parameters, the next two columns show the total number of such specialized methods that would be created and the factor by which this number is greater than the number of reachable methods in the original program.

Program	Algorithm	% monomorphic call sites	% bimorphic call sites	% megamorphic call sites	Running time, seconds
List creation	TCA$^{expand\text{-}this}$	80.2	7.0	12.8	0.76
	TCA^{types}	93.0	4.7	2.3	1.30
	TCA$^{types\text{-}terms}$	95.4	2.3	2.3	1.32
List & Vector creation	TCA$^{expand\text{-}this}$	73.2	4.1	22.3	1.89
	TCA^{types}	86.0	2.1	11.9	1.58
	TCA$^{types\text{-}terms}$	88.1	4.5	7.5	1.41
List create and sort	TCA$^{expand\text{-}this}$	77.6	6.4	16.0	0.77
	TCA^{types}	87.2	9.6	3.2	1.54
	TCA$^{types\text{-}terms}$	89.4	8.5	2.1	1.58
List & Vector create and sort	TCA$^{expand\text{-}this}$	72.2	2.4	25.2	2.30
	TCA^{types}	85.5	3.8	9.7	1.64
	TCA$^{types\text{-}terms}$	89.2	2.6	8.2	1.47
List create, sort and print	TCA$^{expand\text{-}this}$	78.6	4.1	17.4	1.29
	TCA^{types}	87.8	9.2	3.1	5.43
	TCA$^{types\text{-}terms}$	89.8	8.2	2.0	3.25
lexicographic Sort	TCA$^{expand\text{-}this}$	72.7	2.8	24.5	1.50
	TCA^{types}	85.8	7.6	5.6	5.91
	TCA$^{types\text{-}terms}$	89.1	6.5	4.40	4.08
Page rank	TCA$^{expand\text{-}this}$	59.4	8.5	32.1	10.28
	TCA^{types}	77.4	7.6	15.1	11.22
	TCA$^{types\text{-}terms}$	85.9	9.9	4.3	6.24
Round robin	TCA$^{expand\text{-}this}$	72.6	8.1	19.3	9.69
	TCA^{types}	87.9	8.1	4.0	8.79
	TCA$^{types\text{-}terms}$	87.9	8.9	3.2	3.91
Dotty type-checker	TCA$^{expand\text{-}this}$	55.6	1.8	42.6	893.52
	TCA^{types}	82.3	0.6	17.1	1071.71
	TCA$^{types\text{-}terms}$	90.7	2.6	6.7	637.10

Table 5.2 – Results of the TCA$^{expand\text{-}this}$, TCA^{types}, and TCA$^{types\text{-}terms}$ analyses on the benchmark programs. The next three columns show the percentage of call sites found to be monomorphic, bimorphic, and megamorphic by each analysis. For consistency, to enable comparisons between the three analyses, we take as the universe of all call sites only those in methods found to be reachable by the most precise analysis, TCA$^{types\text{-}terms}$. Otherwise, the results would be confounded by the fact that each analysis discovers a different set of reachable methods and therefore a different set of reachable call sites. The final column gives the running time of the analysis.

quality compiler for Scala. We tested our implementation on the full test suite of Dotty, which includes 1403 Scala programs. To the best of our knowledge, our analyses soundly handle the entire Scala language dialect supported by Dotty, including Dotty-specific extensions to Scala such as trait parameters[4] and repeated by name parameters[5].

The analysis runs after the type checker stage of Dotty. At this stage, all expressions have their original, unerased and unsimplified Scala types. This means that our implementation correctly handles types that may contain generic types and path dependent types [Odersky, 2014, §3.5]. When the analysis requires subtyping checks, we use the implementation of subtype testing included in the Dotty compiler.

In this section, we first evaluate the $\text{TCA}^{\textit{types-terms}}$ analysis implemented in Dotty, and then show how it can be used for program performance.

5.5.1 Analysis Evaluation

We have evaluated our implementation on the nine Scala programs listed in Tables 5.1 and 5.2. The first six programs were selected to exercise the Scala collections library, which is implemented in a very generic style with multiple layers of abstraction. The collections library is also highly megamorphic: for example, it contains 214 named subclasses of `Iterable`. The next two benchmarks are moderately sized applications implemented in idiomatic Scala. The largest benchmark is the parser and type checker of the Dotty compiler itself. The Dotty compiler is still under development, and only recently became able to bootstrap itself. Further development of the Dotty compiler is necessary before it can compile more mainstream Scala applications.

To construct each call graph, we provided all of the dependencies written in Scala as source code to the analysis. All Scala programs also implicitly depend on the Java Standard Library, which is in the form of Java bytecode that our implementation does not analyze. We made conservative assumptions about the effects of the Java library, and used the Separate Compilation Assumption [Ali and Lhoták, 2012; Ali and Lhoták, 2013] to construct a sound partial call graph for the parts of the program that were written in Scala and therefore available for analysis. The only methods of the Java standard library called by any of our benchmark programs and their Scala dependencies are the methods of the `java.lang.Object` and `java.lang.Comparable` classes.

We ran all of our experiments on a machine with a quad core 2.8 GHz Intel i7-4980HQ CPU (running in 64-bit mode) and capped available memory for experiments to 768 MB of RAM.

[4] ↑http://docs.scala-lang.org/sips/pending/trait-parameters.html
[5] ↑http://docs.scala-lang.org/sips/pending/repeated-byname.html

Research Questions

Our evaluation aims to answer the following Research Questions:

RQ1. How do the three analysis algorithms compare in regard to the precision of the call graphs that they generate?

RQ2. Type and term argument propagation increase the size of the set R by tracking methods multiple times with different type and term arguments. How severe is the increase?

RQ3. How usable are the call graphs generated by the three analysis algorithms for the purposes of specialization and inlining?

RQ4. How many call sites can the algorithms prove to be monomorphic?

RQ5. How does tracking of type and term arguments affect the running time of the analysis?

Results

RQ1. Relative to $\text{TCA}^{\textit{expand-this}}$, call graphs constructed by $\text{TCA}^{\textit{types}}$ have 22% fewer reachable classes and 56% fewer reachable methods on average. The most significant cause of the precision improvement was that $\text{TCA}^{\textit{types}}$ precisely resolved calls on generic super classes where $\text{TCA}^{\textit{expand-this}}$ was imprecise. For example, while a call on a `Seq[T]` could dispatch to both `List[Int]` and `Vector[Double]` according to $\text{TCA}^{\textit{expand-this}}$, $\text{TCA}^{\textit{types}}$ would analyze the call separately within the context of the two different type arguments.

On the Dotty typechecker, the $\text{TCA}^{\textit{types}}$ call graph has 15 % fewer reachable methods than the $\text{TCA}^{\textit{expand-this}}$ call graph. The improvement is smaller because Dotty makes little use of the generic collections in the standard library. For example, Dotty uses its own custom-tuned implementations of sets. Of 629 classes with reachable methods, only 40 are from the standard library.

On average over all of the benchmark programs, the analysis $\text{TCA}^{\textit{types-terms}}$ further reduces the number of reachable methods by 5% compared to $\text{TCA}^{\textit{types}}$.

The number of megamorphic call sites is, on average, 70% lower with $\text{TCA}^{\textit{types}}$ than with $\text{TCA}^{\textit{expand-this}}$. $\text{TCA}^{\textit{types-terms}}$ further reduces the number of megamorphic call sites to 32% fewer than $\text{TCA}^{\textit{types}}$.

On the Dotty type checker, $\text{TCA}^{\textit{types-terms}}$ reduces the number of megamorphic call sites by

60% compared to TCA^{types}. The main source of this improvement is `apply` methods, which implement closures.

RQ2. We might expect that the number of reachable contexts would grow as the amount of context sensitivity is increased. In fact, due to the substantial improvement in precision and the decrease in the number of reachable methods, the average number of reachable contexts is 53% *smaller* in TCA^{types} than in $\text{TCA}^{expand\text{-}this}$. $\text{TCA}^{types\text{-}terms}$ does generate more reachable contexts than TCA^{types}, but, in general, still fewer than $\text{TCA}^{expand\text{-}this}$.

The Dotty typechecker is a special case in this regard. It has a substantial number of closures that are passed as arguments, with multiple different closures being passed to the same method. Tracking all of these closures requires four times as many reachable method contexts in $\text{TCA}^{types\text{-}terms}$ as there are reachable methods in $\text{TCA}^{expand\text{-}this}$.

As we mentioned in Section 5.4.1, it is theoretically possible for the number of contexts to grow without bound, and we must stop generating new contexts after a fixed limit has been exceeded in order for the analysis to terminate. We did not observe unbounded growth in any of the benchmark programs. To determine how to select the limit, we counted the maximum number of contexts for any given reachable method for each benchmark. The maximum number of contexts was six or less for all of the benchmarks, except for the special case of the Dotty typechecker. This contains a function `track(String)(Closure)` that is used to track the number of times a particular computation is performed. This function is called with 43 different closures, and term argument type propagation tracks all of them as separate contexts. Aside from this function, only five other functions in the Dotty typechecker are analyzed with more than 10 contexts.

RQ3. The call graphs generated by the three algorithms provide information about the concrete type arguments with which each type parameter in the program can be instantiated. Our intended application is to specialize each generic method for each of the type arguments that it may be called with. Each method that has been specialized in this way can be easily inlined as an additional step, either in a static optimizer or in a JIT compiler.

The type argument information provided by the context-insensitive $\text{TCA}^{expand\text{-}this}$ analysis is too imprecise to be practical for this application. It indicates that each method should be specialized 22 times on average.

Both of the context-sensitive analyses, TCA^{types} and $\text{TCA}^{types\text{-}terms}$, provide much more usable information for specialization. They indicate that, on average, methods need to be specialized 1.50 times.

RQ4. Our intended applications of call graphs, specialization and inlining, apply to call sites that have only a single possible target method (are monomorphic). The precision of

many other analyses such as points-to analysis and escape analysis benefits significantly from precise knowledge of the targets of virtual calls. We therefore measure the ability of different algorithms to resolve each call site to a unique target method.

Adding type propagation in TCA^{types} substantially increases the percentage of call sites that are statically monomorphic compared to $\text{TCA}^{expand\text{-}this}$, by around 10 percentage points on small programs and by around 20 percentage points on large programs. $\text{TCA}^{types\text{-}terms}$ further increases monomorphic call sites by up to eight percentage points on large programs.

RQ5. We might expect that the more precise context-sensitive analyses require more time than $\text{TCA}^{expand\text{-}this}$. This is indeed the case on some of the small programs that exercise the library: TCA^{types} takes up to four times as long as $\text{TCA}^{expand\text{-}this}$. This is due to more complex rules that require more work to process each call site. However, on the three larger programs, TCA^{types} takes on average only 20% more time than $\text{TCA}^{expand\text{-}this}$, and $\text{TCA}^{types\text{-}terms}$ is actually always *faster* than $\text{TCA}^{expand\text{-}this}$. This is explained by the more precise (and therefore smaller) sets R and $\hat{\Sigma}$ computed by the context-sensitive algorithms. A major source of the speedup of $\text{TCA}^{types\text{-}terms}$ over TCA^{types} is that the implementation of substituting a type for a type parameter that occurs inside a complicated type is slow. In many cases, term argument type propagation can copy the entire (already substituted type) faster than it would take to replace the type parameters within it.

5.5.2 Application to Specialization

The evaluation so far has focused on the output of the $\text{TCA}^{types\text{-}terms}$ analysis. In this section, we show how the analysis improves the effectiveness of specialization.

Generic classes and methods can be compiled to low-level code using two approaches. A heterogeneous approach duplicates the generic code and adapts it for every set of type arguments [Kennedy and Syme, 2001; Morrison et al., 1991]. This produces many low-level versions of a generic class or method, each adapted to efficiently handle a single type of data. A homogeneous approach generates a single copy with the type parameters erased to their upper bound, commonly `Object`, that can accommodate values of any type [Bracha et al., 1998].

Similar approaches have also been developed for functional languages with polymorphic types. Intentional type analysis [Harper and Morrisett, 1995] introduces user-facing syntax that is similar to runtime reflection that can be used to inspect types and generate specialized classes at run time. For functional programs requiring boxing, [Henglein and Jørgensen, 1994] introduces a rewriting algorithm that places the boxing and unboxing operations to minimize the number of coercions executed according to a formal optimality criterion.

Both approaches have benefits and drawbacks. Although the homogeneous approach minimizes the amount of generated low-level code, it has poor performance: each time a value of

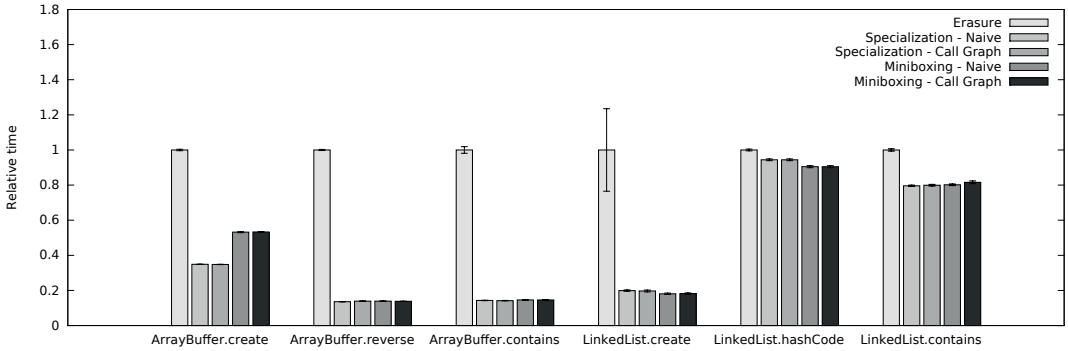


Figure 5.4 – Graphical representation of the data presented in Table 5.3, in milliseconds. Lower is better.

primitive type flows in to and out of generic code, it must be boxed into a freshly-allocated object and respectively unboxed back to its primitive type [Leroy, 1992]. The heterogeneous approach avoids boxing and unboxing, but it requires knowing the set of possible type arguments. Furthermore, the number of combinations of type arguments used to instantiate a generic class or method grows exponentially with the number of type parameters, making the heterogeneous approach impractical. Both Java and Scala use the homogeneous translation by default, despite its negative effect on performance.

Specialization is a technique that allows compiling selected classes and methods using the heterogeneous approach [Dragos, 2010; Dragos and Odersky, 2009; Goetz, 2014], while leaving the rest of the generic code to use the default homogeneous translation. In Scala, specialization allows the programmer to annotate the type parameter of a class or method as `@specialized`. Based on this annotation, the compiler generates 10 versions of the code, one for the universal `Object` type and one for each of the nine primitive Scala types. When the class or method has n type parameters annotated as `@specialized`, the compiler generates 10^n versions of the code. The compiler also allows a more fine-grained annotation to specialize a type parameter only for a specified subset of the primitive types. For example, the annotation `@specialized(Int)` would cause two versions of the code to be generated, one for primitive integers and the other for the universal `Object` type (in which all other primitive types can be encoded using boxing). To make use of these newly created code variants, the compiler rewrites each generic class instantiation and each generic method call to refer to the appropriate specialized version indicated by the type arguments.

Specialization produces significant speedups, sometimes in excess of 10x, because boxing and unboxing operations often end up in hot loops [Dragos, 2010; Dragos and Odersky, 2009]. However, the increase in code size quickly becomes impractical. For example, specializing a map data structure, which has two type parameters, generates 100 variants, which makes distribution infeasible. A function type with two arguments and one return value requires three type parameters, and therefore an unreasonable 1000 variants.

	`ArrayBuffer.append`		`ArrayBuffer.reverse`	
	time	speedup	time	speedup
Erasure	37.3 ± 0.1	1x	12.5 ± 0.1	1x
Specialization - Naive	13.0 ± 0.1	2.9x	1.7 ± 0.1	7.4x
Specialization - Call Graph	13.0 ± 0.1	2.9x	1.7 ± 0.1	7.4x
Miniboxing - Naive	19.9 ± 0.1	1.9x	1.7 ± 0.1	7.4
Miniboxing - Call Graph	19.9 ± 0.1	1.9x	1.7 ± 0.1	7.4x
	`ArrayBuffer.contains`		`LinkedList.contains`	
	time	speedup	time	speedup
Erasure	3108.0 ± 59.1	1x	2871.8 ± 19.2	1x
Specialization - Naive	445.8 ± 4.2	7.0x	2286.1 ± 11.6	1.3x
Specialization - Call Graph	442.8 ± 2.2	7.0x	2296.0 ± 15.8	1.3x
Miniboxing - Naive	453.4 ± 3.6	6.9x	2303.9 ± 13.7	1.2x
Miniboxing - Call Graph	457.2 ± 3.7	6.8x	2333.5 ± 24.8	1.2x
	`LinkedList` creation		`LinkedList.hashCode`	
	time	speedup	time	speedup
Erasure	171.2 ± 40.3	1x	17.0 ± 0.1	1x
Specialization - Naive	34.1 ± 0.7	5.0x	16.9 ± 0.1	1.0x
Specialization - Call Graph	33.7 ± 0.8	5.1x	16.9 ± 0.1	1.0x
Miniboxing - Naive	31.0 ± 0.6	5.5x	16.2 ± 0.1	1.0x
Miniboxing - Call Graph	31.1 ± 0.6	5.5x	16.2 ± 0.1	1.0x

Table 5.3 – Benchmark running time, for 3 million elements. The time is reported in milliseconds. Lower is better.

Miniboxing [Ureche et al., 2013] is an alternative heterogeneous approach that encodes multiple primitive values in a single (larger) slot, thus reducing the number of variants from 10^n to 3^n. Using miniboxing, the map data structure, `Map[Key, Value]`, requires only nine variants, while the two-argument function, `Function2[T1, T2, R]`, requires 27 variants. As we will see later, the TCA^{types} analysis can further reduce the number of variants generated by miniboxing.

The fundamental problem remains: both specialization and miniboxing trigger excessive bytecode growth, making them infeasible to use as the default compilation scheme for generics. To avoid this excessive bytecode growth, programmers must carefully choose which type parameters are to be specialized. Furthermore, they must decide the exact primitive types that each type parameter should be specialized for, as this can reduce the generated bytecode. These two decisions require deep knowledge of the entire code base, including dependent libraries and applications. Yet different applications use a library in different ways, and no specific set of annotations of a library is ideal for all applications that may use it. Additionally, when an annotation (or a primitive type within an annotation) is missing, it can significantly harm performance [Ureche et al., 2015]. Therefore, when good performance is required, programmers often err on the side of specializing for all primitive types, accepting the consequent large increases in bytecode size.

The TCA^{types} analysis solves this problem by inferring the specialization annotations automatically. In particular, the necessary information is, for each generic class or method, the set of type argument instantiations of its type parameters. This set is exactly the set of contexts explored by the TCA^{types} analysis. Note that this information is not generally obtainable from just a (context-insensitive) call graph. The automatic inference of the specialization annotations depends on the specific contexts that we have introduced in the TCA^{types} analysis.

When the TCA^{types} analysis is employed, the specialization annotations generated contain the exact primitives used in the code and nothing more, reducing the bytecode generated as much as possible while avoiding the boxing operations completely. In the case of miniboxing, the TCA^{types} analysis can indicate if any of the miniboxing encodings is redundant, again saving the creation of redundant heterogeneous variants.

Specialization guided by the TCA^{types} analysis results is fully correct in an open-world context. The specialization transformation does not depend on any soundness assumptions about the specialization annotations, which are normally provided by the programmer. If a type parameter is instantiated by a type argument that was not included in the annotation, the generated code falls back to the default universal `Object`-based implementation and its associated boxing and unboxing. Therefore, unanalyzed code that passes type arguments that the analysis is not aware of will still work correctly, although, understandably, it will not enjoy the same performance improvement as the analyzed code.

To test the effectiveness of our analyses, we have applied them to specialization and miniboxing, reproducing the performance experiments from the miniboxing paper [Ureche et al., 2013]. The benchmarks are adapted from two collection classes in the Scala standard library, `ArrayBuffer` and (linked) `List`, and selected to cover code patterns commonly used throughout the collection library. They cover a wide range of scenarios: both contiguous and sparse memory storage, custom equality checks, hash code computations, and tight loops that can be further optimized by the JIT compiler (e.g., `ArrayBuffer.reverse`). Each benchmark method is exercised by a driver program that executes it on collections of three million integers. The setup is similar to the one used in the miniboxing paper.

To evaluate the automated inference of specialization annotations, we used the following experimental setup: We first compiled the benchmark programs with the Dotty compiler and the TCA^{types} analysis. In general, the TCA$^{types\text{-}terms}$ analysis could be more precise, but on these benchmark programs, both analyses produce the same results. The type contexts found by the analysis were translated into specialization annotations inserted into the code. The annotated code was then compiled with the standard Scala compiler and evaluated for performance. We used the standard Scala compiler for this last step for consistency with the experiments in the miniboxing paper, and because the porting of the specialization transformations from the standard Scala compiler to Dotty is still in progress. Once the specialization feature is completely ported to Dotty, the overall process can be implemented in a single compilation pass that performs the analysis and applies the specializations.

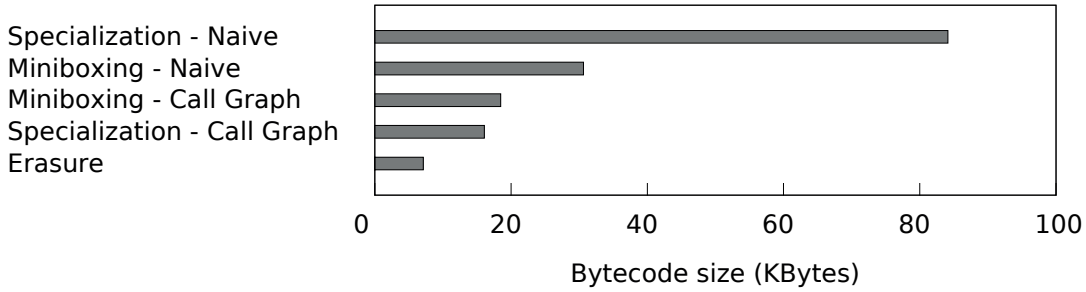


Figure 5.5 – Graphical representation of the data in Table 5.4, showing the bytecode size in kilobytes. Lower is better.

We ran the benchmarks on a server machine with an 8-core Intel i7-4770 processor with the frequency fixed at 3GHz, running the Oracle Java distribution 1.7.0-79 on the Ubuntu 12.04.5 LTS operating system. We used the JMH benchmarking framework [Shipilev, 2016] as a harness, due to its close integration with the OpenJDK execution platform: for each benchmark, JMH started the Java Virtual Machine (JVM) with 3GB of memory, warmed up the benchmark code until it was compiled by the HotSpot Just-in-time (JIT) C2 compiler, and then took 20 measurements. To minimize the noise, the process was repeated 10 times for each benchmark. This ensured that the variability introduced by the JIT compiler, the garbage collector (GC) and other processes running on the server was reduced as much as possible.

The performance results are shown in Table 5.3 and Figure 5.4. The "Erasure" results are for an unannotated program compiled using a homogeneous translation. The "Specialization - Naive" results simulate a fully heterogeneous translation by annotating every type parameter with `@specialize`, and using the implementation of the specialization transformation in the standard Scala compiler to generate clones of the methods. The "Specialization - Call Graph" results evaluate a program with annotations for specialization inferred by the TCA^{types} analysis, and specialized by the standard implementation in the Scala compiler. The same types of naive vs call-graph-based annotations are shown for the "Miniboxing" transformation.

Transformation	Bytecode Size (Bytes)
Specialization - Naive	86146
Miniboxing - Naive	31372
Miniboxing - Call Graph	18918
Specialization - Call Graph	16458
Erasure	7291

Table 5.4 – The bytecode size produced by specializing the ArrayBuffer and LinkedList classes with different approaches. Lower is better.

Although the last four compilation strategies achieve similar speedups over the baseline "Erasure" configuration, there is a stark difference in the size of the generated bytecode. The total

bytecode size for the two data structures is shown in Table 5.4. Figure 5.5 shows the same data graphically. The fully heterogeneous translation ("Specialization - Naive") requires a prohibitive 11.8x increase in the size of the code compared to the standard homogeneous translation. Miniboxing ("Miniboxing - Naive") reduces this overhead to a still substantial 4.3x. Using the TCA^{types} analysis to drive the two heterogeneous transformations produces the same performance while further reducing the bytecode size by 5.2x for specialization and 1.7x for miniboxing (the "Specialization - Call Graph" and "Miniboxing - Call Graph" entries, compared with their "Naive" counterparts).

In fact, the code size increase can easily be reduced even further by a tighter integration of the analysis and the specialization transformation. In the current implementation of specialization, if two or more type parameters are annotated, the compiler generates specialized versions of the code for the outer product of the possible argument types. For example, if the keys and values of a map can each be of type `Int` or `Long`, the compiler generates all four combinations. However, the analysis could have more precise information that indicates, for example, that only `Map[Int,Int]` and `Map[Long,Long]` are ever instantiated. Using this information, the specialization transformation would generate only two versions instead of four. However, the current annotation mechanism is not expressive enough to encode this precise information that the analysis provides.

5.6 Related Work

We survey two separate areas of related work. First, we discuss the main intended application of our analysis: specialization techniques that have been proposed for Scala and similar languages. Second, we discuss context sensitivity in call graph construction in general, in various programming languages, and compare our analysis to other related analyses.

5.6.1 Specialization Techniques

In the context of generating efficient Java bytecode from Scala programs, [Dragos, 2010] observes that "compilation of polymorphic code through type erasure gives compact code but performance on primitive types is significantly hurt". Consider the following method `foo`:

```
796 def foo[A](a: A) = a
797
798 foo[Int](1)
```

This code is compiled as follows

```
799 def foo(a: Object) = a
800
801 foo(new Integer(1)).asInstanceOf[Integer].value
```

Dragos proposes a specialization technique for Scala that requires the programmer to mark

methods to be specialized. The compiler generates specialized versions of each such method for each primitive type. If such a `@specialized` annotation were applied to the `foo` method in our example, the compiler would generate the following code:

```
802 def foo(a: Object) = a
803 def foo_i(a: Int)  = a // synthetic clone
804
805 foo_i(1)
```

The implementation conservatively generates clones for all nine of the primitive types in Scala, as well as the reference type (erased to `Object`). For a method with n type parameters, 10^n clones are needed. This limits the use of specialization in Scala. For example, the standard library type `Function2` that represents a function with two parameters has three type parameters (one for the type of each parameter, and a third for the return type). Specializing `Function2` would require $10^3 = 1000$ clones, which is impractical.

Miniboxing [Ureche et al., 2013] is a technique that reduces the number of clones required from 10^n to 2^n. It encodes all primitive types into a single type, a 64-bit `long`, and uses a marker byte to indicate the original type. For each type parameter, only two clones are needed: one for primitive types (encoded as `long`), and one for reference types (encoded as `Object`). This approach makes it viable to mark as `@miniboxed` methods with up to six type parameters.

Wider use of miniboxing suggested that similar specialization techniques could harm performance if specialized code is called frequently from generic code and vice versa [Ureche et al., 2015]. Consider the following example:

```
806 def foo[A](a: A) = a
807 def bar[@miniboxed A](a: A) = while(true) foo(a)
808 def bar1[A](a: A) = while(true) foo(a)
```

In order to call the generic method `foo`, the specialized method `bar` will need to box `a` in every iteration. In contrast, the value `a` in the generic method `bar1` will already be boxed before `bar1` is called, so it will not have to be boxed again in every iteration of the loop. The miniboxing implementation tries to help users solve this problem by providing comprehensive warnings that suggest possible changes to the code [Ureche et al., 2015].

Similar techniques are available as part of the .Net runtime [Kennedy and Syme, 2001] and are under development for Java as part of Project Valhalla [Goetz, 2014].

5.6.2 Call Graph Construction and Context Sensitivity

Context sensitivity has been studied extensively in call graphs for dynamically typed functional languages [Shivers, 1988]. However, because of Scala's expressive static type system, call graph construction algorithms for statically-typed languages are more closely related. In object-oriented languages, call graph construction and points-to analysis are interdependent,

because virtual calls are resolved using the runtime type of the receiver object pointed to by the call site.

For Java, the most thoroughly studied forms of context are call strings [Shivers, 1988] and object sensitivity [Milanova et al., 2002, 2005]. Analyses using these forms of context sensitivity have a high cost, and much work has been done to balance the analysis cost against the precision of the analysis results [Bravenboer and Smaragdakis, 2009; Kastrinis and Smaragdakis, 2013; Smaragdakis et al., 2011, 2014; Sridharan and Bodík, 2006; Xu and Rountev, 2008; Xu et al., 2009; Yan et al., 2011]. In Java, context sensitivity has been found to improve the precision of pointer information. Its effect on call graph precision is more modest [Lhoták and Hendren, 2006; Lhoták and Hendren, 2008; Smaragdakis et al., 2011, 2014], unless very sophisticated context abstractions are used [Feng et al., 2015]. In Scala, where the use of generic type parameters and abstract type members is pervasive, our static-type-based context-sensitive analysis, that can precisely model these features, significantly improves call graph precision.

The technique of using type arguments as context is most closely related to the C# type analysis of [Sallenave and Ducournau, 2012]. Their analysis adds type arguments as context to types of instantiated objects (their analogue of the set $\hat{\Sigma}$). In contrast, our analysis adds context to reachable methods (the set R). The goal of their analysis is to specialize the memory layout of objects, in contrast to our goal of specializing method implementations. As we discussed in Section 5.4.2, the transformation that propagates type parameters from outer classes and methods into inner methods already gives our analysis the precision that would be gained from adding context to instantiated object types.

The technique of using term argument types as context is most closely related to the Cartesian Product Algorithm [Agesen, 1995] and object sensitivity [Milanova et al., 2002, 2005]. Both of these techniques analyze a method in contexts determined by the runtime types of their parameters (CPA) or of only their receiver (object sensitivity). The key difference compared with our technique is that these contexts are estimates of the *dynamic* types of the objects that may flow to the parameters, while our contexts are the *statically* declared types of the arguments at the call site of the method. This difference is important for scalability. In the existing approaches, the number of contexts grows with the number of types instantiated anywhere in the program that flow to the parameters (raised to the power of the number of parameters in the case of CPA). In our approach, the number of contexts of a method is bounded by the number of its call sites (although those call sites may themselves be replicated in different contexts of the caller).

As we indicated in Section 5.3, our analysis is defined as an extension of the context-insensitive Scala call graph construction analysis of [Ali et al., 2014]. Our implementation analyzes only the Scala source code presented to the Dotty compiler, not any of the Java bytecode that forms the rest of the complete program. We use the Separate Compilation Assumption to construct a sound partial call graph for the part of the program that is available for analysis [Ali and Lhoták, 2012; Ali and Lhoták, 2013].

5.7 Conclusion

We have presented several extensions to the $\text{TCA}^{\textit{expand-this}}$ algorithm of [Ali et al., 2014] that both improve call graph precision and decrease analysis time for non-trivial Scala programs. Our algorithms consider type arguments and term argument types, and use them to select more precise targets for virtual dispatch.

We implemented the algorithms in the context of the Dotty compiler and compared their precision and running times on a collection of Scala programs. We have found that $\text{TCA}^{\textit{types}}$ is significantly more precise than $\text{TCA}^{\textit{expand-this}}$, indicating that tracking type parameters would allow a great improvement in precision for common Scala code. Furthermore, we showed that $\text{TCA}^{\textit{types-terms}}$ is slightly more precise than $\text{TCA}^{\textit{expand-this}}$, but is substantially faster, indicating that tracking the static types of the arguments at each call site is beneficial. In particular, the call graphs generated by the context-insensitive $\text{TCA}^{\textit{expand-this}}$ algorithm are too imprecise to be usable for method specialization and inlining. The call graphs from both the $\text{TCA}^{\textit{types}}$ and $\text{TCA}^{\textit{types-terms}}$ algorithms are very precise for this client optimization: they would require specializing the average method only 1.5 times in the worst case, and often much less.

Our work suggests that expressive type systems can not only protect users from writing incorrect code, but could also be used to gather more knowledge about the program in order to enable additional performance optimizations.

While our work was primarily focused on Scala, the ideas contained therein are applicable to other statically typed languages with generic types. In particular, type and term propagation could be used to improve call graph construction algorithms for Java, C#, C++, Haskell, Swift, and D.

6 Example analysis: Extending common subexpression elimination to Idempotent expression

Common subexpression elimination (CSE) is a popular compiler optimization that can improve performance by removing redundant computations if they are idempotent. It is usually done only for primitive operations because these are easily determined to be idempotent. Due to the functional nature of Scala, many non-primitive methods are also idempotent. In this paper, we identify several common idioms in Scala programs whose performance can benefit from CSE. We present an analysis that finds idempotent methods and the calls to them. We have implemented and evaluated this analysis in the Dotty Linker.

6.1 Motivation

The original research pertaining to common subexpression elimination was performed for imperative languages and was aimed at reducing the number of repeated arithmetic operations performed in a method. The eliminated operations were pure, which meant that the optimization was preserving semantics, and users were unable to observe the difference.

While putting the previous research into the perspective of common high-level functional languages, such as Scala, we can observe that arithmetic operations are uncommon [Chitil, 1998]. We observe that the optimization preserves semantics even in cases where the methods are not strictly pure. We introduce the weaker notion of idempotence and we demonstrate that idempotent methods are common in the Dotty compiler.

We define a method to be *idempotent* if, when called twice from another method with the same values as arguments, the second call does not perform observable side-effects and returns the same value as the first call.

This property should hold independently of the values of the arguments. Listing 6.1 provides several examples of methods that are idempotent and some that are not. In particular, `cachedApply` is idempotent, while `apply` is not, because if they are each called twice with the

same arguments, `apply` would produce any side effects in `fun` twice.

Note that every strictly pure function is idempotent according to this definition, because it does not have side-effects (not only on the first call), and its return value depends only on the arguments that are passed. The Scala language has a strong functional background and has several features that are idempotent and could benefit from the reuse of already pre-computed values.

6.1.1 Lazy Values

The Scala Language Specification [Odersky, 2014] defines the notion of *lazy value definitions* as values that are computed the fist time they are accessed. If the computation is successful, future accesses to the same *lazy value* should return the already computed value.

Lazy values are often used by programmers as well as by library designers in order to simplify and organize their code. Consider the code pattern presented below.

In this example, taken from Dotty Namer, the `lazy val lhsType` is not a normal `val`, because the computed value may not be needed, and it is not a `def`, because the computation is costly; if it is needed, then it should be computed only once.

```
844 lazy val lhsType = fullyDefinedType(cookedRhsType, "right-hand side", mdef.pos)
845
846 if (sym.is(Final, butNot = Method) && lhsType.isConstantType)
847   lhsType
848 else inherited
```

Reading a lazy value is an idempotent operation according to the definition of the semantics of lazy values. Although the implementation of lazy values is optimized to make subsequent reads fast, the runtime cost is still substantial. CSE is obviously applicable to reads of lazy values, and it can significantly improve the performance of programs that use them extensively.

6.1.2 Implicit conversions

The Scala Language Specification [Odersky, 2014] defines the notion of implicit conversion as a user-defined method that is inserted by the compiler when an instance of one type is needed but an instance of a different type is provided. Consider the following example:

```
849 implicit def wrapIntArray(xs: Array[Int]): WrappedArray[Int] =
850 if (xs ne null) new WrappedArray.ofInt(xs) else null
851
852 def takesIntSeq(seq: Seq[Int]) = seq.length
853
854 takesIntSeq(Array(1, 2, 3))
```

Because the `Array` type is a Scala representation of Java arrays, it is not a subtype of `Seq`. In

```
809 object Idempotent{ // those examples are idempotent
810   def fibonacci(id: Int): = {
811     if(id <= 1) 1
812     else fibbonacci(id - 1) + fibbonacci(id - 2)
813   }
814
815   private val cache = mutable.Map[Int, Int]()
816   def cachedApply(fun: Int => Int, arg: Int): R = {
817     if (cache.contains(arg)) cache(arg)
818     else {
819       result = fun(arg)
820       cache(arg) = result
821       result
822     }
823   }
824
825   def compose[A, B, C](fun1: A => B, fun2: B => C) = {
826     arg: A =>
827       fun2(fun1(arg))
828   }
829 }
830
831 object NotIdempotent {
832   // those examples are not idempotent
833   def echo(a: String) = println(a)
834
835   def apply(fun: Int => Int, arg: Int): Int =
836     fun(arg)
837
838   var field = 0
839   def readField = field
840   def setField(newValue: Int) = {
841     field = newValue
842   }
843 }
```

Listing 6.1 – Idempotency examples

order for the call of `takesIntSeq` to compile, the implicit conversion `wrapIntArray` is inserted by the compiler:

```
855 takesIntSeq(wrapIntArray(Array(1, 2, 3)))
```

Implicit conversions are silently applied by the compiler, so their presence is not obvious in the source code. Because of this, most implicit conversions defined by programmers in the Scala community are pure and thus idempotent (though this is not required by the language specification).

6.1.3 Domain specific knowledge

Many methods are intended to inquire about information concerning some logically immutable object, and therefore return the same result if called twice. This is ubiquitous in purely functional libraries, but is often found in other areas, as well. We have found several examples of complex computations inside the Dotty compiler which are idempotent based on the domain specific knowledge:

- The Dotty compiler uses a logically immutable `Tree` class to represent nodes of the abstract syntax tree. Some of these nodes are lazy because they need to be lazily loaded from TASTY, the serialization format used for separate compilation. After the first access that loads the tree node, the node no longer changes, so operations on it are idempotent. This applies to the `ValDef`, `DefDef` and `Template` tree nodes.
- An object of the `Denotation` class defines the meaning of a name in the context of some specific object expression. Computing the `Denotation` is a costly operation that may require re-reading the classpath and recomputing members of other symbols using involved logic, but after it is computed, it stays the same during a given phase. Therefore, operations on a `Denotation` are idempotent.
- A `Name` is the representation of an identifier in the source program. Dotty defines many operations which compute various properties of a name, such as `isConstructor`. All of these operations are idempotent because `Name` objects are immutable.

As will be shown later in Section 6.3.1, this initial user-provided information was enough to infer idempotence of many derived methods in Dotty that are commonly used by compiler developers, such as `tree.symbol`

6.2 Implementation

The analysis and transformation have been implemented as a part of the Dotty Linker, an optimizing compiler based on Dotty, a compiler for the Scala Language.

We have introduced an additional annotation `@idempotent` that can be used by users to mark some methods as idempotent. The compiler checks that if a subclass overrides a method that is annotated as `@idempotent`, then the overriding method must also be annotated as `@idempotent`.

6.2.1 Idempotency inference

In order for the transformation to decide which results of method calls can be reused, it needs to know which methods are idempotent. The implemented transformation starts with the following assumptions:

- lazy vals are idempotent as specified by the Scala Language specification[Odersky, 2014];
- accesses to immutable local variables are idempotent;
- calls to accessors of immutable fields are idempotent;
- arithmetic operations are pure;
- methods annotated by the developer as `@idempotent` are idempotent.

Starting with this initial set, the inference algorithm discovers additional idempotent methods using a simple observation: if a method calls only idempotent methods, it is idempotent itself. This inference rule is iterated until a fixed point is reached.

The algorithm takes as input a call graph of the program that is currently being compiled. Our proposed implementation uses the technique of Chapter 5 to construct the input call graph. For every method m reachable through the call graph from program entry points, the algorithm generates a list of all target methods that m could call from any of its call sites. In order to account for dynamic dispatch, the call graph is used to determine which target methods could be called from each call site.

The inference algorithm is shown below. In the algorithm, the set of all possible targets that could be invoked by a method is written as `method.calls`.

The use of a precise call graph makes it possible to infer the idempotency of a method that calls a target method t even if t is overridden by non-idempotent methods, as long as the call graph construction algorithm can prove that those non-idempotent overriding methods are not actually called from the call site. In the example presented in Listing 6.2, the method `foo` defined in trait `Interface` and called in method `main` has a non-idempotent implementation defined in class `DebugImplementation`. A closed-world call-graph construction algorithm is able to infer the call to `foo` to be idempotent in this example, because the `DebugImplementation` class is never allocated.

```
def inferIdempotency(idempotentMethods: Set[Method],
         allMethods: Set[Method]) = {
  val newMethods =  allMethods.filter(method =>
      method.calls ⊂ idempotentMethods
      ) \ idempotentMethods

  if (newMethods.isEmpty) idempotentMethods
  else inferIdempotency(newMethods ∪ idempotentMethods, allMethods)
}
```

```
trait Interface {
  def foo(a: Int): Int
}

class Implementation {
  def foo(a: Int) = 1
}

class DebugImplementation extends Implementation {
  def foo(a: Int) = {
    println("foo")
    super.foo(a)
  }
}

object Main{
  def main(args: Array[String]): Unit = {
    val i: Interface  = new Implementation
    i.foo
  }
}
```

Listing 6.2 – Reachability example

On idempotence of immutable field accessors There is an exception to the idempotence of accessors of immutable fields: they are not necessarily idempotent inside constructors, because constructors initialize (i.e., mutate) the immutable field. Consider the example presented in Listing 6.3. This class uses some intricacies of the Scala field initialization order to observe an uninitialized value.

The crucial observation here is that multiple values of field accessors can only be observed inside a constructor itself. This is because only constructors can initialize the underlying fields of `val`s with new values. Any other method, even if called from a constructor, would always consistently observe the same value returned by getters during its entire execution, because it cannot change the value stored inside the `val`.

```
886 class HasConstructor {
887   println(field + 1)
888     // prints 1
889   val field = 2
890   println(field + 1)
891     // prints 3
892 }
```

Listing 6.3 – Constructor example

```
Denotation.info,
Symbol.denot,
SymDenotation.flags,
SymDenotation.is,
TypeProxy.underlying,
NamedType.denot.
```

Figure 6.1 – Methods annotated as `@idempotent`

This means that while optimizing the body of the constructor, we cannot assume the idempotence of accessors of the fields of the class being constructed. However, this assumption does hold in all other methods.

6.3 Evaluation results

We have evaluated the implemented algorithm on the Dotty source code. Dotty has 55807 lines of source code excluding blank lines or comments that define 3595 classes, 437 traits and 64 objects. We have annotated a very small set of methods used in Dotty as `@idempotent`, using domain specific knowledge. The full list of annotated methods is provided in Figure 6.1. Most of these methods need to be annotated because they encapsulate a carefully controlled laziness. For example, consider `SymDenotation.is`:

```
893 private[this] var myFlags: FlagSet = adaptFlags(initFlags)
894
895 /** The flag set */
896 @Idempotent
897 final def flags(implicit ctx: Context): FlagSet = { ensureCompleted(); myFlags }
898
899 /** Has this denotation one of the flags in `fs` set? */
900 @Idempotent
901 final def is(fs: FlagSet)(implicit ctx: Context) = {
902   (if (fs <= FromStartFlags) myFlags else flags) is fs
903 }
```

Listing 6.4 – SymDenotation.scala

In this example, if the flags being passed to `is` in the `fs` argument are are a subset of `FromStartFlags`, the evaluation could proceed without needing to force the computation of flags done by the `flags` method. This method cannot statically be proven idempotent, as it accesses a mutable variable `myFlags` directly. This is a common pattern seen in the methods named above.

6.3.1 Research Questions

RQ1. How many methods can be discovered to be idempotent using only language specific knowledge?

RQ2. How quickly does the number of idempotent methods grow based on the number of methods annotated by hand?

RQ3. How long does the inference algorithm take to run?

RQ4. Without the closed-world assumption, how many idempotent methods would be inferred?

RQ5. Without language specific knowledge about immutable fields and lazy vals, how many idempotent methods would have been inferred?

6.3.2 Results

RQ1. When not annotating any methods as idempotent and using only the assumptions provided in Section 6.2.1, we start with a set of 835 methods that are idempotent due to the language specification as accessors of immutable fields or lazy val getters. By using the inference algorithm we can additionally infer 7112 methods, out of 23401 methods in Dotty, as idempotent.

RQ2. By annotating six more definitions in Dotty as idempotent, as presented in Figure 6.1, we started with a set of 841 methods assumed to be idempotent and have inferred 7356 methods as idempotent based on this, adding 244 new methods. Those methods include some of commonly used methods in the Dotty codebase such as `Symbol.name`, `Tree.symbol`, and `SymDenotation.enclosingClass`.

RQ3. Every iteration of this algorithm needs to consider all the calls from all the methods. An example can be constructed to show that there are programs on which the algorithm is

cubic in the number of definitions. In those examples, the algorithm would need a linear number of iterations and every iteration would take quadratic time to perform.

In practice we have found the running time of this algorithm to be very low. For the full Dotty codebase, it takes six iterations for the fixed-point computation to converge.

RQ4. Without a closed-world assumption, the algorithm needs to be modified to infer idempotency only if a method is known to be final, as otherwise it could be overridden by a non-idempotent method. We have run the inference algorithm with this additional restriction and the number of inferred methods is 510 and 496 respectively with and without user-defined annotations. Note that the Dotty codebase uses final methods extensively, defining 4960 effectively final methods.

RQ5. If we drop the language specific knowledge about immutable fields and lazy vals, and only assume that arithmetic operations are idempotent, we can infer only 210 methods as idempotent.

6.4 Related Work

6.4.1 Global value numbering

C2, the Java HotSpot Server Compiler, performs common subexpression elimination, constant propagation, global value numbering, and global code motion. The implementation does not make any language specific assumptions and, thus, cannot optimize any of the examples presented in this paper (except for the first example illustrating common subexpressions in an arithmetic expression). It uses an implementation based on [Click, 1995; Rosen et al., 1988] that has been rigorously tested for two decades in production environments. The implementation is very fast and runs in near-linear time: an important attribute for just-in-time compilers.

6.4.2 Partial redundancy elimination

Partial redundancy elimination [Briggs and Cooper, 1994; Chow et al., 1997] is a related technique that eliminates expressions that are computed redundantly on some of the paths through the program. It is a generalization of common subexpression elimination as it would also eliminate redundant expressions that are computed on all the paths.

Unlike common subexpression elimination, partial redundancy elimination may introduce computations that were not required on a specific path, which may slow down the running time of the program. In order to account for this, both static [Horspool and Ho, 1997] and profiling-based [Gupta et al., 1998] cost analyses have been proposed.

Partial redundancy elimination cannot be extended to idempotent expressions in a straightforward way. It uses code motion to reorder the computation of expressions. If those computations are idempotent, but not pure, the first calls to those expressions may have observable side effects, and reordering them changes the behavior of the program.

6.4.3 Purity inference

This work can benefit from specialized analysis and inference systems that infer properties stronger then idempotency. Methods inferred to be pure by purity inference algorithms [Huang et al., 2012] can be used to increase the size of the seed for the idempotency inference algorithm.

6.4.4 Side effect analysis

Several effect systems have been implemented for Scala. Rytz [Rytz, 2014] proposes a practical effect system that is able to additionally express conditional purity based on the types of arguments, such as the purity of the `apply` function presented in Listing 6.1, if it is given a pure argument. Our implementation is currently not able to express this, but we expect this extension to be straightforward. Side effect analysis is an area of on-going active research and the proposed optimizations would benefit from advances in this area.

6.4.5 Pure languages

In languages such as Haskell, where all expressions are pure and referentially transparent, all expressions are idempotent. Though seemingly straightforward, the implementation of common subexpression elimination in the Glasgow Haskell Compiler is quite tricky, as it may affect the laziness of the program [ghc, 2016b]. Instead, there is a predefined set of patterns that the Glasgow Haskell Compiler optimizes. The F.A.Q. section [ghc, 2016a] suggests that users who care about common subexpression elimination should do it by hand.

The previous work for Haskell indicates that common subexpressions are uncommon in Haskell [Chitil, 1998]. The evaluation approach defines several syntactic restrictions. The study has found that subexpressions meeting those restrictions are rarely introduced in Haskell programs, either by Haskell programmers or by the Glasgow Haskell Compiler itself. In their conclusion, however, they acknowledge that their results are difficult to transfer to other functional languages.

6.5 Conclusion

We have proposed a notion of method idempotency and a common subexpression elimination technique that allows the enlargement of the set of expressions that can be optimized

to include calls to user and library-defined idempotent methods. We have explained the interaction with the kinds of control flow present in Scala.

We have found that language specific knowledge is sufficient to discover a substantial number of idempotent functions, even in the absence of user input. We have proposed an algorithm that uses language-specific knowledge as a seed and is able to infer idempotency of other methods. We have demonstrated the viability of this strategy on the Dotty compiler, where approximately one third of methods were proven idempotent using the proposed technique.

We believe that there is a substantial opportunity to optimize repeated calls to these methods and we are working on a transformation that would either prove or refute this hypothesis.

7 Local optimizations

7.1 Motivation

Performing global call graph analysis requires significant resources. We have found that performing optimizations to a single method locally before global optimizations amounts to a simplification that allows us to:

- generate smaller code that runs faster in the interpreter;
- perform language-specific optimizations that general JVM optimizers are not able to perform;
- speed up global analysis by simplifying local trees that serve as input for global analysis;
- permit other phases to be simpler by generating code with minor inefficiencies that will be later removed by local optimizations.

7.2 Local optimizations

In this chapter, we will use the term local optimizations to refer to optimizations that optimize a single method and do not possess whole-program knowledge.

Local optimizations are pairs of visitor and transformer:

```
937 trait LocalOptimisation {
938   /** Gathers information on trees, to be run first. */
939   def visitor(implicit ctx: Context): Tree => Unit
940   /** Does the actual Tree => Tree transformation. */
941   def transformer(implicit ctx: Context): Tree => Tree
942   /** Clears all the local state, to be run last. */
943   def clear(): Unit
944   }
```

Listing 7.2 – LocalOptimization

```
override def transformDefDef(tree: DefDef)(implicit ctx: Context, info: TransformerInfo
    ): Tree = {
  ...
  var rhs0 = tree.rhs
  var rhs1: Tree = null
  ...
  while (rhs1 ne rhs0) {
    rhs1 = rhs0
    val (visitors, transformers, names) =
     ptimizations.map(x => (x.visitor, x.transformer, x.name)).unzip3
    while (names.nonEmpty) {
      val nextVisitor = visitors.head
      val nextTransformer = transformers.head()
      val name = names.head
      rhs0.foreachSubTree(nextVisitor)
      val rhst = new TreeMap() {
          override def transform(tree: Tree)(implicit ctx: Context): Tree = {
            val innerCtx =
              if (tree.isDef && tree.symbol.exists)
                ctx.withOwner(tree.symbol)
              else ctx
            nextTransformer(ctx)(super.transform(tree)(innerCtx))
          }
        }.transform(rhs0)

        rhs0 = rhst
      }
      names = names.tail
      visitors = visitors.tail
      transformers = transformers.tail
    }
  if (rhs0 ne tree.rhs) tpd.cpy.DefDef(tree)(rhs = rhs0)
  else tree
}
```

Listing 7.1 – The main loop of the Simplify phase

Two traversals of the tree are done. The first traversal collects data necessary to decide which rewritings to apply, while the second one performs those rewritings. Calls to the function returned by `visitor` mutate the inner state of *LocalOptimization* that returned it and populate the information that would be necessary for the transformer.

Attribution

Work presented in this chapter was originally performed by the author of this thesis as part of the Dotty Linker project.

Since then this work has been upstreamed to the main Dotty project by Olivier Blanvillain. Olivier has proposed and implemented the bug isolation technique that was described in this section. The upstreamed version currently has an inferior InlineLocalOpts that is not able to rewrite non-trivial code; therefore the speedups obtained by local optimizations in the Dotty upstream are lower than presented in this section.

7.3 The great Simplifier

Local optimizations are performed by a MiniPhase called Simplifier. A short version of Simplifier is presented in Listing 7.1. This miniphase applies local optimisations to the given method one after another until a fixed point has been reached. As such, there is a requirement for all the optimizations to share a termination measure that will ensure that a fixed point will actually be reached.

Because of this, all implemented optimizations are strictly shrinking.

7.4 Implemented optimizations

The following rewritings were implemented, listed in order of execution:

7.4.1 InlineCaseIntrinsics

Rewrites calls to Dotty and Scala2 case class methods that have known behavior: For Dotty case classes CC:

- `CC.apply(...)` → `new CC(...)`
- `CC.unapply(arg): CC` → `arg`
- `CC.unapply(arg): Boolean` → true

For Scala2 case classes:

- `CC.unapply(arg): Option[CC]` →
  `if (arg.isInstanceOf[CC])new Some(new TupleN (arg._1, ...)) else None`

This prepares the code that works with case classes to be further optimized by the next rewritings.

7.4.2 RemoveUnnecessaryNullChecks

This rewriting tracks null checks that have already been performed either explicitly(through a condition) or implicitly (through a method call) and removes the null checks that are known to always succeed. Specific rules are: `a eq null` is replaced by false when:

- a has a singleton type. This covers ThisType, Super and Literal constants;
- there has been a method call on a before this check;
- in case `a.tpe.isNotNull`, which will trigger when Dotty will be extended with non-nullable types.

7.4.3 InlineOptions

Inline calls on Options that are statically known:

- `Some(foo).isEmpty` → false
- `Some(foo).isDefined` → true
- `Some(foo).get` → `foo`
- `None.isEmpty` → true
- `None.isDefined` → false

7.4.4 InlineLabelsCalledOnce

Inlines code blocks that are accessible through jumps and only from a single location.

7.4.5 Valify

Replaces mutable variables that are never written after the first read with vals. The transformation is equivalent to the following rewriting

```
var a = expr1;
/* code that does not read a, but may assign to it */
a = expr2;
/* code that may read a */
```

to

```
expr1;
/* code that does not read a, with assigmnet to a dropped, but computations of assigned
     value kept */
val a = expr2;
/* code that may read a */
```

7.4.6 Devalify

Inlines immutable variables that are aliases to other immutable variables or to immutable fields accessed multiple times through an immutable path. Here is an illustration:

```
val a = expr1;
val b = a; // will be eliminated, all references to b will be replaced by a

case class C(int a)

val c = new C(a)
val d = c.a
val e = c.a // will be eliminated, all referenced to e will be replaced with d
```

7.4.7 Jumpjump

Replaces jumps to blocks that contain only a single jump with the later jump.

7.4.8 DropGoodCasts

Eliminates casts, type tests and null tests for values whose type is either statically known at compile time or has been tested before.

7.4.9 DropNoEffects

Removes side-effect free expressions from block statements and flattens nested blocks. The following rewriting is performed:

- drop pure references that have their value discarded;
- for a selection of a pure field from a qualifier that has its value discarded, drop the

selection but keep the computation of the qualifier;

- for a nested label method that has its returned value always discarded, change the method to return Unit.

7.4.10 InlineLocalObjects

Finds instances of case classes with trivial constructors that never escape the scope and that only receive calls to field accessors; creates local variables to store copies of the fields of those objects and rewrites writes to those fields to also write to those local variables; replaces calls to field accessors by references to those local variables.

This transformation is necessarily quite involved because it is able to handle nested label methods generated by pattern matching.

This does not actually eliminate the local object, but rewrites the code so that is is never read. This object will be removed by a combination of Devalify and DropNoEffects.

Here is an example:

```
961 <label> def bar = new Tuple(3, 4)
962 val a = if (test) new Tuple(1, 2) else bar
963 println(a._1 + a._2)
```

is rewritten to

```
964 <label> def bar = {
965   a$$1 = 3
966   a$$2 = 4
967   new Tuple(a$$1, a$$2)
968 }
969
970 var a$$1 = 0
971 var a$$2 = 0
972 val a = if (test) {
973     a$$1 = 1
974     a$$2 = 2
975     new Tuple(a$$1, a$$2)
976 } else bar
977
978 println(a$$1 + a$$2)
```

7.4.11 Varify

Removes vals that are aliases to existing vars that are not mutated anymore:

```
var a = 1
/* code that may mutate a */
val b = a
/* code that does not mutate a*/
```

is transformed to

```
var a = 1
/* code that may mutate a */
/* code that does not mutate a, with b substituted by a*/
```

7.4.12 bubbleUpNothing

The only way that a type-safe expression can have type "Nothing" is if it either never terminates or never returns, as there are no elements of this type.

This means that all expressions that follow a computation of a Nothing-typed expression will never be computed and all pure expressions that directly precede a Nothing-typed expression will not be observed. This warrants the following rewritings (where "???" represents a Nothing-typed expression):

- `Block(stats1::pureStat::???:::others, expr)` → `Block(stats1, ???)`
- `if (???) then thenp else elsep` → `???`
- `recv.func(args1..., ???, ...)` → `Block(recv :: args1, ???)`

This transformation can be seen as a language-specific extension of dead code elimination.

7.4.13 ConstantFold

Constant expressions are folded to their result. Arithmetic expressions are regularized to have their constants on the left side. For example:
`2 * a * b * 5 + 3 * (c + 1)` → `3 + 10 * a * b + 3 * c`.

This rewriting is also responsible for simplifying the `if` expressions with the following rules:

- `if (test1) {code1} else {code1}`
  →
  `test1; code1`

- `if (test1) {if (test2) code1 else code2} else {code2}`
  →
  `if (test1 && test2) code1 else code2`

- `if (test1) {if (test2) code1 else code2} else {code2}`
  →
  `if (test1 && !test2) code2 else code1`

- `if (test1) {code1} else {if (test2) code1 else code2}`
  →
  `if (test1 || test2) code1 else code2`

- `if (test1) {code1} else {if (test2) code2 else code1}`
  →
  `if (test1 || !test2) code1 else code2`

7.5 Example

7.5.1 Pattern matching on case classes

Consider the method `foo` in the example below:

```
case class CC(a: Int, b: Object)
def foo(x: Any): Int = {
    val (a, b) = x match {
      case CC(s @ 1, CC(t, _)) =>
        (s , 2)
      case _ => (42, 43)
    }
    a + b
}
```

Without local optimizations, the method will be transformed to the bytecode equivalent of the following Java code:

```
public int foo(Object x) {
        var3_2 = x;
        if (!(var3_2 instanceof CC)) ** GOTO lbl-1000
        var4_3 = (CC)var3_2;
        var5_4 = CC$.MODULE$.unapply((CC)var3_2);
        s = var5_4._1();
        var7_6 = var5_4._2();
        if (1 != s) ** GOTO lbl-1000
        var8_7 = s;
        if (var7_6 instanceof CC) {
            var9_8 = (CC)var7_6;
            var10_9 = CC$.MODULE$.unapply((CC)var7_6);
            var11_10 = var10_9._2();
            v0 = Tuple2..MODULE$.apply((Object)BoxesRunTime.boxToInteger((int)1), (
    Object)BoxesRunTime.boxToInteger((int)2));
        } else lbl-1000: // 3 sources:
        {
            v0 = Tuple2..MODULE$.apply((Object)BoxesRunTime.boxToInteger((int)42), (
    Object)BoxesRunTime.boxToInteger((int)43));
        }
        var2_11 = v0;
        a = BoxesRunTime.unboxToInt((Object)var2_11._1());
        b = BoxesRunTime.unboxToInt((Object)var2_11._2());
        return a + b;
    }
```

With the above optimizations enabled, the following code is generated:

```
1018 public int foo(Object x) {
1019         CC cC;
1020         int n = 0;
1021         int n2 = 0;
1022         if (x instanceof CC && 1 == (cC = (CC)x)._1() && cC._2() instanceof CC) {
1023             n = 1;
1024             n2 = 2;
1025         } else {
1026             n = 42;
1027             n2 = 43;
1028         }
1029         return n + n2;
1030     }
```

Here we will show how the optimizations described above allowed us to generate this more efficient code. We start with the following code generated by the Dotty pipeline:

```
def foo(x: Any): Int = {
  val $1$: (Int, Int) = {
    case val selector12: Any = x
    {
      def case31(): (Int, Int) = {
        def case41(): (Int, Int) = {
          def matchFail21(): (Int, Int) = throw new MatchError(selector12)
          {
            {
              Tuple2.apply[Int^, Int^](42, 43)
            }
          }
        }
        if selector12.isInstanceOf[CC] then {
          case val x21: CC = selector12.asInstanceOf[CC]
          {
            case val x31: CC = CC.unapply(selector12.asInstanceOf[CC])
            {
              case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
              case val p41: Object = x31._2
              if 1.==(s) then {
                case val x51: Int(1) = s
                if p41.isInstanceOf[CC] then {
                  case val x61: CC = p41.asInstanceOf[CC]
                  {
                    case val x71: CC = CC.unapply(p41.asInstanceOf[CC])
                    {
                      case val p81: Object = x71._2
                      {
                        Tuple2.apply[Int^, Int^](1, 2)
                      }
                    }
                  }
                } else case41()
              } else case41()
            }
          }
        } else case41()
      }
      case31()
    }
  }
  val a: Int = $1$._1
  val b: Int = $1$._2
  a.+(b)
}
```

The first phase to run is InlineCaseIntrinsics. It replaces two case-class `apply` calls and two `unapply` calls, resulting in the following code (changed parts are bold):

```
1077 def foo(x: Any): Int = {
1078   val $1$: (Int, Int) = {
1079     case val selector12: Any = x
1080     {
1081       def case31(): (Int, Int) = {
1082         def case41(): (Int, Int) = {
1083           def matchFail21(): (Int, Int) = throw new MatchError(selector12)
1084           {{
1085               new Tuple2[Int, Int](42, 43)
1086           }}
1087         }
1088         if selector12.isInstanceOf[CC] then {
1089           case val x21: CC = selector12.asInstanceOf[CC]
1090           {
1091             case val x31: CC = selector12.asInstanceOf[CC]
1092             {
1093               case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1094               case val p41: Object = x31._2
1095               if 1.==(s) then {
1096                 case val x51: Int(1) = s
1097                 if p41.isInstanceOf[CC] then {
1098                   case val x61: CC = p41.asInstanceOf[CC]
1099                   {
1100                     case val x71: CC = p41.asInstanceOf[CC]
1101                     {
1102                       case val p81: Object = x71._2
1103                       {
1104                         new Tuple2[Int, Int](1, 2)
1105                       }
1106                     }
1107                   }
1108                 } else case41()
1109               } else case41()
1110             }
1111           }
1112         } else case41()
1113       }
1114       case31()
1115     }
1116   }
1117   val a: Int = $1$._1
1118   val b: Int = $1$._2
1119   a.+(b)
1120 }
```

InlineLabelsCalledOnce has inlined `case31`, and removed matchFail21, as it was never called (this pattern match never fails).

```
1121 def foo(x: Any): Int = {
1122   val $1$: (Int, Int) = {
1123     case val selector12: Any = x
1124     {
1125       {      // this all has been inlined
1126         def case41(): (Int, Int) = {
1127           {{
1128               new Tuple2[Int, Int](42, 43)
1129           }}
1130         }
1131         if selector12.isInstanceOf[CC] then {
1132           case val x21: CC = selector12.asInstanceOf[CC]
1133           {
1134             case val x31: CC = selector12.asInstanceOf[CC]
1135             {
1136               case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1137               case val p41: Object = x31._2
1138               if 1.==(s) then {
1139                 case val x51: Int(1) = s
1140                 if p41.isInstanceOf[CC] then {
1141                   case val x61: CC = p41.asInstanceOf[CC]
1142                   {
1143                     case val x71: CC = p41.asInstanceOf[CC]
1144                     {
1145                       case val p81: Object = x71._2
1146                       {
1147                         new Tuple2[Int, Int](1, 2)
1148                       }
1149                     }
1150                   }
1151                 } else case41()
1152               } else case41()
1153             }
1154           }
1155         } else case41()
1156       }
1157     }
1158   }
1159   val a: Int = $1$._1
1160   val b: Int = $1$._2
1161   a.+(b)
1162 }
```

Later, Devalify has eliminated the redundant local variables `a`, `b`, `selector12`, `x21`, `x51`, `x61`, `p82`, generating the following code:

```
1163 def foo(x: Any): Int =  {
1164   val $1$: (Int, Int) = {
1165     {
1166       {
1167         def case41(): (Int, Int) = {
1168           {
1169             {
1170               new Tuple2[Int, Int](42, 43)
1171             }
1172           }
1173         }
1174         if x.isInstanceOf[CC] then {
1175           x.asInstanceOf[CC]
1176           {
1177             case val x31: CC = x.asInstanceOf[CC]
1178             {
1179               case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1180               case val p41: Object = x31._2
1181               if 1.==(s) then {
1182                 s
1183                 if p41.isInstanceOf[CC] then {
1184                   p41.asInstanceOf[CC]
1185                   {
1186                     case val x71: CC = p41.asInstanceOf[CC]
1187                     {
1188                       x71._2
1189                       {
1190                         new Tuple2[Int, Int](1, 2)
1191                       }
1192                     }
1193                   }
1194                 } else case41()
1195               } else case41()
1196             }
1197           }
1198         } else case41()
1199       }
1200     }
1201   }
1202   $1$._1.+($1$._2)
1203 }
```

As you can see, there are some casts in the statement positions remaining after Devalify, as it does not know if they will succeed. DropGoodCasts will remove those two casts that are known to succeed:

```
1204 def foo(x: Any): Int =  {
1205   val $1$: (Int, Int) = {
1206     {
1207       {
1208         def case41(): (Int, Int) = {
1209           {
1210             {
1211               new Tuple2[Int, Int](42, 43)
1212             }
1213           }
1214         }
1215         if x.isInstanceOf[CC] then {
1216          // cast removed
1217           {
1218             case val x31: CC = x.asInstanceOf[CC]
1219             {
1220               case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1221               case val p41: Object = x31._2
1222               if 1.==(s) then {
1223                 s
1224                 if p41.isInstanceOf[CC] then {
1225                 // cast removed
1226                   {
1227                     case val x71: CC = p41.asInstanceOf[CC]
1228                     {
1229                       x71._2
1230                       {
1231                         new Tuple2[Int, Int](1, 2)
1232                       }
1233                     }
1234                   }
1235                 } else case41()
1236               } else case41()
1237             }
1238           }
1239         } else case41()
1240       }
1241     }
1242   }
1243   $1$._1.+($1$._2)
1244 }
```

Now, dropNoEffects is eliminating all the pure expressions that have their value discarded and will flatten blocks:

```
1245 def foo(x: Any): Int = {
1246   val $1$: (Int, Int) = {
1247     def case41(): (Int, Int) = new Tuple2[Int, Int](42, 43)
1248     if x.isInstanceOf[CC] then {
1249       case val x31: CC = x.asInstanceOf[CC]
1250       case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1251       case val p41: Object = x31._2
1252       if 1.==(s) then
1253         if p41.isInstanceOf[CC] then {
1254           case val x71: CC = p41.asInstanceOf[CC]
1255           new Tuple2[Int, Int](1, 2)
1256         } else case41()
1257        else case41()
1258     } else case41()
1259   }
1260   $1$._1.+($1$._2)
1261 }
```

InlineLocalObjects will implement a strategy to get rid of the local tuple 1 that never escapes the scope and will replace it by two local variables representing fields:

```
1262 def foo(x: Any): Int = {
1263   var $1$$_1: Int = 0
1264   var $1$$_2: Int = 0
1265   val $1$: (Int, Int) = {
1266     def case41(): (Int, Int) = {
1267       $1$$_1 = 42
1268       $1$$_2 = 43
1269       new Tuple2[Int, Int]($1$$_1, $1$$_2)
1270     }
1271     if x.isInstanceOf[CC] then {
1272       case val x31: CC = x.asInstanceOf[CC]
1273       case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1274       case val p41: Object = x31._2
1275       if 1.==(s) then
1276         if p41.isInstanceOf[CC] then {
1277           case val x71: CC = p41.asInstanceOf[CC]
1278           {
1279             $1$$_1 = 1
1280             $1$$_2 = 2
1281             new Tuple2[Int, Int]($1$$_1, $1$$_2)
1282           }
1283         } else case41()
1284        else case41()
1285     } else case41()
1286   }
1287   $1$$_1.+($1$$_2)
1288 }
```

ConstantFold has figured out that the two branches of the **if** statement are the same and has joined them:

```
1289  def foo(x: Any): Int = {
1290   var $1$$_1: Int = 0
1291   var $1$$_2: Int = 0
1292   val $1$: (Int, Int) = {
1293     def case41(): (Int, Int) = {
1294       $1$$_1 = 42
1295       $1$$_2 = 43
1296       new Tuple2[Int, Int]($1$$_1, $1$$_2)
1297     }
1298     if x.isInstanceOf[CC] then {
1299       case val x31: CC = x.asInstanceOf[CC]
1300       case val s: Int(1) = x31._1.asInstanceOf[Int(1)]
1301       case val p41: Object = x31._2
1302       if 1.==(s).&&(p41.isInstanceOf[CC]) then {
1303         case val x71: CC = p41.asInstanceOf[CC]
1304         {
1305           $1$$_1 = 1
1306           $1$$_2 = 2
1307           new Tuple2[Int, Int]($1$$_1, $1$$_2)
1308         }
1309       } else case41()
1310     } else case41()
1311   }
1312   $1$$_1.+($1$$_2)
1313 }
```

At this point the second iteration of the optimization loop takes place. The first transformation that had effect was Devalify, which removed `x71, $1$, s, p41`. The most important removal is `$1$`, as it will allow us to eliminate tuple creation later.

```
1315  def foo(x: Any): Int = {
1316   var $1$$_1: Int = 0
1317   var $1$$_2: Int = 0
1318   { // result of this blog used to be assigned to $1$
1319     def case41(): (Int, Int) = {
1320       $1$$_1 = 42
1321       $1$$_2 = 43
1322       new Tuple2[Int, Int]($1$$_1, $1$$_2)
1323     }
1324     if x.isInstanceOf[CC] then {
1325       case val x31: CC = x.asInstanceOf[CC]
1326       if 1.==(x31._1.asInstanceOf[Int(1)]).&&(x31._2.isInstanceOf[CC]) then {
1327         p41.asInstanceOf[CC]
1328         {
1329           $1$$_1 = 1
1330           $1$$_2 = 2
```

```
1331            new Tuple2[Int, Int]($1$$_1, $1$$_2)
1332          }
1333        } else case41()
1334      } else case41()
1335    }
1336    $1$$_1.+($1$$_2)
1337  }
```

As before, we have casts left by Devalify that have their results discarded and will never fail. `p41.asInstanceOf[CC]` will be eliminated by dropGoodCasts. Now, dropNoEffects is able to eliminate tuple allocations!:

```
1338  def foo(x: Any): Int = {
1339    var $1$$_1: Int = 0
1340    var $1$$_2: Int = 0
1341    def case41(): (Int, Int) = {
1342      $1$$_1 = 42
1343      $1$$_2 = 43
1344      ()
1345    }
1346    if x.isInstanceOf[CC] then {
1347      case val x31: CC = x.asInstanceOf[CC]
1348      if 1.==(x31._1.asInstanceOf[Int(1)]).&&(x31._2.isInstanceOf[CC]) then {
1349        $1$$_1 = 1
1350        $1$$_2 = 2
1351        ()
1352      } else case41()
1353    } else case41()
1354    $1$$_1.+($1$$_2)
1355  }
```

This is the set of iterations that allowed us to generate much improved code. There are still several rewriting opportunities that are missed, however: `case41` is called in both **`else`** branches of the **`if`** statements, but the inner **`if`** needs some pre-initialization before it will be able to make the test.

7.5.2 Pattern matching on tuples of booleans

Consider the code snippet below:

```
1356    def booleans(a: Object) = {
1357      val (b1, b2) = (a.isInstanceOf[CC], a.isInstanceOf[List[Int]])
1358      (b1, b2) match {
1359        case (true, true) => true
1360        case (false, false) => true
1361        case _ => false
1362      }
1363    }
```

The current Dotty with optimizations disabled will compile it to bytecode equivalent to the Java code below:

```
1364 public boolean booleans(Object a) {
1365         Tuple2 tuple2 = Tuple2..MODULE$.apply((Object)BoxesRunTime.boxToBoolean((
     boolean)(a instanceof CC)), (Object)BoxesRunTime.boxToBoolean((boolean)(a
     instanceof List)));
1366         boolean b1 = BoxesRunTime.unboxToBoolean((Object)tuple2._1());
1367         boolean b2 = BoxesRunTime.unboxToBoolean((Object)tuple2._2());
1368         Tuple2 tuple22 = Tuple2..MODULE$.apply((Object)BoxesRunTime.boxToBoolean((
     boolean)b1), (Object)BoxesRunTime.boxToBoolean((boolean)b2));
1369         Option option = Tuple2..MODULE$.unapply(tuple22);
1370         if (option.isDefined()) {
1371             Tuple2 tuple23 = (Tuple2)option.get();
1372             boolean bl = BoxesRunTime.unboxToBoolean((Object)tuple23._1());
1373             boolean bl2 = BoxesRunTime.unboxToBoolean((Object)tuple23._2());
1374             if (bl) {
1375                 boolean bl3 = bl;
1376                 if (bl2) {
1377                     boolean bl4 = bl2;
1378                     return true;
1379                 }
1380             }
1381         }
1382         Option option2 = Tuple2..MODULE$.unapply(tuple22);
1383         if (!option2.isDefined()) return false;
1384         Tuple2 tuple24 = (Tuple2)option2.get();
1385         boolean bl = BoxesRunTime.unboxToBoolean((Object)tuple24._1());
1386         boolean bl5 = BoxesRunTime.unboxToBoolean((Object)tuple24._2());
1387         if (bl) return false;
1388         boolean bl6 = bl;
1389         if (bl5) return false;
1390         boolean bl7 = bl5;
1391         return true;
1392     }
```

With local optimizations enabled, this bytecode is generated instead:

```
1393     public boolean booleans(Object a) {
1394         boolean bl = a instanceof CC;
1395         boolean bl2 = a instanceof List;
1396         if (bl && bl2 || !bl && !bl2) {
1397             return true;
1398         }
1399         return false;
1400     }
```

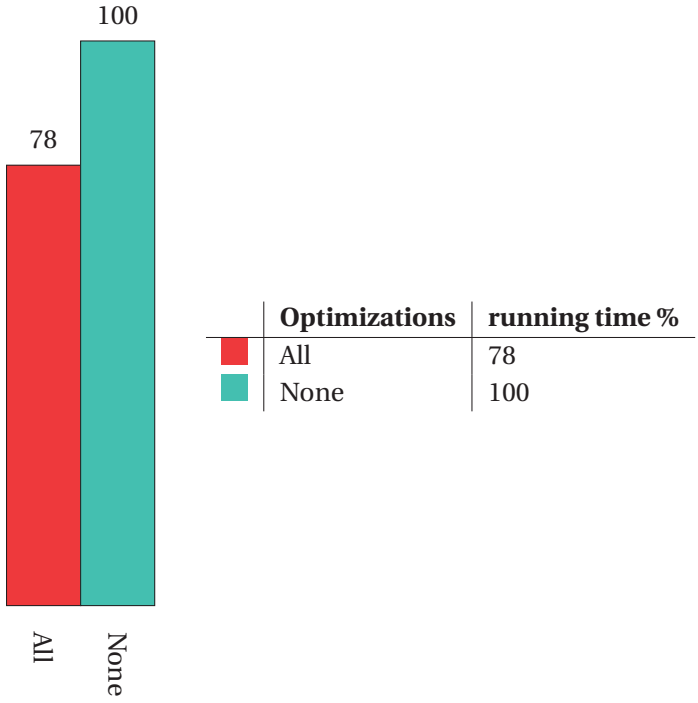
7.6 Evaluation

We have evaluated the performance impact of running the full suite of rewritings on the Dotty compiler itself. We have evaluated the implementation in the following modes:

- enabling a single transformation;
- enabling all optimizations at once;
- enabling all optimizations but one.

We have used Dotty itself as an application to evaluate these optimizations. The measured times have been scaled so that the speed of Dotty without local optimizations is taken to be 100%. The results are presented in Figure 7.1, Figure 7.2 and Figure 7.3.

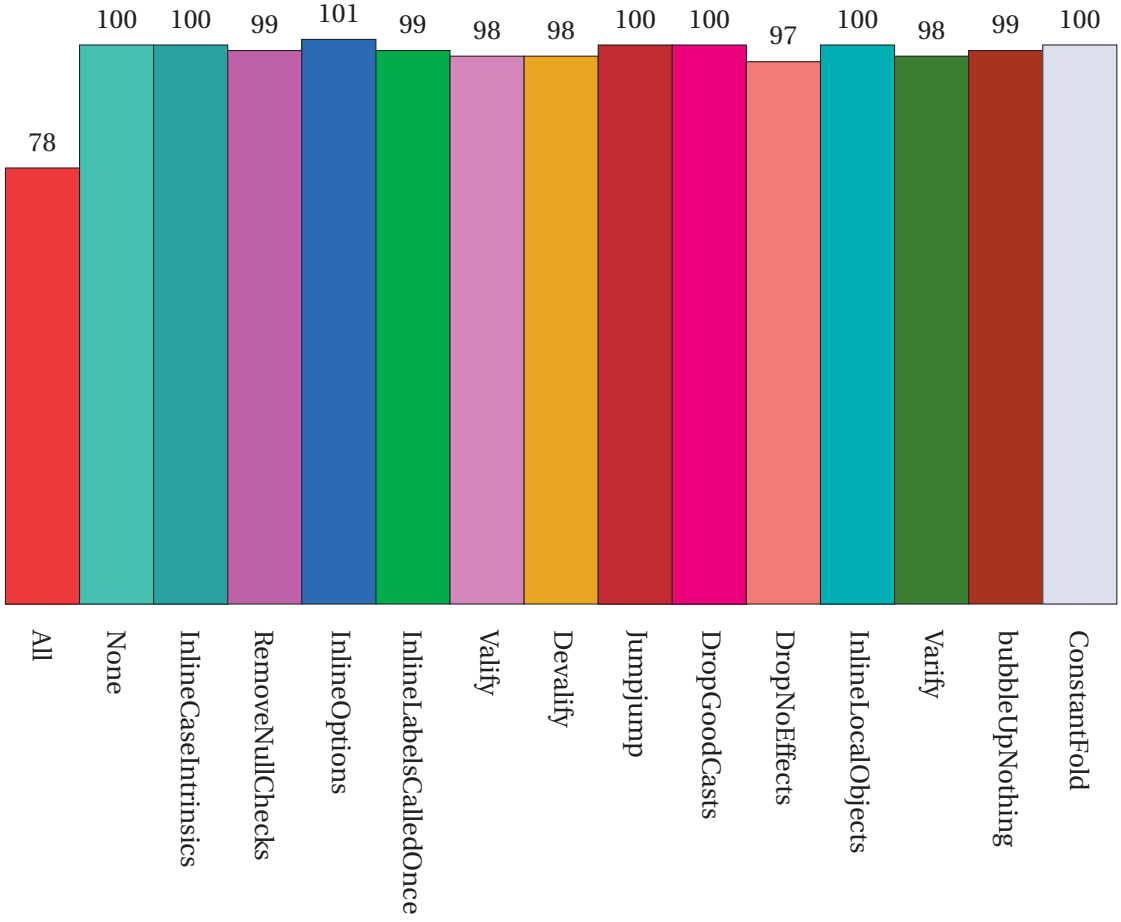
Figure 7.1 indicates that these optimizations introduce a substantial speedup for the generated code, amounting to 22% less time needed to compile Dotty with an optimized Dotty.



Optimizations	running time %
All	78
None	100

Figure 7.1 – Speedup by applying all optimizations

Figure 7.2 shows that none of the optimizations are very powerful in isolation. Each one of these optimizations triggers rarely and has a small effect, but they frequently trigger each other. The biggest speedup provided by a transformation in isolation is 3%; this is obtained by DropNoEffects.

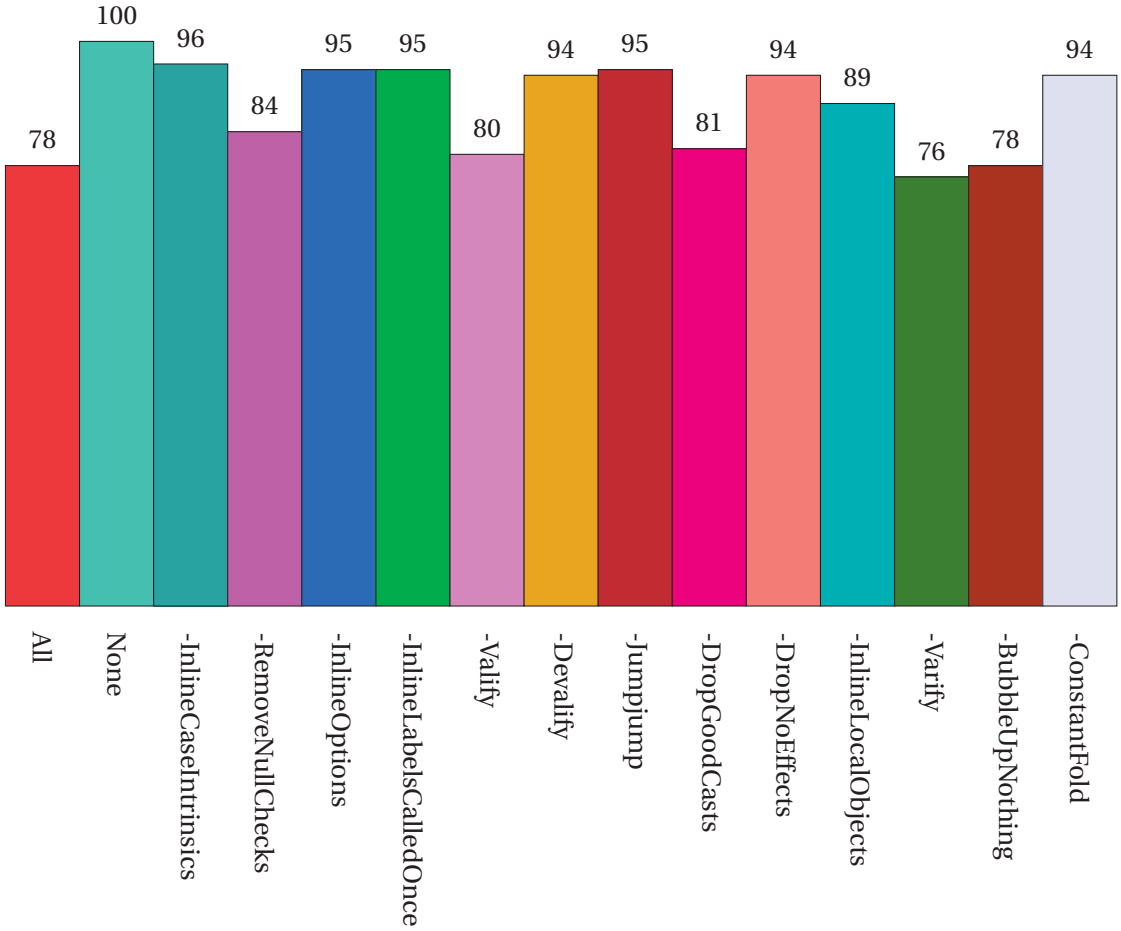


Optimizations	running time %
All	78
None	100
InlineCaseIntrinsics	100
RemoveNullChecks	99
InlineOptions	101
InlineLabelsCalledOnce	99
Valify	98
Devalify	98
Jumpjump	100
DropGoodCasts	100
DropNoEffects	97
InlineLocalObjects	100
Varify	98
bubbleUpNothing	99
ConstantFold	100

Figure 7.2 – Speedup by enabling a single optimization

Figure 7.3 Shows the impact of disabling individual transformations. This graph helps classify transformations by their importance. In particular:

- Disabling any one of InlineCaseIntrinsics, InlineOptions, InlineLabelsCalledOnce, Devalify, ConstantFold, or DropNoEffects makes the performance to regress to 94–95%. All these rewritings are necessary to efficiently optimize pattern matching; disabling any one of them stops optimization early. Disabling any of these transformations loses 17% of the speedup out of 22%.
- InlineLocalObjects is in the second "cohort" by order of importance. Disabling it leaves us with 11% speedup, leaving 11% of potential additional speedup unattained.
- Disabling RemoveNullChecks would remove 6% of the speedup.
- Valify, Varify and BubbleUpNothing are minor transformations that rarely enable others and thus don't contribute much to the speedup. Varify actually stops other transformations from happening by marking locals as vars and introduces a slowdown.



Optimizations	running time %
All	78
None	100
-InlineCaseIntrinsics	96
-RemoveNullChecks	84
-InlineOptions	95
-InlineLabelsCalledOnce	95
-Valify	80
-Devalify	94
-Jumpjump	95
-DropGoodCasts	81
-DropNoEffects	94
-InlineLocalObjects	89
-Varify	76
-BubbleUpNothing	78
-ConstantFold	94

Figure 7.3 – Speedup by enabling all optimizations but one

8 Conclusions and Future Work

8.1 Conclusions

We have demonstrated that the strength of expressive type systems can be used to create compilers that are both maintainable and fast. We have also shown that the underlying type system can be used to create a natural context for context-sensitive analyses, in particular, call graph construction algorithms.

These findings are part of the Dotty project that started as an experiment searching for a better architecture for a Scala compiler. The architecture and code decisions presented in this thesis are the current design of the Dotty compiler at the moment of writing and have not been changed for the last 2 years. This design has been a success and future version of Scala 3 language is planned to be using Dotty as the main compiler [Moors, 2011], [Petrashko, 2011].

8.1.1 MiniPhases

A MiniPhases-based design for compiler has been shown to be a practical high level design of tree transformations in a pass-based compiler. It introduces a natural separation of concerns that helps maintainability by i) fixing traversal oder for transformation to be in-order traversal; ii) separating transformations for different tree node kinds. This introduces a uniform way to write transformations that improves maintainability. At the same time, both these invariants can be utilized to efficiently fuse multiple transformations. This achieves both maintainability and performance in a single design.

8.1.2 CallGraph construction with types as contexts

We have presented $\text{TCA}^{\textit{types-terms}}$, a context sensitive callgraph construction algorithm that uses typing context for context-sensitivity. This kind of context is able to take advantage of the underlying type system of a language. For programming languages that have highly expressive type systems, such context sensitivity allows to build call graphs that are both more precise

and faster to build.

8.2 Future work

8.2.1 Term specialization

The work presented in this thesis for call-graph construction has been demonstrated on class specialization for type parameters. But the callgraph both in the formalization and in the implementation treats type parameters and term parameters uniformly. This suggests that the same analysis can be used to create copies of methods or classes where either arguments or parts of the environment have types that are more precise than the static types observed at the definition site:

```
def delegate[T](arg: T)(fun: T => T) = doApply[T](arg, fun)
def doApply[T](arg: T, fun: T => T) = fun(arg)

delegate(1)(x => x) + delegate(2)(x => x + 1) + delegate(3)(x => x + 2)
```

Will be rewritten to

```
def delegate[T](arg: T)(fun: T => T) = doApply[T](arg, fun)
def doApply[T](arg: T, fun: T => T) = fun(arg)

// duplicated due to term specialization

def delegate1(arg: 1.type)(fun: Lambda1) = doApply1(arg, fun)
def doApply1(arg: 1.type)(fun: Lambda1) = fun(arg)
 // where Lambda1 is type that indicates that this is a lambda
 // with underlying function x: 1.type => x

def delegate2(arg: 2.type)(fun: Lambda2) = doApply2(arg, fun)
def doApply2(arg: 2.type)(fun: Lambda2) = fun(arg)
 // where Lambda2 is type that indicates that this is a lambda
 // with underlying function x: 2.type => x + 1


def delegate3(arg: 3.type)(fun: Lambda3) = doApply3(arg, fun)
def doApply3(arg: 3.type)(fun: Lambda3) = fun(arg)
 // where Lambda3 is type that indicates that this is a lambda
 // with underlying function x: 3.type => x + 2


delegate1(1)(x => x) + delegate2(1)(x => x + 1) + delegate3(1)(x => x + 2)
```

In particular, it would be nice to see this approach applied to The Inlining problem.

8.2.2 The Inlining problem

In 2011, Dr. Cliff Click presented an Inlining problem that stops contemporary JVMs from optimizing functional-style code. Consider a snippet below:

```
1428  def foo(a: Int, b: Int) = {
1429    a ^ b
1430  }
1431  def compute(until: Int): Int = {
1432    var s = 0;
1433    for (i <- 0 to until)
1434      s = foo(s, i)
1435
1436    s
1437  }
```

and compare it with seemingly equivalent Java snippet:

```
1438    public int foo(int a, int b) {
1439        return a ^ b;
1440    }
1441    public int compute(int until) {
1442        int s = 0;
1443        for (int i = 0; i <= until; i++)
1444          s = foo(s, i);
1445        return s;
1446    }
```

Unfortunately these two snippets behave substantially differently performance-wise in a real-world system. The reason is clear after we consider the desugaring of the Scala snippet:

```
1447    def foo(a: Int, b: Int): Int = {
1448        a.^(b)
1449      }
1450    def compute(until: Int): Int = {
1451        val s: scala.runtime.IntRef = scala.runtime.IntRef$create(0)
1452        scala.runtime.RichInt.to$extension0(intWrapper(0), until).foreach(
1453          {
1454            new Function1{ def apply(i: Int): Unit =
1455              {
1456                val ev$1: Int = this.foo(s.elem, i)
1457                s.elem = ev$1
1458              }
1459          }
1460        )
1461        s.elem
1462      }
```

It becomes clear that the `for` loop is desugared in Scala into a call to `foreach` that takes the

body of the cycle as a lambda.

```
1463 // scala.immutable.collection.Range
1464 final override def foreach[@specialized(Unit) U](f: Int => U) {
1465   val isCommonCase = (start != Int.MinValue || end != Int.MinValue)
1466   var i = start
1467   var count = 0
1468   val terminal = terminalElement
1469   val step = this.step
1470   while(
1471     if(isCommonCase) { i != terminal }
1472     else             { count < numRangeElements }
1473   ) {
1474     f(i)
1475     count += 1
1476     i += step
1477   }
1478 }
```

Consider Line 1474. All bodies of possible for-cycles on ranges are called on this line. In a simple micro-benchmark, there will be a single target for this call. In real code, the invocation on Line 1474 is always megamorphic.

This call would become monomorphic if foreach was inlined here, but unfortunately this rarely happens: foreach is likely to become hot first and it will be compiled first and will not be re-compiled and re-profiled for other callers.

Quoting a short summary by Dr. Cliff Click:

> "The Problem" is simply this: new languages on the JVM (e.g. JRuby) and new programming paradigms (e.g. Fork Join) have exposed a weakness in the current crop of inlining heuristics. Inlining is not happening in a crucial point in hot code exposed by these languages, and the lack of inlining is hurting performance in a major way.

Dr. Cliff Click also proposed a possible solution: ask programmers to write their programs in a "megamorphic inlining friendly" coding style, and move virtual dispatch outside of the cycle by hand. Unfortunately, it is very hard do this operation manually if the cycle is inside the standard library of the language, like in the example above.

But it can be done automatically, with good call graph construction. Both in this example and in a lot more complex ones, the call graph construction algorithm presented in Chapter 5 is able to figure out that a specific lambda defined by a for-loop reaches the call on Line 1474. This knowledge can be used to implement either of two rewritings that would make the code above inlinable:

- Use term specialization to duplicate the path from the lambda to Line 1474. This makes the call monomorphic again and brings back performance but has the disadvantage of also duplicating the code that does not need to be duplicated, such as Lines 1466-1469. This is easy to implement and a prototype was implemented that works for this test-case.
- Use knowledge from the call graph to move the cycle on Lines 1470-1477 inside the iterator, see Listing 8.1. This dupplicates the body of the cycle into the class that represents the labmda and is close to the suggestion of Dr. Click. This is harder to implement as it needs to be able to detect nesting in a cycle across virtual dispatches.

  Note that the body of `foreach$apply` method is completely identical in every anonymous subclass created from `Function$Range$Foreach`. The reason that we do not just inherit a single implementation from the `Function$Range$Foreach` class is to make the call to apply inside it monomorphic.

Note that both proposed techniques can be used in an open world and do not require the closed world assumption as they keep the generic path intact.

8.2.3 MiniPhasing more of the compiler

As has been seen in Chapter 1, a substantial amount of time in the compiler is spent outside of the MiniPhases, in particular in Typers(Frontend and Erasure) and Backend. It would be nice to see if parts of the work that are currently performed by them can be converted into MiniPhases. The author of this thesis has successfully moved substantial amount of the logic that was previously in the Backend into small MiniPhases, namely, collection of entry points, creation of static method in the right place, preparation of static calls and preparation of local methods that will be compiled into local jumps.

8.2.4 Adding more pre and post-conditions and checking their completeness

Currently, the completeness of pre and post-conditions of the MiniPhses are not checked either statically or dynamically.

One possible technique to dynamically check both post and pre-conditions is to fuzz-test phase ordering. During compilation with phases reordered randomly, either compilation should succeed and emit the right result, or post-&pre- conditions should have triggered.

```
def foo(a: Int, b: Int): Int = {
  a.^(b)
  }
def compute(until: Int): Int = {
  val s: scala.runtime.IntRef = scala.runtime.IntRef$create(0)
  scala.runtime.RichInt.to$extension0(intWrapper(0), until).foreach(
    {
      new Function$Range$Foreach{
        def apply(i: Int): Unit = {
          val ev$1: Int = this.foo(s.elem, i)
          s.elem = ev$1
        }
        def foreach$apply(r: Range, isCommonCase: Boolean,
                                terminal: Int, step: Int): Unit = {
          while(if (isCommonCase) { i != terminal }
                  else            { count < numRangeElements }
           ) {
            apply(i)
            count += 1
            i += step
          }
        }
      }
    }
  )
  s.elem
}

// scala.immutable.collection.Range
final override def foreach[@specialized(Unit) U](f: Int => U) {
  val isCommonCase = (start != Int.MinValue || end != Int.MinValue)
  var i = start
  var count = 0
  val terminal = terminalElement
  val step = this.step
  if (f.isInstanceOf[Function$Range$Foreach])
    f.asInstanceOf[Function$Range$Foreach].apply(this, isCommonCase, terminal, step)
  else {
    while(
      if(isCommonCase) { i != terminal }
      else             { count < numRangeElements }
    ) {
      f(i)
      count += 1
      i += step
    }
  }
}
```

Listing 8.1 – Pushing virtual dispatch out of the cycle

Bibliography

(2016a). Haskell wiki: Faq.

(2016b). Haskell wiki: Ghc optimisations.

Agesen, O. (1995). The Cartesian product algorithm. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–51.

Alblas, H. (1991). Attribute evaluation methods. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 48–113, Berlin, Heidelberg. Springer Berlin Heidelberg.

Ali, K. and Lhoták, O. (2012). Application-only call graph construction. In Noble, J., editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 688–712. Springer.

Ali, K. and Lhoták, O. (2013). Averroes: Whole-program analysis without the whole program. In Castagna, G., editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer.

Ali, K., Rapoport, M., Lhoták, O., Dolby, J., and Tip, F. (2014). Constructing call graphs of scala programs. In Jones, R., editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 54–79. Springer Berlin Heidelberg.

Ali, K., Rapoport, M., Lhoták, O., Dolby, J., and Tip, F. (2015). Type-based call graph construction algorithms for Scala. *ACM Trans. Softw. Eng. Methodol.*, 25(1):9:1–9:43.

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA. ACM.

Bravenboer, M. and Smaragdakis, Y. (2009). Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA. ACM.

Briggs, P. and Cooper, K. D. (1994). Effective partial redundancy elimination. In *ACM SIGPLAN Notices*, volume 29, pages 159–170. ACM.

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

Chitil, O. (1998). Common subexpressions are uncommon in lazy functional languages. In Clack, C., Hammond, K., and Davie, T., editors, *Implementation of Functional Languages: 9th International Workshop, IFL'97 St. Andrews, Scotland, UK September 10–12, 1997 Selected Papers*, pages 53–71, Berlin, Heidelberg. Springer Berlin Heidelberg.

Chow, F., Chan, S., Kennedy, R., Liu, S.-M., Lo, R., and Tu, P. (1997). A new algorithm for partial redundancy elimination based on ssa form. In *ACM SIGPLAN Notices*, volume 32, pages 273–286. ACM.

Click, C. (1995). Global code motion/global value numbering. In *ACM SIGPLAN Notices*, volume 30, pages 246–257. ACM.

Click, C. (2011). Fixing the inlining "problem".

Coppel, Y. (2008). Reflecting scala.

Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: from lists to streams to nothing at all. In Hinze, R. and Ramsey, N., editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 315–326. ACM.

Dean, J., Grove, D., and Chambers, C. (1995). Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101.

Dotty, t. (2015). Dotty documentation: Name based pattern matching.

Dragos, I. (2010). *Compiling Scala for Performance*. PhD thesis, IC, Lausanne.

Dragos, I. and Odersky, M. (2009). Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47. ACM.

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

Feng, Y., Wang, X., Dillig, I., and Lin, C. (2015). EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 520–534. ACM.

Gill, A. J. (1996). *Cheap deforestation for non-strict functional languages*. PhD thesis, University of Glasgow, UK.

Goetz, B. (2014). State of the Specialization.

Goetz, B. and Rose, J. (2017). Pattern matching for java – runtime and translation.

Gupta, R., Berson, D. A., and Fang, J. Z. (1998). Path profile guided partial redundancy elimination using speculation. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 230–239. IEEE.

Harper, R. and Morrisett, G. (1995). Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141. ACM.

Henglein, F. and Jørgensen, J. (1994). Formally optimal boxing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–226. ACM.

Horspool, R. N. and Ho, H. (1997). Partial redundancy elimination driven by a cost-benefit analysis. In *Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on*, pages 111–118. IEEE.

Huang, W., Milanova, A., Dietl, W., and Ernst, M. D. (2012). Reim & reiminfer: Checking and inference of reference immutability and method purity. In *ACM SIGPLAN Notices*, volume 47, pages 879–896. ACM.

Intel Corporation (2016). Intel 64 and IA-32 architectures optimization reference manual.

Jo, Y. and Kulkarni, M. (2011). 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. ACM.

Jo, Y. and Kulkarni, M. (2012). Automatically enhancing locality for tree traversals with traversal splicing. In Leavens, G. T. and Dwyer, M. B., 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.

Johnsson, T. (1985). Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, pages 190–203. Springer.

Jourdan, M. (1991). A survey of parallel attribute evaluation methods. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 234–255, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kastens, U. (1980). Ordered attributed grammars. *Acta Informatica*, 13(3):229–256.

Kastens, U. (1991). Implementation of visit-oriented attribute evaluators. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*, pages 114–139, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kastrinis, G. and Smaragdakis, Y. (2013). Hybrid context-sensitivity for points-to analysis. In Boehm, H. and Flanagan, C., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 423–434. ACM.

Kennedy, A. and Syme, D. (2001). Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices*, volume 36, pages 1–12. ACM.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145.

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

Lepper, M. and Trancón y Widemann, B. (2011). Optimization of visitor performance by reflection-based analysis. In Cabot, J. and Visser, E., editors, *Theory and Practice of Model Transformations: 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, pages 15–30, Berlin, Heidelberg. Springer Berlin Heidelberg.

Leroy, X. (1992). Unboxed Objects and Polymorphic Typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 177–188, New York, NY, USA. ACM.

Lewis, P., Rosenkrantz, D., and Stearns, R. (1974). Attributed translations. *Journal of Computer and System Sciences*, 9(3):279 – 307.

Lhoták, O. and Hendren, L. (2003). Scaling Java points-to analysis using Spark. In Hedin, G., editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland. Springer.

Lhoták, O. and Hendren, L. (2006). Context-sensitive points-to analysis: is it worth it? In Mycroft, A. and Zeller, A., editors, *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna. Springer.

Lhoták, O. and Hendren, L. (2008). Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53.

Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Meyerovich, L. A., Torok, M. E., Atkinson, E., and Bodik, R. (2013). 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. ACM.

Milanova, A., Rountev, A., and Ryder, B. G. (2002). Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. ACM Press.

Milanova, A., Rountev, A., and Ryder, B. G. (2005). Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41.

Moors, A. (2011). Scala 2.12 & beyond.

Morrison, R., Dearle, A., Connor, R. C. H., and Brown, A. L. (1991). An Ad Hoc Approach to the Implementation of Polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371.

Nystrom, N., Clarkson, M. R., and Myers, A. C. (2003). Polyglot: An extensible compiler framework for java. In Hedin, G., 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. Springer Berlin Heidelberg.

Odersky, M. (2014). The scala language specification v 2.9.

Odersky, M., Martres, G., and Petrashko, D. (2016). Implementing higher-kinded types in dotty. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, SCALA 2016, pages 51–60, New York, NY, USA. ACM.

Odersky, M. and Zenger, M. (2005). Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 41–57, New York, NY, USA. ACM Press.

Petrashko, D. (2011). Announcing dotty 0.1.2-rc1, a major step towards scala 3.

Petrashko, D., Doeraene, S., and Odersky, M. (2011). Sip 25 - @static fields and methods in scala objects(si-4581).

Petrashko, D., Lhoták, O., and Odersky, M. (2017). Miniphases: Compilation using modular and efficient tree transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 201–216, New York, NY, USA. ACM.

Petrashko, D., Ureche, V., Lhoták, O., and Odersky, M. (2016). Call graphs for languages with parametric polymorphism. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 394–409, New York, NY, USA. ACM.

Phillips, P. (2013). scalac2 pull request: Pattern matcher: extractors become name-based.

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

Rajbhandari, S., Kim, J., Krishnamoorthy, S., Pouchet, L., Rastello, F., Harrison, R. J., and Sadayappan, P. (2016a). A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In West, J. and Pancake, C. M., 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.

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

Riis Nielson, H. (1983). Computation sequences: A way to characterize classes of attribute grammars. *Acta Informatica*, 19(3):255–268.

Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27. ACM.

Rytz, L. (2014). *A Practical Effect System for Scala.* PhD thesis, IC, Lausanne.

Sallenave, O. and Ducournau, R. (2012). Lightweight generics in embedded systems through static analysis. In Wilhelm, R., Falk, H., and Yi, W., editors, *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012, LCTES '12, Beijing, China - June 12 - 13, 2012*, pages 11–20. ACM.

Sarkar, D., Waddell, O., and Dybvig, R. K. (2005). Educational pearl: A nanopass framework for compiler education. *J. Funct. Program.*, 15(5):653–667.

Shipilev, A. (2011). Jvm anatomy park #10: String.intern().

Shipilev, A. (2016). OpenJDK JMH Project.

Shivers, O. (1988). Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press.

Smaragdakis, Y., Bravenboer, M., and Lhoták, O. (2011). Pick your contexts well: understanding object-sensitivity. In Ball, T. and Sagiv, M., editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 17–30. ACM.

Smaragdakis, Y., Kastrinis, G., and Balatsouras, G. (2014). Introspective analysis: context-sensitivity, across the board. In O'Boyle, M. F. P. and Pingali, K., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 50. ACM.

Sridharan, M. and Bodík, R. (2006). Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA. ACM Press.

Ureche, V., Stojanovic, M., Beguet, R., Stucki, N., and Odersky, M. (2015). Improving the interoperation between generics translations. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 113–124, New York, NY, USA. ACM.

Ureche, V., Talau, C., and Odersky, M. (2013). Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *ACM SIGPLAN Notices*, volume 48, pages 73–92. ACM.

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

Weijiang, Y., Balakrishna, S., Liu, J., and Kulkarni, M. (2015). 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. ACM.

Xu, G. and Rountev, A. (2008). Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 225–236, New York, NY, USA. ACM.

Xu, G., Rountev, A., and Sridharan, M. (2009). Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In Drossopoulou, S., editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 98–122. Springer.

Yan, D., Xu, G. H., and Rountev, A. (2011). Demand-driven context-sensitive alias analysis for Java. In Dwyer, M. B. and Tip, F., editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 155–165. ACM.

Dmytro Petrashko
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
dmitry.petrashko@gmail.com

INTERESTS

Static analysis;
Compiler construction;
Developer productivity;
Programming language theory and implementation;
I/O efficient algorithms.

HIGHLIGHTS

Designed and implemented compiler middle-end and backend for Dotty compiler, the future Scala compiler, together with Martin Odersky. Designed abstractions for the compiler that substantially lowered maintenance cost as well as barriers to entry for new contributors while at the same time reducing compilation time.

Contributed 800+ commits, 160k+ lines of code to Dotty, boostrapped the compiler and fixed 200+ issues. Introduced an extensive self-verification system into compiler that allows to discover and localize bugs easily.

EXPERIENCE

Doctoral Assistant, *École Polytechnique Fédérale de Lausanne*
September 2013 - Present, Lausanne, Switzerland
Working on evolution of *Scala* (http://www.scala-lang.org/), an object-oriented functional programming and scripting statically typed language, designed to concisely express solutions in an elegant, type-safe and lightweight manner.

Dotty (http://dotty.epfl.ch/) is a compiler for Scala that is being developed by EPFL that is faster, easier to maintain and evolve. On the language level, it simplifies Scala by removing extraneous syntax (e.g. no XML literals), and boiling down Scala's types into a smaller set of more fundamental constructs.

Activities & contributions:

- Co-designed architecture of compiler middle-end and backend;
- Bootstrapped the compiler;
- Tracked down a major performance bottleneck in current Scala compiler that is bad memory locality and long object retention;
- Co-designed and implemented the notion of Mini-Phases, that avoids the bottleneck found in current Scala compiler. Mini-Phases are also a convenient abstraction that allows to express AST transformations in an isolated and maintainable way, while fusing them together in runtime for performance;
- Co-designed and implemented YCheck, extensible self-verification infrastructure of Dotty that is the basis of the continuous integration and testing of the Dotty compiler;
- Co-designed Typed AST(TASTY) – a new interchange format to be used by Scala compilers and tools in Scala ecosystem.
- Implemented many phases of compiler, including: type erasure, recursive call optimization, lazy vals transformation, pattern matching;

ScalaBlitz (https://scala-blitz.github.io/). A data-parallel programming framework that optimizes collection operations and offers superior performance to that provided by the Scala standard library collections, by reducing abstraction overheads and taking advantage of code-patterns that contemporary Java VMs and CPUs can execute efficiently.

Activities & contributions:

- Co-designed and implemented macro-based parallel collections;
- Performed rigorous benchmarking, including low-level assembly benchmarking;
- Obtained performance comparable to hand-tuned code written in C++ that uses Intel Threading Building Blocks library;
- Developed a method that allows applying optimizations available in *ScalaBlitz* without modifying legacy code.

Co-founder, technical lead, *Center of Distance Education*
February 2008 - June 2012, Moscow, Russia

Co-founded a startup together with two professors from Moscow Institute of Physics and Technology. A startup around a distributed system for performing big-scale near-realtime video broadcasting. The intended user-base are students that plan to take high school exit exams and want to get tutoring from best teachers available in university.

Most notorious event had to do with one of our big video broadcasts to around 18000 students (1 Mbit/second per student on average. More at peak times that happen at the same time for all students). Our load triggered connectivity issues in several data-centers that ignored our warnings that were sent weeks upfront. This was a good trial of our fail-over mechanism that worked perfectly, hiding the issue from users.

Responsibilities & activities:

- Designing a high-throughput distributed system from scratch with hard requirements on user-experience and failover times;
- Optimizing the system to reduce operational costs;
- Hiring people to perform various tasks for project, including forming new teams of developers and tracking their progress as well as training them to use novel technology;
- Performing long-term technical planning and participating and evaluating long-term technical opportunities for the business;
- Making sure that system can run under high load safely if I'm on an multi-hour exam and team has knowledge how to react in case of failures in my absence.

Project Lead, *Moscow Institute of Open Education*
June 2012 - July 2014, Moscow, Russia

All Russian students take subject exams at the same day after finishing high-school. I was leading governmental project to migrate those exams from paper to an automatic web-based system that would severely reduce operational costs and time needed to check the exams.

The system had to be easy to use both for students as well as people checking the submissions. Semi-automatic graders were provided to ease the work of people evaluating the solutions such as pre-grading and custom techniques used to assign similar solutions to the same graders.

Responsibilities & activities:

- Gathering and analyzing requests from business and governmental customers.
- Taking care of formal standards of private data protection and data retention. Preparing system for governmental certification;
- Developing project architecture and documentation, based on orchestration of multiple cloud systems(Amazon AWS and MS Azure) to support project server architecture during high-load;
- Collaborating with other teams to integrate statistical intrusion detection system and reporting to track causality in the running production system;
- Managing a team of 5 developers.

Software Developer intern, *Wikimart.ru*
February 2012 - July 2012, Moscow, Russia

The Wikimart is a Amazon-like system where users can look for products offered by Wikimart. The most common type of query was a range query, e.g. a query on product price. The underlying system used Cassadra, where those queries are executed very inefficiently, requiring a full scan of stored data. Most known algorithms that improve execution time of such queries require use of locks and hence are inefficient in distributed systems. On the contrary, Fenwick trees does not require forced synchronization and provide eventual consistency guaranties with logarithmic time per operation.

Responsibilities & activities:
- Analyzing production system to isolate a bottleneck in performance;
- Designed a novel algorithm for range queries, implemented and deployed it. Which led to reduction of the average response time from 300ms to 20ms, while 99-percentile decreased from 1500ms to 300ms.

Researcher, *Keldysh Institute of Applied Mathematics*
June 2011 - July 2013, Moscow, Russia

Responsibilities & activities:
- Modification of Treibers Intelligent Driver Model for multiple number of road lanes, training it on the transport flow of Moscow and then applying it to analyze the behavior on the Moscow Ring Road;
- Development of the practical algorithm that finds the shortest path with specified accuracy in graphs with the known dynamics of edge changes, e.g. the graph obtained from the trained Treibers Intelligent Driver Model. This algorithm is a modification of Dijkstras algorithm in the external memory, with ALT-modification and NaturalCuts heuristics.

SCIENTIFIC PUBLICATIONS

- Petrashko D., Lhoták O., Odersky M. “Miniphases: Compilation using Modular and Efficient Tree Transformations”. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2016;
- Petrashko D., Lhoták O., Ureche V., Odersky M. “Call Graphs for Languages with Parametric Polymorphism”. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2016;
- Odersky M., Martres G., Petrashko D.“Implementing Higher-Kinded Types in Dotty”, *Scala Symposium 2016*, October 3031, 2016, Amsterdam, P. 51-60;
- Prokopec A., Petrashko D., Odersky M. “Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections.” *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on.* IEEE;
- Petrashko D. “Investigation on transport flow behavior depending on safe distance”(in Russian) *54th Moscow Institute of Physics and Technology conference:*

Problems of fundamental, applied and technical sciences in contemporary society, 2012, Russia, Moscow, P. 99-103;

- Gasnikov A., Dorn Y., Ivkin N., Ishmanov M., Obidina T., Petrashko D, Holodov Y., Hohlov M., Chehovich Y. "Some actual problems of traffic flow mathematical modeling"(in Russian) *Intelligent Information Processing of the 9th International Conference*, IIP-2012, Montenegro, Budva, P. 211-214;
- Gasnikov A., Gasnikova E., Petrashko D. "Macro-system approach to web-page ranking models"(in Russian); *Information Technology and Systems conference*, 2012, Russia, Petrozavodsk.

TEACHING EXPERIENCE

- **Teaching assistant**. Advanced Compiler Construction, Spring 2016.
  École Polytechnique Fédérale de Lausanne
- **Teaching assistant**. Functional programming, Winter 2015.
  École Polytechnique Fédérale de Lausanne
- **Teaching assistant**. Advanced Compiler Construction, Spring 2015.
  École Polytechnique Fédérale de Lausanne

SUPERVISED STUDENTS

At EPFL, research groups offer substantial projects for B.Sc./M.Sc. students to complete for credit. EPFL PhD students design and supervise these projects, as well as M.Sc. thesis projects.

- M.Sc. project by Renucci A. "AutoCollections" 2016;
- B.Sc. project by Peterssen A. "Delaying arrays: efficient immutable arrays" 2016;
- M.Sc. project by Renucci A. "Common Subexpression Elimination in Dotty" 2015;
- M.Sc. project by Sikiaridis A. " Implementing Method Type Specialisation in Dotty" 2015;
- M.Sc. thesis by Martres G. "Implementing value classes in Dotty, a compiler for Scala". 2015;
- M.Sc. project by Martres G. Co-supervised with Nada Amin. "Investigating subtyping in Dotty". 2014;
- M.Sc. project by Angel A. "BlitzViews: parallel macro-generated lazy collections". 2014.

OPEN SOURCE PROJECTS

Dotty (https://github.com/lampepfl/dotty) Dotty is a platform to try out new language concepts and compiler technologies for Scala.

ScalaBlitz (https://scala-blitz.github.io/). A data-parallel programming framework that optimizes collection operations.

SELECTED CONFERENCE TALKS

- D. Petrashko " Dotty is coming: how to prepare for migration", Scala Days 2017, Chicago, USA , April 18th-21st, 2017;
- D. Petrashko "What should every (Dotty) developer know about hardware", Scala eXchange 2016, London, UK , December 8th-9th, 2016;
- D. Petrashko "How do we make the Dotty compiler fast", Invited talk, Virtual Machine Meetup 2016, Lugano, Switzerland , September 1st-2nd, 2016;
- D. Petrashko "How do we make the Dotty compiler fast", JVM Language summit 2016 organized by Oracle Corporation, Santa Clara, August 1st-4th, 2016;
- D. Petrashko "Dotty Linker: Precise Types Bring Performance", ScalaDays 2016, New York, May 9th-13th, 2016;
- D. Petrashko "Dotty Linker: Precise Types Bring Performance", ScalaDays 2016, Berlin, June 13th-17th, 2016;

- D. Petrashko “Scala & Dotty current status”, invited keynote, ScalaUA 2016, Kiev, April 8th, 2016;
- D. Petrashko “Making sense of initialization order in Scala”, invited keynote, Scalar 2016, Warsaw, April 16th, 2016;
- D. Petrashko “AutoSpecialization in Dotty”, FlatMap(2016) a functional programming conference, Oslo, Norway, May 2nd-3rd, 2016;
- D. Petrashko “From Scala to Dotty”(in Russian), invited keynote, Scala Meetup , Kiev, December 30th, 2015;
- D. Petrashko “Whats new in Dotty”, Fby.by: functional conference of Belarus, Minsk, Nov 28, 2015;
- D. Petrashko “Dotty: Exploring the future of Scala”, invited, ScalaWorld Lake District, UK, 2015;
- D. Petrashko “Making your Scala applications smaller and faster with the Dotty linker“, Scaladays, Amsterdam, Jun 8-10, 2015;
- D. Petrashko “Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections“, 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing, Turku, Finland, March 4-6, 2015;
- D. Petrashko “Lightning-Fast Standard Collections With ScalaBlitz”, Scala Days, Berlin, Jun 16-18, 2014;
- A. Prokopec, D. Petrashko “Macro-based Scala Parallel Collections”, Scala eXchange, London, Dec 2-3, 2013.