ANNE

Contents

List of Figures

Acknowledgements

I would first like to thank my advisor, Prof. Martin Odersky, for giving me the chance to be part of his amazing team. His advice and support during these years have proven invaluable, and he was and still is an inspiration for me.

I would like to thank the members of my jury, Karl Aberer, Andrew Kennedy, Sebastian Hack and Viktor Kuncak for taking the time to read my thesis and all the insightful comments that helped me improve it.

Michel Schinz and Philippe Altherr have helped me tremendously as I was making my first steps in the Scala compiler, and as an assistant at EPFL. I would like to thank as well to the initial Lamp team: Vincent Cremet, Burak Emir, Stephane Micheloud, Nikolay Mihaylov and Erik Stenman. Working with them was a pleasure, and I learned a great deal of things from each of them.

During my PhD studies I have crossed two generations of students at Lamp, and the new team proved to be equally exceptional. I would like to thank Adriaan Moors for all the great conversations over coffee, beer or just a walk. I would like to thank him, Tiark Rompf and Antonio Cunei for carefully proofreading parts of my thesis. Gilles Dubochet, Miguel Garcia, Philipp Haller, Ingo Maier, Donna Malayeri, Hubert Plociniczak and Lukas Rytz made my time in the lab very enjoyable, and provided great company for lunch and coffee breaks, where interesting conversations always emerged.

I would like to thank my girlfriend Tanja for all her help, support and encouragement. She made even the most intense months of thesis writing be enjoyable. Finally, I can never thank enough my parents for the love and trust they provided during my entire life. This work is dedicated to them.

Astratto

Scala è un nuovo linguaggio di programmazione che unisce la programmazione orientata agli oggetti con la programmazione funzionale. Le sue caratteristiche principali sono l'uniformità e l'estensibilità. Scala offre ai programmatori un grande livello di flessibilità, permettendo loro di accrescere il linguaggio utilizzando librerie. Ciò che sembra essere una caratteristica del linguaggio, è in effetti spesso implementato in una libreria, dando in concreto ai programmatori delle possibilità solitamente riservate ai progettisti di linguaggi.

Il lato negativo di tale flessibilità, peraltro, è che codice dall'aspetto familiare può nascondere costi inaspettati in termini di prestazioni. È importante quindi che i compilatori per il linguaggio Scala riducano tali costi il più possibile.

Abbiamo identificato le seguenti aree in cui le prestazioni dei programmi scritti in Scala possono risentire di effetti significativi: funzioni di ordine superiore e chiusure, e contenitori generici usati con tipi primitivi. Presentiamo due approcci complementari per il miglioramento delle prestazioni in tali aree: *ottimizzazione* e *specializzazione*.

L'ottimizzazione nel compilatore può ridurre i costi, grazie ad una combinazione di espansione in linea condotta in modo aggressivo, di una versione estesa della propagazione delle copie, e dell'eliminazione del codice non raggiungibile. Utilizzando tale approccio, è possibile eliminare sia le funzioni anonime che la conversione di tipi primitivi in oggetti. Usando macchine virtuali disponibili allo stato attuale, possiamo mostrare come nel caso di numerosi benchmark tali costrutti possano essere eseguiti fino a cinque volte più rapidamente usando una opportuna ottimizzazione.

La nostra proposta consiste in un nuovo approccio per la compilazione del polimorfismo parametrico nel caso vengano utilizzati tipi primitivi, unendo ad uno schema di traduzione omogeneo una specializzazione del codice per tipi primitivi configurata dall'utente. I parametri di tipo possono essere contrassegnati da annotazioni in modo da ottenere una consequente specializzazione del codice in uso. La nostra proposta utilizza una specializzazione applicata al sito di definizione, il che rende possibile una compilazione modulare, nonché l'assenza di conversione di tipi primitivi in oggetti quando sia il sito di definizione che il sito di utilizzo siano specializzati. Le classi specializzate sono compatibili con il codice non specializzato, ed il codice non a conoscenza della specializzazione può operare su istanze specializzate; ciò significa che la spe-

cializzazione è *opportunistica*. Inoltre, presentiamo la formalizzazione di un sottoinsieme ristretto di Scala che utilizza la specializzazione, e dimostriamo che la specializzazione preserva i tipi. Abbiamo implementato tale trasformazione nell'ambito del compilatore per il linguaggio Scala; ne descriviamo i miglioramenti che ne derivano utilizzando diversi benchmark, mostrando come la specializzazione possa rendere i programmi più veloci di oltre due volte.

Parole chiave: compilatore, ottimizzazione, polimorfismo parametrico, generici, specializzazione, boxing, Scala.

Abstract

Scala is a new programming language bringing together object-oriented and functional programming. Its defining features are uniformity and extensibility. Scala offers great flexibility for programmers, allowing them to grow the language through libraries. Oftentimes what seems like a language feature is in fact implemented in a library, effectively giving programmers the power of language designers.

The downside of this flexibility is that familiar looking code may hide unexpected performance costs. It is important for Scala compilers to bring down this cost as much as possible.

We identify several areas of impact for Scala performance: higher-order functions and closures, and generic containers used with primitive types. We present two complementary approaches for improving performance in these areas: *optimizations* and *specialization*.

Compiler optimization can bring down the cost through a combination of aggressive inlining of higher-order functions, an extended version of copy-propagation and dead-code elimination. Both anonymous functions and boxing can be eliminated by this approach. We show on a number of benchmarks that these language features can be up to 5 times faster when properly optimized, on current day JVMs.

We propose a new approach to compiling parametric polymorphism for performance at primitive types. We mix a homogeneous translation scheme with user-directed specialization for primitive types. Type parameters may be annotated to require specialization of code depending on them. We propose definition-site specialization for primitive types, achieving separate compilation and no boxing when both the definition and call site are specialized. Specialized classes are compatible with unspecialized code, and specialization agnostic code can work with specialized instances, meaning that specialization is *opportunistic*. We present a formalism of a small subset of Scala with specialization and prove that specialization preserves types. We implemented this translation in the Scala compiler and report on improvements on a set of benchmarks, showing that specialization can make programs more than two times faster.

Keywords: compiler, optimization, parametric polymorphism, generics, specialization, boxing, Scala.

Chapter 1

Introduction

Compiler optimizations can significantly improve Scala code when running on the Java Virtual Machine. Java compilers generally rely on the JVM for optimizations, and that proved a viable approach in practice. However, Scala (and any language that is sufficiently different from Java) should not rely entirely on VM optimizations, as there are important benefits to be gained from static optimizations. In this thesis we show how two features of Scala can be made significantly faster through static compiler optimizations. Higher-order functions and closures can be optimized together to yield much better performance when run on a state of the art JVM. Generic code instantiated with primitive types usually incurs a penalty due to boxing and unboxing, and we show a technique that can eliminate the overhead by specializing (some) generic definitions.

Virtual machines are known for more than 40 years, dating back to the 60s. As Java and .NET became popular in the late 90s, virtual machines (VMs) became mainstream [37, 6]. Virtual machines are very attractive because compilers get simpler and programs become more portable. The VM abstracts over the target architecture and provides a high-level instruction set. In fact, VMs offer numerous advantages over the traditional model, relieving compiler writers of issues such as memory management, garbage collection, register allocation, scheduling, the choice of an object-model. Moreover, better monitoring and debugging tools are generally available, and they can be reused for different languages targeting the same platform.

VMs have moved from the initial interpreted model to more efficient, just-in-time compiled execution [6]. VMs are well-positioned to optimize code, as they can gather information during program execution and identify *hot-spots*, code paths that are executed very frequently. By focusing on the hot-spots, the VM can spend more time producing better native code, resulting in better performance as long as the cost of compilation is amortized by running the application long enough. Optimizations such as value numbering, common subexpression elimination, constant propagation/folding, inlining, are regularly performed by nowadays VMs [29, 44, 5].

The optimizations performed by VMs are limited by several factors. Firstly, optimizations are performed while the application is running, so they need to be fast. Secondly, they have to remain general enough to accommodate many languages (over 30 languages for the JVM, and over 60 for .NET [59, 58]). And lastly, VMs can not optimize for all features found in languages that target them, so they aim for the most common subset.

Even when compiling for a virtual machine, optimizing compilers can significantly improve performance. Compilers have more knowledge about the program and the language they are compiling, and they can spend more time optimizing. For instance, a richer type system may point out opportunities for inlining. Furthermore, they may choose different compilation techniques when translating language features that have no direct correspondence on the VM. For example, the JVM does not have support for generic classes. A compiler for a language with generics may choose to compile them using type erasure when performance is not critical, and using specialization when it is.

In this work we focus on compiling the Scala language and improving performance when running on the Java Virtual Machine. One of the features of Scala is higher-order functions. Higher-order functions are heavily used throughout the standard library, and all collections implement a `foreach` method for iterating through their elements

```
items.foreach(x => println(x))
```

The argument to `foreach` is a function literal, which is compiled to an object that implements a known interface. The object may "capture" variables from the environment where `foreach` is applied. While general and elegant, this way of structuring programs loses in terms of performance when compared to a simple for-loop. When both the higher-order function and its argument are known statically, they should be optimized together and completely eliminate the additional object.

Generic classes and methods in Scala may be instantiated with primitive types. Since the JVM does not have support for generics, Scala has to translate such classes and methods to plain, monomorphic definitions. It does so using type erasure, which requires primitive values to be *boxed* and *unboxed* as they enter and leave generic code. The extra indirection and object allocation seriously impact performance of using primitive types with generic code.

To improve performance in such cases, this thesis proposes two compiler techniques: optimization and specialization. The former tackles higher-order functions and the overhead of closures when the caller and callee are statically known. The latter tackles the overhead incurred by boxing primitive values when passed to generic code.

1.1 The Scala language

Scala fuses object-oriented and functional programming in a type-safe way [40]. From the object-oriented world, Scala takes the concept of class, and follows

the principle that "everything is an object". From the functional world, it brings in algebraic data types, pattern matching, anonymous functions and closures. Staying true to the above principle, algebraic data types are encoded as classes (*case classes*), pattern matches as partial functions or extractor objects, functions as generic interfaces of one method and finally closures as objects implementing a function interface[1].

Scala was designed to be interoperable with Java and the .NET platform [1]. Owing to the design of the two platforms, it distinguishes behind the scenes between *primitive* types (such as `Int`, `Long`, `Double`) and *reference* types. This distinction is necessary for having efficient execution of arithmetic primitives on the two platforms. However, to the programmer all values appear as objects, and the compiler adds the necessary *boxing* operations to turn a primitive value into a heap-allocated object when necessary. As described in more detail in Chapter 2, this can be a source of important performance degradation.

Despite having primitive types, Scala has a unified type hierarchy. The top type is called `Any`, with immediate subtypes `AnyRef`, the supertype of all reference types, and `AnyVal`, the supertype of all primitive types. Parametric definitions can be instantiated with any type, including primitives. Scala compiles generics through type-erasure, and primitive values need to be boxed when entering generic code.

Classes and methods can be parameterized with types which may have bounds. Furthermore, type parameters can be annotated as being *covariant* or *contravariant*, allowing to extend the subtyping relationship from the type argument to the instantiated type. For example, the interface for unary functions is defined like this:

```
trait Function1[+R, -T] {
  def apply(x: T): R
}
```

Function subtyping has the usual definition, covariant in the return type and contravariant in the arguments, denoted by the `+` and `-` respectively. For instance, `Function1[String, Any]` is a subtype of `Function1[Any, String]`. In order to preserve soundness, covariant and contravariant type parameters have restrictions on where they may appear in a class definition [40].

One of the design goals of Scala is to be extensible. Programmers can easily embed Domain Specific Languages (DSLs) in Scala, relying on features such as *implicits* and *call-by-name* parameters. For example, the following code to test a stack class is written using the ScalaTest [57] framework. None of the "keywords" is part of the Scala language:

```
"A Stack" should "pop values in last-in-first-out order" in {
  val stack = new Stack[Int]
  stack.push(1)
  stack.push(2)
```

[1]Closures capture the environment as usual. The environment is represented as fields of the anonymous class from which the closure object is instantiated.

```
    stack.pop() should equal (2)
    stack.pop() should equal (1)
  }
```

In Section 2.3 we review the most important tools in the Scala programmer's bag of tricks.

1.2 Contributions

This thesis identifies several features of Scala that can be improved when compiling for the JVM, and concentrates on closures and generic code instantiated with primitive types. Higher-order methods and closures are pervasive in Scala programs, and essential when implementing DSLs because call-by-name parameters are encoded as nullary functions. Generic classes and interfaces are used throughout the standard library and form the backbone of a uniform collection library. When they are used with primitive types, performance suffers.

We describe, implement and evaluate two complementary techniques that improve the performance of Scala programs.

The main contributions of this thesis are:

- We describe and implement a number of static compiler optimizations that can improve the cost of using higher-order functions and closures. Our solution works with separately compiled libraries, and improves performance significantly when running on the JVM.
- We propose a translation technique for generic classes that gives good performance for primitive values. Programmers may choose to specialize generic code when performance is critical. Our approach is compatible with the current translation technique (using type erasure), and supports separate compilation and variance.
- We give a formal description of specialization using a Featherweight Java-like calculus, and prove that our translation preserves typing.
- We evaluate both techniques on a set of benchmark programs and report on performance improvements.

Both optimization and specialization are part of the current release of the Scala compiler (version 2.8.0). Some late developments were added after the official release and are to be included in the next release.

1.3 Overview

This thesis is organized as follows: Chapter 2 discusses in detail the performance implications of various Scala features, and gives a glimpse of the techniques we employ to bring down their cost.

Chapter 3 is dedicated to compiler optimizations, and describes the infrastructure for optimizations found in the Scala compiler, together with a number of optimizations.

Chapter 4 describes in detail our approach for compiling generic code for performance on primitive types, which we name *opportunistic specialization*. We formalize our translation for a subset of Scala, and discuss the implementation challenges when supporting the whole language.

We used the current implementation to validate our techniques in Chapter 5. We show how optimization and specialization can improve the execution time of several benchmarks, and asses the impact on code size.

We discuss related work in Chapter 6, and conclude and provide directions for future work in Chapter 7.

Chapter 2

Language Extensibility and Performance

> If I could speak the language of rabbits, they would be amazed, and I would be their king.
>
> Rajesh Koothrappali

2.1 Introduction

The ability to grow the language through libraries is a key aspect of Scala. Its syntax allows users to write code that looks like built-in features, keeping the language small. For instance, the standard library provides a `BigInt` class that is indistinguishable from the standard `Int` type, and the `for` loop on integers is provided through a `Range` class.

This approach is elegant and gives programmers the power of language designers. However, everything comes at a price, and in this case the price is efficiency. Familiar looking code, like an assert statement or a for loop may conceal unexpected costs. While library designers are usually aware of these implications, users are often surprised by such performance hits.

In this chapter we present several features of Scala that make it especially suited for building expressive libraries. We then take a closer look at their performance implications and sketch an approach for optimizing compilers that can improve on their cost.

2.2 Domain Specific Languages

Programming languages research has traditionally focused on general purpose programming languages which aim to be good at solving any programming

```
def Term: Parser = AbsOrVar ~ rep(AbsOrVar)
def AbsOrVar: Parser =
    ident
  | "\\" ~ ident ~ "." ~ Term
  | "(" ~ Term ~ ")"
  | "fix" ~ Term
  | failure("illegal start of ..")
```

Figure 2.1: A parser using the combinator parser DSL.

task. Decades of research have given us numerous languages and several paradigms for solving general programming problems. Recently there is a shift of focus towards Domain Specific Languages (DSL), languages suitable for solving one specific task. They are usually small, simple and use a notation that is familiar to their domain. Specialists in the domain of interest can easily understand and write programs in a DSL. Examples of DSLs are lex and yacc for describing grammars, Excel for describing spreadsheets, or SQL for database queries and updates.

According to Taha [52], DSLs can be defined by four characteristics:

- The domain is well-defined
- The notation is clear
- The informal meaning is clear
- The formal meaning is clear and implemented

The first three points are more a concern of the language designer, who is in a position to make decisions about the domain, notation and the meaning of the language. In the following we focus on the last point, where we believe Scala makes a notable contribution in the implementation of DSLs.

Just like traditional programming languages, one can implement a DSL starting with a grammar, writing a parser and moving on to a full-blown interpreter. The downside is that building a full interpreter is time-consuming and makes the evolution of DSLs costly. In [24], Hudak argues for Domain Specific *Embedded* Languages, DSLs that are built on top of an existing, general purpose language. DSELs are implemented as libraries, and share the syntax and implementation of the host language.

2.2.1 DSLs in Scala

Scala is well suited for building DSLs by allowing syntactic extensions in an easy and natural way. For example, Figure 2.1 shows how to specify a parser using the standard combinator parsing library DSL:

This example shows the definition of two non-terminals of the lambda calculus grammar. Terms consist of one or more abstractions or variables. `AbsOrVar` is either an identifier, a lambda abstraction, a parenthesized term or the keyword "fix" followed by a term. If none of the above is found, an error is issued and parsing fails. It is important to note that this code has no special support from the compiler, all operators being user-defined.

The two operators at work, `~` and `|`, are user-defined methods on class `Parser`. Any unary method in Scala can be used in infix notation by skipping the dot and the parenthesis around its argument. The given string literals denote keywords, and are turned by *implicit conversions* into parsers matching that exact string.

2.3 The Scala toolbox

In the following sections we describe a few features of Scala that make it suitable for developing embedded DSLs, the cost they carry and how an optimizing compiler may alleviate it.

2.3.1 Higher-order functions

Scala supports higher-order functions and has convenient syntax for function literals. For instance, method `foreach` is defined like this:

```
def foreach[U](f: A => U) = // ..
```

Here, `foreach` takes a function from type `A` (we assume `foreach` is defined inside a generic collection of elements of type `A`) to `U` (most of the times `U` is instantiated to `Unit`). Iterating over such a collection is then done like this

```
xs foreach { x =>
  // x is bound successively to each element in xs
}
```

Notice how infix notation makes `foreach` look like a built-in feature of the language (which it is, for C#, Ada, Java, D, JavaScript).

The ability to extend the language with new control structures demands a way to delay evaluation of terms. To this end, Scala provides call-by-name parameters, which allow programs to pass unevaluated arguments to a method. Such arguments are evaluated each time their value is needed. Behind the scenes, the compiler transforms such arguments into nullary functions. For example, we may define an `assert` method whose diagnostic message is only evaluated if the assertion fails:

```
def assert(b: Boolean, msg: => String) =
  if (!b) throw new AssertionFailed(msg)

//..
assert(tree.depth < 6,
```

```
  "Too shallow: " + (new TreePrinter(tree)).toString)
```

2.3.2 Implicit parameters and views

When designing a library it often happens that an existing type has to be augmented with new methods. For instance, a DSL may want to reuse the primitive values of the host language, but specific functionality is (naturally) missing on those types. To enable after-the-fact extension of existing types, Scala proposes a mechanism based on implicit values and *views*.

A parameter marked **implicit** can be filled automatically by the compiler when the programmer does not provide an explicit value. All values in scope that are marked **implicit** are eligible. Chapter 7 in the Scala specification [40] provides a complete description of how the scope is formed and how a value is chosen.

Views allow the user to define implicit conversions between a type `T` and a type `U`, by defining an implicit value `v` of type `T => U`. Whenever the expected type of an expression is `U`, and the static type is `T`, the compiler will convert it by applying `v` to the expression. This leads to a convenient way of augmenting functionality of existing types, including primitive types. In the parser example in Figure 2.1, string literals are lifted to full-blown parsers by a defining a view from `String` to `Parser`.

```
implicit def kw(str: String): Parser[String] =
  accept(str) // ...
```

A view is also applied when a member is selected on a type that does not define it. In the same example, method `~` is not a member of `String`, so the compiler looks for a view to a type that has a member named `~`. Views facilitate a pattern for extending existing classes, used throughout the library for augmenting primitive types. DSLs may use them to lift primitive types in the host language to values in their domain. For example, the standard library adds method `abs` to integers by an implicit conversion to a "carrier" class for the additional operations:

```
final class RichInt(val n: Int) {
  def abs: Int = if (n < 0) -n else n
  // additional operations may be defined
}

implicit def intWrapper(x: Int) =
  new runtime.RichInt(x)

// users can then write
x.abs
```

2.3.3 For comprehensions

Scala provides an extensible way to iterate over collections by means of *for comprehensions*. A **for** expression is translated to a sequence of method calls to `foreach`, `map` and `withFilter`.

In its most general form a for-comprehension contains any number of *generators* and *filters*. For instance,

```
for (i <- xs; j <- ys; if (i % j == 0)) print (i, j)
```

prints i and j only when i is a multiple of j. The first two statements are generators, binding i and j to each element of xs and ys respectively. This is achieved by translating the given comprehension into a series of method calls:

```
xs.foreach(i =>
  ys.withFilter(j => i % j == 0).foreach(j => print(i, j)))
```

Any type that has these methods can be used as a generator inside a for comprehension. Even more, Scala does not provide a for loop as in most imperative programming languages, instead it has a `Range` class in the standard library with the required methods. However, to the programmer it looks like the language has built-in support for iterating over integers.

```
for (i <- 1 to 10) print(i)
```

2.4 Uniform collection libraries

Uniformity is one of the bases on which Scala is built: class members are accessed using the same notation, regardless of whether they are stored or computed (sometimes called the "uniform access principle"); classes, methods and types may be parameterized by type and value; any definition may be nested, and may contain any other definition; parameterized types may be instantiated with any type, including primitive types or type variables; every term produces a value. It should come as no surprise that the standard library follows the same philosophy.

The Scala collections library [39] provides implementations for commonly used data structures. It is based on the very general concept of `Traversable` structures, further divided in maps, sets and sequences. More than a collection of classes, it is a framework where new collections can be added easily, avoiding code duplication.

2.4.1 Java arrays

Along with the eight primitive types, the JVM provides arrays at the machine level. JVM arrays are typed and mapped to contiguous regions in memory, so they provide a very efficient alternative to user-defined collections. Unlike, say, linked-lists, which allocate each element in a separate object, arrays promise

very good cache behavior by exposing data locality. Furthermore, the JVM may apply optimizations such as array bounds check elimination. It follows that arrays are essential for good performance on the JVM, and Scala has to accommodate them.

The question is how to fit arrays in the collection library. Arrays are sequences, and based on the uniformity principle, they should be part of the collection library. Arrays should be polymorphic in the element type, and allow the same operations as the standard `Vector` class. Arrays could look like

```
class Array[A] extends IndexedSeq[A] {
  // ...
}
```

However, the problem we face is that arrays are typed at the bytecode level. An array of integers is different from an array of doubles, and each one has specific VM instructions. All arrays opcodes (instantiation, indexing and updates) require the type of the element, so writing generic code on arrays poses problems. Suppose we wanted to write a method that copies the contents of one array into another:

```
def copy[A](xs: Array[A], ys: Array[A]) {
    var i = 0
    while (i < xs.length) {
      ys(i) = xs(i)
      i += 1
    }
}
```

The code for `copy` needs to read and update an array of an unknown type `A`. The compiler cannot emit the required bytecode, as the array type may change with each call to `copy`. Instead of the specific bytecodes, it has to perform a series of instance tests to determine the runtime type of the array[1]. Unfortunately, the performance of code is seriously hurt, very bad news for a collection that is meant for high performance.

This situation is not specific to arrays, though it does carry the largest impact in this context. Any generic class that may be instantiated at primitive types will perform much worse than similar code that has full type information. A prime example is function literals: function literals in Scala implement a `FunctionN` interface that is polymorphic in its result and argument types. Whenever the function operates at primitive types, boxing (wrapping of primitive types inside object on the heap) and unboxing is performed at call sites (for a more detailed discussion see § 2.5.1).

We propose user-directed specialization as a solution to this problem: the compiler generates specialized code for some type instantiations, as directed by the user. Code is tailored to match the type instantiations, using the efficient bytecode instructions. Furthermore, whenever the static type information at a caller site permits, the call is rewritten towards the specialized version. This

[1] Initially it was using reflection, but that proved to be too slow.

approach is described in Chapter 4.

2.5 The abstraction penalty

In the following we look at the code generated by the Scala compiler, and highlight where performance may be hurt. To illustrate, we'll be looking at a simple for loop that prints integers between 1 and 10. As mentioned before, Scala does not have a specific for-loop, as does Java or C++. Instead, the standard library comes with a class `Range` that defines `foreach`, `filter`, `withFilter`, `map` and `flatMap`. This solution is more general and more extensible than having a special case in the compiler, and allows users can write their own looping constructs. Coming back to the example, integers are lifted to ranges by implicit conversion, so that programmers can simply write:

```
for (i <- 1 to 10) println(i)
```

The call to method `to` (in infix notation) is not resolved, so an implicit conversion from `Int` to a type that has such a member is looked up. As mentioned previously, the standard library provides an implicit conversion, so the code is type-checked as

```
for (i <- Predef.intWrapper(1).to(10)) Predef.println(i)
```

The for loop is desugared to an explicit call to `foreach` and a function literal:

```
Predef.intWrapper(1).to(10).foreach[Unit](((i: Int) => Predef.println(i)))
```

Scala translates closures to anonymous classes. For each function arity there is a corresponding trait that defines the `apply` method with the right number of parameters. In this example, the anonymous class extends trait `Function1`:

```
Predef.intWrapper(1).to(10).foreach[Unit]({
  final class anonfun extends Function1[Unit, Int] with ScalaObject {
    final def apply(i: Int): Unit = Predef.println(i);
  };
  new anonfun()
})
```

but the actual code that is generated after type erasure is

```
Predef.intWrapper(1).to(10).foreach[Unit]({
  final class anonfun extends Function1 with ScalaObject {
    final def apply(i: Int): Unit = Predef.println(i);
     final def apply(i: Object): Object = {
       apply(Int.unbox(i))
       BoxedUnit.UNIT
     }
  };
  new anonfun()
})
```

Through successive transformations, the innocent looking for-loop now requires four method calls[2], two additional loaded classes, two new objects, plus boxing/unboxing. In order to describe what boxing is, we need to talk about compiling generics for the Java Virtual Machine.

2.5.1 Type erasure

The Java Virtual Machine (JVM) does not have support for generic classes. Polymorphism needs to be "compiled away" so that the code presented to the VM has no type parameters. There have been various proposals for adding genericity to the Java language while keeping the current VM design, but all can be roughly divided in two categories: those based on type erasure, and those based on code specialization. Type erasure [11] is based on removing type parameters and keeping a single version of the code, that runs on unmodified JVMs. All type parameters are replaced by type `Object`, the top of the type hierarchy, so generic code always operates on references. The alternative is to generate specialized code for each (or some) type instantiation, removing type parameters from generated code as well. The cost is increased code size (and depending when specialization is performed, class load time) and complexity of implementation[3]. A more detailed discussion follows in Chapter 6.

Similar to Java 1.5, Scala implements generics through type erasure. We will use a few examples to show how to define and instantiate parameterized classes, then how erasure transforms generic code. We begin by defining a generic class for linked lists, similar to the one found in the standard library:

```
abstract class List[+A] {
  def head: A
  def tail: List[A]
  def prepend[B >: A](x: B): List[B]
}
```

This introduces an abstract class for lists which has only three operations: `head`, returning the first element of the list, `tail`, returning a list containing all but the first element, and `prepend`, an operation for obtaining a new list containing the given element followed by the current list. The plus sign in front of the type parameter marks it as covariant, meaning that when `List` is instantiated with types in a subtype relationship, lists of those types are themselves in a subtype relationship. The signature of method `prepend` says that one can prepend values of any type that is a supertype of `A`, and it gets back a list of the less precise type. For instance, one can prepend a plain `Object` to a list of strings, but it gets back a list of objects. This is safe, since strings are objects.

We continue by defining two concrete classes, one for empty lists and one for lists carrying at least one element.

[2] In addition to three visible calls in the example, there is the call to `apply`; and this is without counting constructor calls for the extra objects.

[3] Another concern when Java generics were considered was compatibility with pre-generics code, which is ensured by the type erasure approach.

```
case object Nil extends List[Nothing] {
  def head = error("Head of empty list")
  def tail = error("Tail of empty list")
  def prepend[B](x: B) = new Cons(x, Nil)
}

case class Cons[+A](val head: A, val tail: List[A]) extends List[A] {
  def prepend[B >: A](x: B) = new Cons(x, this)
}
```

The implementation classes are `case classes`, meaning they can be used in pattern matching. `Cons` takes two parameters, head and tail, which are public members and implement the inherited abstract methods.

Lists defined so far are not very useful, so we add a `map` method:

```
class List[+A] {
  // .. same as before
  def map[B](f: A => B): List[B] = this match {
    case Nil => Nil
    case Cons(x, tail) => tail.map(f).prepend(f(x))
  }
}
```

Next we take a look at the code after erasure. As mentioned earlier, the Scala compiler removes type parameters and generates a single version of the code. Type variables are replaced by `Object` (more precisely by their upper bound, but for simplicity this example uses only unbounded type parameters).

```
abstract class List extends java.lang.Object with ScalaObject {
  def head: Object;
  def tail: List;
  def prepend(x: Object): List;
  def map(f: Function1): List = this match {
    case Nil => Nil
    case Cons(x, tail) => tail.map(f).prepend(f.apply(x))
  }
}
```

One can observe how the class definition does not take any type parameters, and wherever `A` appeared before now stands `Object`. The function type taken by `map` is desugared to (erased) trait `Function1`. For brevity we omit the translation of the two concrete classes, and look next at an example that adds one to each element of a given list:

```
val xs: List[Int] = // some list
xs map (x => x + 1)
```

The function is translated to an anonymous function, and after erasure the code looks as shown in Figure 2.2.

Function literals are translated to anonymous functions that carry their code in method `apply`. In this example `map` is instantiated at type `Int`, which is a *primi-*

```
xs.map({
  final class anonfun extends Object with Function1 with ScalaObject {
    final def apply(x: Int): Int = x + 1
    final @bridge def apply(v1: Object): Object = Int.box(apply(Int.unbox(v1)))
  };
  (new anonfun(): Function1)
}
```

Figure 2.2: Map example after erasure.

tive type. For efficiency reasons, the JVM distinguishes between primitive types (numeric and boolean) and reference types, and provides different bytecodes for each of them. Reference types are used exclusively for objects living on the heap, while primitive types have value semantics.

In the type erasure model there is a single version of the generic code. Naturally, that code needs to operate on the most general type possible, `Object`. To accommodate primitive types, the compiler needs to add *boxing* and *unboxing* operations whenever a primitive value is passed or received from generic code. Boxing simply instantiates a carrier object living on the heap, and initializes it to the given value.

Notice how this example has a second `apply` method, marked *bridge*. A bridge method is a synthetic forwarder that implements an abstract method[4] whose signature differs from the signature of the implementing method. The reasons become clear when we look at how `Function1` is erased:

```
trait Function1[+R, -A] {
  def apply(x: A): R
}
```

```
trait Function1 {
  def apply(x: Object): Object
}
```

After erasure, method `apply` is defined in terms of `Object`, but the implementing method in the anonymous class is defined in terms of `Int`. The JVM mandates that an overriding method has to have exactly the same signature as the method that it overrides, so the compiler has to add a bridge method that overrides the inherited method with the original signature, which performs the necessary boxing and then delegates to the actual implementation.

Erasure may also need to add casts around the points where generic code is instantiated. This is because a type parameter is erased to its upper bound (in this example, `Object`), but the instantiation may be at a more specific type (for instance, `List[String]`). Such casts are guaranteed not to fail, but they are needed so that the JVM can verify the emitted code.

Now we can see better what overhead is incurred by a simple `map` call: each element of the list is first **unboxed**, passed to the real `apply` method, incremented and then **boxed** again. An optimizing compiler should eliminate the overhead induced by boxing when it is known statically what closure is instantiated and called. In fact, whenever the anonymous function is never stored nor

[4] The same solution is applied for overriding concrete generic methods.

used more than once, the code should be inlined and the whole class should be optimized away, saving both code size and memory footprint.

2.6 A look ahead

In this section we describe what techniques we use to eliminate the overhead imposed by closures and erasure. A central concern of our design is to allow separate compilation. A more detailed description of the implementation follows in the next chapters, while the coming sections focus on a conceptual description of what is being performed during an optimized compilation run.

2.6.1 Compiler optimizations

Traditional optimizations techniques, like inlining, are not always effective. Code is often spread across several compiled libraries, and many short-lived objects on the heap make analysis harder. Difficulties arise from both control-flow and data-flow imprecision: virtual, non-final methods cannot be resolved statically, and values stored in objects on the heap (such as boxed values) hide their flow.

We put forward a number of optimizations implemented in the Scala compiler: a boxing optimization (that removes unnecessary boxing and unboxing operations), a type-propagation analysis (that obtains more precise types for values on the stack and local variables) and a copy-propagation optimization (that has a simple heap model handling common object patterns like boxed values and closures). All these would be of little use for library code, unless the compiler had a way of analyzing libraries. The Scala compiler is using a *bytecode reader* for reading back compiled code into the intermediate representation on which the optimizer works.

The optimizer is organized as a series of phases that operate on an intermediate representation called ICode. In a first phase, methods are inlined. Then a copy-propagation phase tries to remove all references to the closure object (or environment). A dead-code elimination pass removes unnecessary assignments, mainly coming from the inlining phase, and all unreferenced closure objects (including their class).

Inlining

Automatic inlining, or *procedure integration*, replaces method calls with their bodies [36]. This is an extremely useful optimization, because besides saving the overhead of the call, it allows one to optimize a method in the context of the caller. The additional information gained by this step may statically resolve calls to other methods, allowing further integration.

In Scala, this step is essential for eliminating closures:

```
xs.foreach(x => print(x))
```

Suppose the compiler inlines `foreach` and brings the loop in the context of the caller. The function application inside the loop can now be resolved and replaced by a simple call to print (a second inline step).

Before inlining can take place, method calls have to be *resolved*. A method call is resolved when there is a single possible implementation that could be called. Method resolution has been studied extensively in the context of optimizations for object-oriented languages [50, 22, 14, 8].

The Scala compiler uses a fairly expensive analysis to derive precise types for local variables. The goal of this analysis is to resolve method calls. Although the JVM performs inlining when it compiles methods to native code, it does so only when a method is final or when the receiver class provides the only implementation loaded in the VM [44, 29]. This technique (Class Hierarchy Analysis [14]) is very fast and works well when there is at most one implementation of an interface method. This is not our case of interest: all method calls to `FunctionN.apply` are truly polymorphic, since it is very likely that more than one implementation of such a trait exists in the system (the standard library alone has more than 1200 anonymous functions). Instead, our data flow analysis aims to derive the most precise type possible for the receiver object. It turns out that in most cases this is the anonymous function type, which is final and statically known. Since we are not relying on whole–program analysis, the inliner can inline only final methods. This is not a problem in our main cases of interest, closures: they are always final classes that extend a `FunctionN` interface.

The decision to inline is taken based on method size (calls inside small methods are usually poor choices, as the JVM native compiler already favors small methods). Higher-order methods are preferred for inlining, as they usually allow complete elimination of their argument. In our previous example, inlining method `map` allows, in a next step, to inline the anonymous function given as argument.

The analysis and inlining phase are repeated until a fix point is reached (no more methods can be inlined) or a size limit is reached. This ensures reasonable compilation times and method sizes (as mentioned before, large methods take a penalty hit as the JIT compiler is more reluctant to compile them).

Copy propagation

Copy propagation is a transformation that replaces the use of a variable x with another variable y, as long as there is a previous assignment $x = y$, and the two variables have not been changed between the definition and the point where x is used [36].

This phase is trying to infer what values passed in the closure environment can be accessed from the method environment. In other words, the analysis tells whether fields of the anonymous class can be proved to be the same as some local variable available in the enclosing method. This case is common after inlining both a higher order function (such as `foreach`) and the closure's `apply` method.

Compared to the textbook version of copy propagation, this analysis adds a simple model of the heap: in addition to local variables and stack positions, values of objects on the heap that are reachable from locals or stack are modeled as simple records. Such records are populated when known constructors are invoked. Examples of constructors that can create non-empty records on the heap are closure constructors (which populate the record with the captured environment), case class constructors (which populate the record with the given case arguments) and box methods (which populate the record with a single field, the boxed value).

Once the analysis has determined copy-of relations between locals and values on the heap, it proceeds by replacing them by the cheapest operation. At the end of this phase, boxed and closure objects may be unreferenced, and they will be cleaned up by the next phase.

Dead-code elimination

A variable is *dead* if its value is not used on any control flow path starting at its definition. Instructions that compute values that are never used are also *dead* [36]. Such code is likely to appear as a result of the previous optimization phases.

The last optimization phase is cleaning up dead code. It uses a standard mark and sweep algorithm: in a first phase, useful instructions are marked, then in a second phase all instructions not marked are removed.

To mark useful instructions, the algorithm starts with instructions that are known to be needed, like those that produce the return value of a method, and side-effecting methods. Then, based on reaching definitions, it recursively marks all instructions that create the definitions used by marked instructions. By the end of this phase there might be altogether unreferenced closure classes: they are removed completely and no code is generated for them.

2.6.2 Opportunistic specialization

Type specialization is complementary to optimizations, and targets the overhead resulting from type erasure. The Scala compiler proposes a novel way of specializing generics *on-demand*, by definition-site annotations on type parameters. For instance, the definition of `Function1` could be

```
trait Function1[@specialized R, @specialized A] {
  def apply(x: A): R
}
```

By default, the Scala compiler specializes `Function1` for all combinations of primitive types for `R` and `A`. The generic class is augmented with specialized variants of method `apply`, and a number of additional classes is generated, carrying specialized definitions of `apply`. For example, the type `Int => Int` would look like

```
xs.map({
  final class anonfun extends Object with Function1$IntInt {
    final def apply$IntInt(x: Int): Int = x + 1
    final def apply(v1: Object): Object = Int.box(apply$IntInt(Int.unbox(v1)))
  };
  (new anonfun(): Function1)
}
```

Figure 2.3: Specialized map example

```
trait Function1[+R, -A] {
  def apply(x: A): R
  def apply$IntInt(x: Int): Int
  def apply$IntLong(x: Int): Long
  // all combinations of primitive types
}
```

```
trait Function1$IntInt {
  def apply$IntInt(x: Int): Int
  def apply(x: Object): Object =
    Int.box(
      apply$IntInt(Int.unbox(x)))
}
```

This specialized interface offers now two methods that "do the same thing": one has a specialized signature that does not need boxing and runs at full-speed, and a generic signature that may work with any object, but which requires boxing. The compiler makes sure the implementations are "in-sync", by deriving the specialized version from the user-defined method.

The generic `apply` found in the specialized subclass is similar to the bridge method seen in Section 2.5.1. The difference is that specialized versions of `apply` are promoted to the generic interface, and accessible to code that does not know the exact subclass of `Function1` they are dealing with. This allows the compiler to reroute calls to the specialized variant *without* knowing the precise subclass where it is implemented. The bridging is needed to keep the two methods in-sync.

Users of this interface may now call the specialized method whenever the static type context allows it. Similarly, instantiations of generic classes are opportunistically rewritten to a specialized subclass whenever the static type indicates it is possible. Figure 2.3 shows the `map` example when specialized:

Assuming method `map` is defined in a context where the element type is known to be `Int` (for instance the list class is specialized as well), it will call the specialized version of apply, `apply$IntInt`, skipping boxing/unboxing altogether. However, specialized instances are compatible with unspecialized code, and a generic version of `apply` still exists, and delegates to the specialized version. This allows the compiler to specialize only parts of the program, and supports separate compilation. A complete description of this technique follows in Chapter 4

Chapter 3

Optimizations

> Although personally I am quite content with existing explosives, I feel we must not stand in the path of improvement.
>
> Winston Churchill

3.1 Introduction

Ever since programming languages began the race for more expressiveness and higher abstraction, compiler writers have been fighting to bring their cost down. As shown previously in § 2.5, there is an *abstraction penalty* for using some of the high-level features of Scala. Higher-order functions and boxing are examples of costly, but also highly-used and useful features of the language. Their compilation scheme cannot be improved in the general case, but we identify cases when the full generality is not needed and show how they can be optimized.

Scala targets the Java Virtual Machine[1]. There is an important body of work on dynamic optimizations implemented in the JVM (see § 6.2.3). VMs have evolved considerably from interpreters to high-performance just-in-time (JIT) compiled runtimes [6]. Adaptive optimization relies on identifying *hot-spots* through runtime performance counters, and then directing the JIT compiler to those methods. By focusing on the code that gets executed the most, the VM can spend more time optimizing the code that yields the best payoff.

Some of the optimizations performed by JIT compilers are the "classical" optimizations like value numbering, constant propagation/folding, common subexpression elimination [29, 49, 44, 5]. Scala benefits from these optimizations, and there is little incentive to duplicate them in the Scala compiler. How-

[1] There is a second backend targeting the .NET platform.

ever, the main observation is that such optimizations happen inside a method, and inlining can easily miss opportunities in languages with higher-order functions like Scala. Looking again at the `foreach` example, the VM may decide to optimize it, but looking at the call-site it may not be able to inline the function application, simply because `foreach` is called with many different functions:

```
override def foreach[U](f: A => U): Unit = {
  var i = 0
  val len = length
  while (i < len) { f(this(i)); i += 1 }
}
```

The call to `f` is truly polymorphic, and there may be thousands of implementations of unary functions. To correctly handle such cases one needs the context of the `foreach` call, or a context-sensitive analysis. We opted for inlining, which is simpler and gives good results (see § 3.3.1 for details on the decision procedure).

The JVM was designed mainly for running Java, and consequently the optimizations are geared towards this language. For the first 8 years of its existence, Java did not have parameterized types [11], and higher-order functions are missing still [43]. Both features are essential to Scala and require optimization for good performance, but because of the differences between the two languages (generics can not be instantiated with primitive types in Java) the VM does not tackle them aggressively enough.

We propose a solution based on more aggressive inlining across compilation units, and complete elimination of closure objects whenever their environment can be inlined. As a beneficial side-effect, boxing can be removed altogether while inlining closures. The next sections describe the optimizing phases in the Scala compiler, starting with the intermediate representation, the bytecode reader, then moving to the data-flow analysis infrastructure and finally the optimization and analysis phases.

3.2 Compiler infrastructure

The Scala compiler is organized in a sequence of phases, each one translating the input language into a simpler form, until the program is close enough to Java to make code generation simple. The front-end uses an abstract syntax tree (AST) that is passed between phases, while the backend uses *ICode*, a stack-based intermediate representation similar to Baf [55] and JIR [15].

A summary of the phases is depicted in Figure 3.1. Most of the transformations are done on the AST, and many of the interesting ones, like lambda lifting (constructing environments for free variables in lambda terms) and mixin (mixin composition, a form of multiple inheritance based on traits [46, 40]) are performed after type erasure. The *ICode* phase translates the AST to ICode, which is then used for several optimization phases, and code generation. The

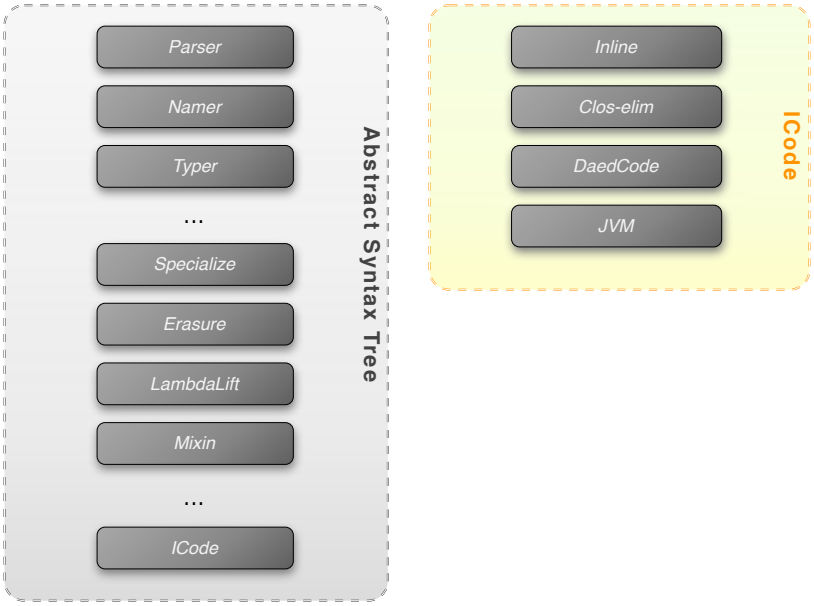


Figure 3.1: Scala compiler phases and intermediate representations

last phase, *JVM*, can be replaced by another backend. Currently, there is a second backend which targets the .NET platform.

3.2.1 Intermediate representation

ICode is a relatively standard control-flow graph based intermediate representation [36]. It is designed to be close to Java bytecode in order to facilitate bytecode parsing, but still be suitable for other target architectures, like the .NET platform. ICode is best introduced through an example:

```
val xs: List[Int]
def sum(start: Int, end: Int) = {
  var i = start
  var sum = 0
  while (i < end) {
    sum += xs(i)
    i += 1
  }
}
```

This method is defined inside a class, but for clarity we ignore the details around `sum`. The code loops on the elements of a `List` of integers and sums them up.

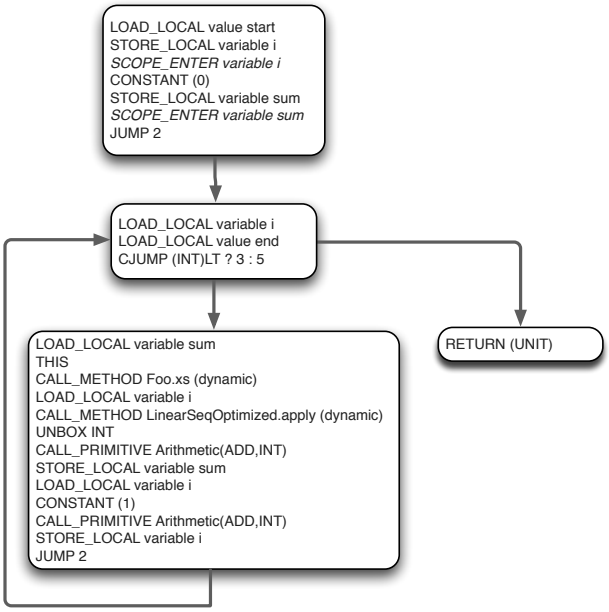


Figure 3.2: The control-flow graph of the `sum` method

The ICode representation is shown in Figure 3.2. Each basic block is a sequence of instructions, and there is only one entry point, at the beginning of the block. Each instruction is executed in sequence, the last instruction is a control-flow instruction (jump, return or switch), and edges connect blocks with their successors. Blocks are referenced by their label, which is simply an integer value.

Similar to Java bytecode, ICode has both local variables (accessed through `LOAD_LOCAL` and `STORE_LOCAL`, and referred by their symbolic name) and an operand stack. This decision was taken to ease parsing compiled libraries and avoid the problem of generating good stack-based code from register-based code (see Raja et al for a discussion of this issue [55]). Furthermore, the relatively simple optimizations we implement would not justify the effort and cost of a more advanced IR like SSA.

Instructions are pretty straight forward, and most of them take their operands from the stack, where they store back their result. The `SCOPE_` instructions are special and are discussed later in this section. All instructions are typed, for instance `CJUMP` is a conditional jump on an integer, using the top of the stack to decide where to jump. Primitives are multiplexed through `CALL_PRIMITIVE`, which is parameterized with the type of the operands and the actual operation to perform. One last thing to note in this example is the use of `UNBOX`, which is a primitive in ICode, taking the top of the stack and unboxing it to the specified

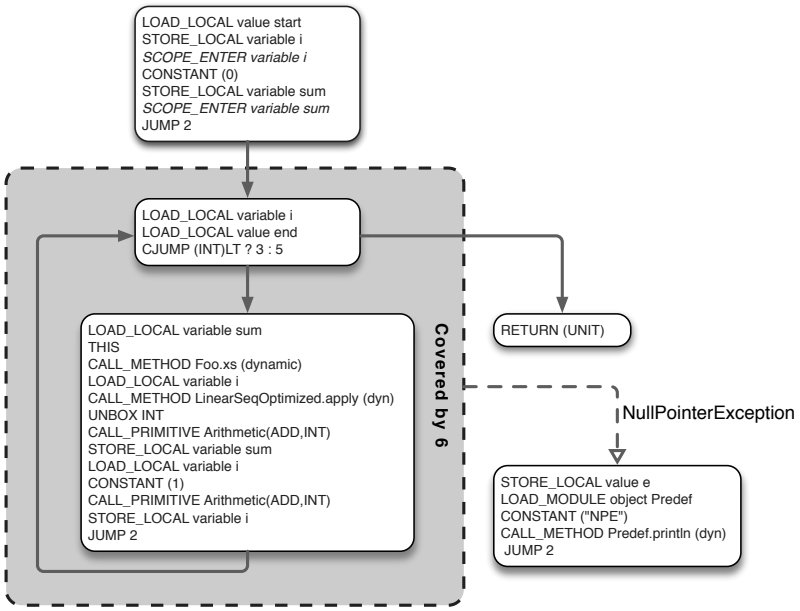


Figure 3.3: The control-flow graph of `sum` with exception handlers

primitive type.

Exception handlers

Exception handlers complicate a bit the design of basic blocks. Suppose we add a handler around the body of `sum`

```
def add(start: Int, end: Int) = {
  var i = start
  var sum = 0
  try {
    while (i < end) {
      sum += xs(i)
      i += 1
    }
  } catch {
    case e: NullPointerException => println("NPE")
  }
}
```

It would be too tedious to break each basic block after every instruction that might throw an exception, since that is the case for most of them. Instead, similar to [15], we consider exceptional successors as special edges, and represent each handler as a pointer to the starting block of a handler, a list of covered

blocks, and the type of the exception that is caught. Note in Figure 3.3 how the exception handler covers the loop body and the test, but not the return instruction.

ICode structure

Program code is represented as a set of ICode classes, and each class in turn is composed of fields and methods. All definitions are referred by their symbolic name, which uses the same entry in the symbol table as all the previous phases.

The Scala compiler assigns a *symbol* to each definition in a program. Distinct definitions may have the same name, but they get different symbols during *name resolution*, right after parsing. In turn, each symbol has a type which may refer to symbols, such as `Int` or `List[Int]`. We chose to use the same representation for symbols in the backend in order to reuse the existing infrastructure, such as overloading resolution and subtype tests.

Methods carry the code associated with a method (represented as a list of basic blocks and the entry point) and provide factory methods for basic blocks and local variables. Related functionality, like merging basic blocks that form a straight sequence is also provided at this level (such code may arise after inlining, for instance).

Local variables, just like every other definition, use a symbolic name. When translated to bytecode, local variables are assigned to *slots*, which are more like registers: they can be used for values of different types, as long as the definition and use are compatible. While the Scala compiler maps local variables to slots on a one-to-one basis, statically typed locals add complexity to bytecode parsing: types have to be inferred for slots, which might not be always possible if the bytecode was generated by other compilers (see § 3.2.2).

Instructions are implemented as case classes, and all instructions carry some additional information, such as the number and types of consumed values off the stack, the line number from which they were generated, and the number of values pushed on the stack.

ICode has the usual structural requirements for stack-based code, such as the stacks at each control-flow merge point have to have the same sizes and be pairwise type-compatible. Each instruction is typed, and the static types of local variables and stack have to match the expected types of the instruction (very much like Java bytecode).

Locals and constats

`LOAD/STORE_LOCAL`, `THIS/STORE_THIS`, `CONSTANT`. Their meaning is straight-forward: the load instructions put the variable on top of the stack, stores pop it off the stack and place it in the given variable. The current instance is handled specially through `THIS`. Constants load their value on top of the stack.

Object manipulation

`LOAD/STORE_FIELD`, `LOAD_MODULE`, `IS_INSTANCE`, `CHECK_CAST`, `MONITOR_ENTER/EXIT`, `BOX`, `UNBOX`, `NEW`. Loading and storing a field operates on the object instance found on top of the stack. The field is specified through its symbol, as an argument to the instruction. Instance checks and casts check that the object on top of the stack has a given class type (specified as a symbol). The last two instructions are used to enter and exit `synchronized` blocks, by entering/exiting the monitor represented by the object on top of the stack.

`BOX/UNBOX` are used to turn primitive types into their wrapper object. They are parameterized with the type of the primitive.

`NEW` creates an instance of a class type. As in Java bytecode, the instance is not initialized. A constructor has to be called explicitly before the object can be used. To make analyses simpler, we keep around a *def-use* chain between a call to `NEW` and the corresponding constructor call.

`LOAD_MODULE` places on top of the stack a Scala top-level object. In the JVM backend, top-level objects are translated to an implementation class and a synthetic static field, initialized on first acces. This instruction is translated to a load of the corresponding field, but having an explicit operation at the ICode level gives freedom of implementation for other backends.

Array manipulation

`CREATE_ARRAY`, `LOAD_ARRAY_ITEM`, `STORE_ARRAY_ITEM`. An n-dimensional array is created by pushing on the stack the size on each dimension. The instruction is parameterized with the element type and the number of dimensions. The instructions for loading and storing elements are parameterized with the element type. The array object and index are taken, as usual, from the stack.

Primitives and methods

`CALL_PRIMITIVE`, `CALL_METHOD`. Primitives are multiplexed, and they are divided in arithmetic, test, logical, conversion, array length and string concatenation. The reason for the last two is that their implementation is platform dependent. In addition to the specific primitive, the instruction takes the type of the operands.

Methods can be called with different conventions: dynamic (normal virtual call), static (no receiver object) and special (static dispatch but with a receiver object – used for constructors and private methods). These conventions are taken directly from Java.

Control flow

`JUMP`, `CJUMP`, `CZJUMP`, `SWITCH`, `RETURN`, `THROW`. All control-flow instructions specify their targets using basic block labels. `CJUMP` compares the two values on top of the stack using the provided test operator and jumps to either of two blocks. `CZJUMP` is similar, but compares the top stack value to zero.

`SWITCH` takes an array of block labels and jumps to the one corresponding to the top of the stack, taken as an index into the array. The last value in the array represents the default destination.

`RETURN` is parameterized with the type of the returned value, and causes the control flow to leave the current method. If the type is `UNIT`, no value is returned to the caller, and the stack may be empty.

`THROW` raises an exception using the value on top of the stack. We use the Java semantics for exceptions: a **`throw`** transfers control to the exception handler that covers the current block, and which handles an exception type that is a supertype of the raised exception. If there is no matching handler, the search continues up the runtime stack until a matching handler is found. If none is found, the thread is terminated [32].

Stack manipulation

`DUP`, `DROP`. `DUP` duplicates the top of the stack. It is parameterized with the type of the stack element it duplicates. `DROP(n)` pops n elements off the stack.

Miscellaneous instructions

`SCOPE_ENTER/EXIT`, `LOAD_EXCEPTION`. Debugging information is kept around at the instruction level, and local variable scope is delimited by `SCOPE_ENTER/EXIT` instructions. They have no runtime effect. `LOAD_EXCEPTION` represents the VM operation of loading the exception object on top of the stack when executing an exception handler. It does not generate any executable code, but it makes the optimization phases more uniform, by having all values on the stack be produced by some instruction.

3.2.2 Bytecode reader

One of the goals of this work is to optimize higher-order functions like `foreach` and `map`. While definitions of these functions are relatively rare in user code, almost all collection classes implement these methods. We need a way to analyze and inline them without requiring the source code. To this purpose we implemented a parser from Java bytecode to ICode.

There are several issues to be considered when lifting low-level bytecode to ICode: *name resolution*, *local variable typing*, *def-use chains* for object initialization and *abstraction recovery*. We begin by looking at the simple cases and then move on to describe each issue in more detail.

Basic blocks

A first pass over the bytecode translates every instruction into its equivalent ICode instruction. For the most part, there is a direct translation for each Java

bytecode. Preliminary ICode is stored into an array, and each jump target address is recorded, while jump instructions are translated into pseudo jumps referring to the pc address. After this pass all basic block boundaries are known, and empty blocks are created for each jump target. A second pass over the code translates pseudo-jumps to proper jumps in terms basic blocks, and distributes each instruction to its own basic block.

Name resolution

ICode uses symbols to refer to classes, methods and fields. These are the same symbols used by the compiler for the front-end, so we need a way to get back to symbols from the (sometimes mangled) names in bytecode. To see why this is important, imagine parsing method `map`, which at some point calls its function parameter `f` on its elements:

```
..
CALL_METHOD scala.Function1.apply()
..
```

The optimizer needs to know that the called method is in fact the same `apply` method that is implemented by the closure that is passed *in the call to* `map`.

Name resolution is problematic because Scala uses a number of techniques to compile features that have no direct correspondence in Java.

splitting Traits are similar to interfaces (essentially, they can be inherited multiple times) but can have concrete methods. Since the JVM does not support this feature, a trait is split into an interface and an *implementation* class. The implementation class carries concrete method implementations as static methods. The bytecode reader has to match the class and interface, and enter the method in the right scope. In a similar way, objects and their companion class are two different classes for the same symbolic name.

mangling Name mangling is used pervasively. Inner classes, top-level objects, private members of traits, to name a few of the cases where this occurs. Most symbol types are computed lazily, so the bytecode reader may read a name that does not exist yet in the symbol table, even though it is part of the program. For instance, inner classes are not added to the package scope[2] unless they are referenced. Suppose the following method call is parsed:

```
CALL_METHOD Outer$Inner.bar()
```

The symbol table contains an entry for class `Outer`, and initially class `Inner` appears as a member of `Outer`. After lambda lifting, inner classes are lifted top-level and their names are mangled. The resulting top-level class is `Outer$Inner`. The lazy nature of typing delays the lifting unless the outer class is referenced somewhere in the *source code* that is currently

[2]This is done in phase *Flatten*, when inner classes are lifted to top-level

compiled. Supposing the bytecode reader encounters that mangled name for the first time, it has to *force* the outer class to find the right symbol.

There are situations in which the name cannot be resolved to a symbol that exists already. Such is the case for anonymous functions, which being local to the method where they are defined, are not part of the interface of any class. In this case the name resolver simply creates a new symbol.

Local variables

ICode requires that each local variable is typed. While Scala-generated bytecode always follows this requirement, `javac` is more liberal, and may use slots for different types. We take a very simple approach: each local acquires the type of its first use. If a subsequent use demands a different, unrelated type, we create a new local variable. This proved to work well in practice, leading to very few "spills". Unlike Miecznikowski et al., we do not try to recover an image as close as possible to the original source code [34], and that allows more freedom in the way we handle local variables.

In order to remain sound, we lose some type information: the type of a local variable can be only a primitive type (`boolean`, `int`, `long`, `double`, `float`) or `Object`. We also assume that the bytecode is *verifiable* by the JVM [32]. This makes any type-conflict of a local variable to correspond to different local variables in the source code.

To see that this is the case, consider a read of slot x. Bytecode is typed, therefore any read or write to x mentions a type (one of the 5 primitive types plus the reference type). Verifiable bytecode implies that every read of a slot x using a type T has an assignment to x using the same type T on *all* control-flow paths leading to it. Therefore, the pair $\langle x, T \rangle$ unambiguously determines all verifiable uses of slot x. If we need to split a slot, it means we encounter an instruction that uses x with an incompatible type T', and verifiability implies that all assignments that flow into (or uses flowing from) the current instruction will be done using T'.

Gagnon et al. describe a more advanced approach to infer types for local variables in bytecode [16], but for our purposes the lack of more precise reference types was not a problem. Type-flow analysis recovers most types during inlining (see § 3.3.1).

Object initialization

As explained in § 3.2.1, `NEW` instructions link to the corresponding constructor call. This def-use chain has to be recovered from bytecode. To see why this is not trivial, consider the following example:

```
new Pair(new A, new B)
```

and the resulting ICode shown in Figure 3.4. Notice that there may be any number of instructions (including other `NEW` or constructor calls) between a `NEW`

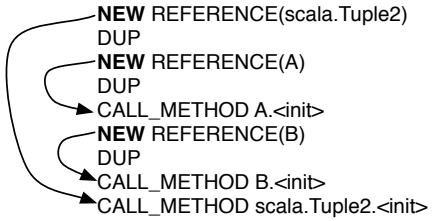


Figure 3.4: ICode for instantiating a pair.

and its corresponding initialization call. In fact, the two instructions don't need to be in the same basic block:

```
new Pair(if (n > 0) new A else new B, new B)
```

This is a global data flow analysis problem, and we rely on a classical reaching definitions analysis to recover the def-use chain for uninitialized objects.

Abstraction recovery

Even though ICode and Java bytecode are very similar, there are a number of operations that require special treatment in the parser:

dup_x1/2 Duplicate and exchange bytecodes are not supported in ICode. The top element of the stack is duplicated and inserted two values down the stack. The operation is simulated using temporary variables (occurs fairly rare in practice).

iinc Increment local variable by a constant. Simulated using arithmetic operations.

modules Loads of the special module instance variable are converted back to one `LOAD_MODULE` instruction.

box/unbox Calls to the runtime library that box/unbox primitive values are converted to explicit `BOX`/`UNBOX` operations.

3.2.3 Data Flow Analysis framework

To facilitate the implementation of various data flow analyses in the compiler we provide a simple framework for iterative forward and backward data flow analysis [38]. The framework is parameterized with a semilattice, that can implement different abstracts values of the analysis domain. The semilattice is specified in terms of the Scala type of the domain values, and an implementation of the least upper bound of two abstract values. For instance, suppose we are interested in *liveness* of local variables [38]. A variable is *live* at a certain program point if there is at least one path in the control-flow starting at

that program point on which the variable is used without being reassigned in between. The lattice is implemented as follows:

```
object livenessLattice extends SemiLattice {
  type Elem = Set[Local]

  val top: Elem = new ListSet[Local]
  val bottom: Elem = new ListSet[Local]
  def lub2(exceptional: Boolean)(a: Elem, b: Elem): Elem = a ++ b
}
```

The abstract values of this analysis are sets of variables. When two control-flow paths merge, the result is the union of the two sets. Intuitively, variables that are live on either of the two possible control-flow paths are live at a split point.

The analysis is fully implemented by providing an implementation for the *transfer function* of a basic block. The transfer function takes a basic block and the current value at the exit of the block (for a backward analysis like liveness) and returns the value at the entry of the block:

```
def blockTransfer(b: BasicBlock, out: lattice.Elem): lattice.Elem =
  gen(b) ++ (out -- kill(b))
```

where the *gen* and *kill* sets describe the effect of a basic block on local variables [38]. *gen* contains variables that get a new value while *kill* are those whose value is invalidated by operations inside *b*.

The framework provides an implementation of a worklist algorithm for computing a fix-point of the transfer functions.

3.3 Optimizations

3.3.1 Inlining

Inlining is the basis on which other optimizations build. Bringing the callee into the caller provides more context for analysis and opportunities for optimizations, so it is very important to be able to inline "interesting" methods, such as `map` or `foreach`, in order to get to the ultimate goal: inline closure methods and remove anonymous function objects altogether.

Inlining depends on being able to statically resolve a method call: to know which implementation is going to be selected at runtime. As described in detail in § 6.2.1, numerous techniques have been proposed in the literature [50, 22, 7, 14, 45]. All of them require the whole-program, and most importantly they are not precise enough: Class Hierarchy Analysis (CHA) considers only methods implemented in subtypes of the static type of the receiver, while Rapid Type Analysis (RTA) prunes them further by keeping only those instances that appear in a call to **new**. Considering the most important use-case, that of closures, the static type of the receiver is always one of the `FunctionN` traits. They have literally hundreds of implementations in a normal program (1200 in the standard library alone). All of them are instantiated.

$$lub(\overline{x \mapsto T_1}, \overline{y \mapsto T_2}) = \overline{x \mapsto lub(T_1, T_2)} \text{ for } x \in \overline{x} \cap \overline{y}$$

$$lub([T_1, rest_1], [T_2, rest_2]) = \begin{cases} [T_1] & \text{if } s_1 \text{ is exceptional} \\ [T_2] & \text{if } s_2 \text{ is exceptional} \\ [lub(T_1, T_2), lub(rest_1, rest_2)] & \text{otherwise} \end{cases}$$

$$lub(\langle v_1, s_1 \rangle, \langle v_2, s_2 \rangle) = \langle lub(v_1, v_2), lub(s_1, s_2) \rangle$$

Figure 3.5: Type Flow Analysis lattice

Our technique is pretty straight forward: propagate types from allocation sites to the call site, through local variables and stack slots. This is similar to Sundaresan's VTA [50], but instead of propagating through the whole program (collapsing methods into one single node in the call graph), we propagate only inside the method. The types are flow sensitive (meaning we may have more precise type for a variable on one of the branches), and the inlining decision can be taken only if the method is `final` (because we do not have a whole-program assumption).

Type flow analysis

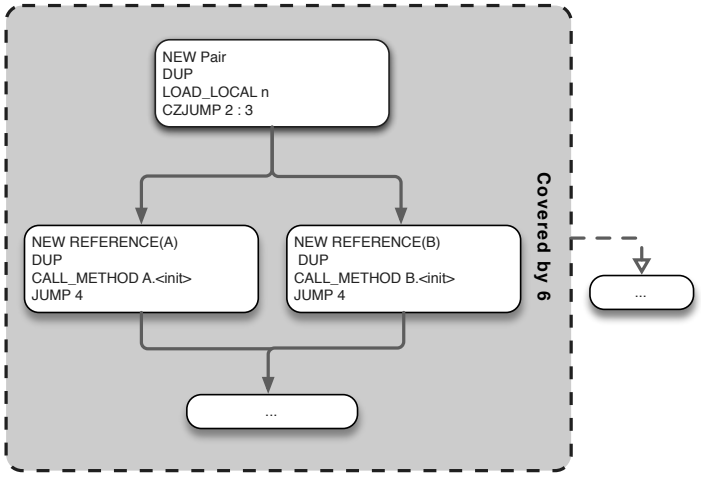
Type Flow Analysis (TFA) infers the type of local variables and stack elements at every point in a method. We use a classical forward data flow analysis, formulated in terms of a type lattice. We begin by defining the type lattice, which is composed of the 9 primitive types and the class hierarchy, having the usual subtyping semantics.

The abstract values of this analysis are pairs of local variable state and stack state:

$$\langle \overline{x \mapsto T}, [T_1, T_2, \ldots T_n] \rangle$$

The first element of the pair maps local variables to types, while the second element is a list of types corresponding to the stack.

We define the ordering relation by a least upper bound operation on the elements of this lattice. We base the operation on the implicit ordering relation defined by the subtyping relation on Scala types, and define a least upper bound for bindings and stacks, as shown in Figure 3.5. Special care has to be taken for control-flow paths involving exceptions. There may be control-flow merge points at the beginning of an exception handler (for instance, when different basic blocks are covered by the same exception handler). The least upper bound in that case has to be the special exception handler stack, containing exactly one element, of the type of the exception being caught. This is a direct consequence of the semantics of Java exception handlers: when an exception handler is invoked, it has exactly one value on the stack (the exception that was thrown). Figure 3.6 shows an exception handler covering an object in-




```
new Pair(if (n > 0) new A else new B, new B)
```

Figure 3.6: Control-flow merge point for exception handlers. The exception handler is a successor of all other basic blocks.

stantiation. The exception handler is a successor of all other basic blocks, but their output stacks may not necessarily have the same number of elements nor types. The least upper bound of any set of stacks flowing into an exception handler is the special exception handler stack. Interestingly, this does not affect local variables, whose state is valid in the exception handler, and whose least upper bound proceeds the normal way.

The analysis is defined in terms of an abstract state and the effect of each ICode instruction. Since all instructions are typed, it is very straightforward to model their effect on the stack and local variable environment. The only instructions that may introduce more precise types are `NEW` and `CHECK_CAST`.

Inlining

TFA provides more precise types at a call site than the static type of the receiver. If the type is precise enough to identify only one possible method implementation, and that method is final, a decision to inline may be taken. The decision depends on *safety* and *heuristic* criteria.

Assuming the icode for the callee method *m* is available, inlining is *safe* inside method *c* when all of the following are true:

visibility Method *m* does not access any private members, or if it does, both *m* and *c* are in the same compilation unit. In the latter case, the member is made public.

hierarchy Method *m* does not call methods through **super**.

stack-compatibility If *m* has exception handlers, the stack in *c* at the call-site has to be empty (except for the arguments to *m*). The reason is that if an exception occurs, the VM would drop all elements off the stack, losing everything that was on the stack before the call to *m*. Without inlining, the exception may be caught inside *m* and the caller stack would be unaffected.

recursive Method m is not recursive. Even though theoretically this is safe, it makes little sense, given that tail recursive calls have been already turned into jumps by an earlier phase.

The heuristics for inlining are taking into account the type of the method, its size, and the combined sizes of the two methods:

+1 The callee is small (one basic block with less than 10 instructions).

+1 The callee is a higher-order method (but not monadic).

+3 The callee is a monadic method (one of `foreach`, `map`, `flatmap`, `withFilter`).

+2 The callee is the `apply` method of a closure.

-1 The caller was straight-line code, and gets turned into a method with more than one basic block.

-1 The callee is a large method.

-2 The callee has been inlined more than twice already in the same method.

This heuristics are used to compute a score for each call-site and callee. If the score is above 0, the optimizer decides to inline. We noticed that too much inlining can hurt performance by increasing the time spent in the JIT compiler, so we increased the cost of the first inline operation by staring with a negative score. After one inline decision has been taken, the initial score is zero for further call-sites in the same method, allowing more inlining to occur, for instance in the body of the recently expanded method.

We also penalize inlining inside closure `apply` methods, even though this means we miss some opportunities to eliminate closure classes. The reasoning is that if a method can be statically resolved inside a closure, it will also be statically resolved when the closure is itself inlined. Cascading inlines of this sort are much more likely to yield good results, since they can reveal nested loops, while a large expanded closure is unlikely to benefit from it unless itself inlined.

The heuristics can be overridden by the user, who can demand a method to be inlined whenever it is safe to do so, by using the `@inline` annotation.

The decision to invoke the bytecode reader when the ICode of a class is not available is driven by another set of heuristics. All methods in the scala runtime may trigger loading, together with methods annotated with `@inline` and **`final`** monadic methods.

Inlining proper is relatively straight-forward, although there are a few places that require special attention:

- The active exception handlers at the call site are extended to cover the inlined blocks. Exception handlers in the callee are merged into the caller.
- *def-use* chains between object instantiation and initialization have to be updated to the new context. Each occurrence of NEW is recorded in a table, mapping the old constructor call with the new instance of the NEW instruction. A pass through the table at the end of inlining updates all chains to the new instructions.
- RETURN instructions executing on a stack with more than one element have to be adapted by dropping the surplus.

Inlining is applied iteratively until reaching a fixpoint, or the maximum number of inline operations. At the end of inlining there is a normalization pass through the method control-flow graph, which removes unnecessary jumps. A sequence of basic blocks that have a single predecessor and a single successor is collapsed into one basic block.

3.3.2 Copy propagation

The goal of this optimization pass is to remove as much indirection as possible. We are interested mostly in finding pairs of local variables and object fields that have the same value, and replacing field loads by local loads. The success of this phase depends on the ability of the inliner to inline interesting methods, like higher-order functions and their closure arguments. For example, consider this for loop:

```
def sample {
  val d = 1
  for (i <- 1 to 10) println(i + d)
}
```

The actual code after for-comprehension expansion and implicit application is (see § 2.5):

```
def sample(): Unit = {
  val d$1: Int = 1;
  Predef.intWrapper(1).to(10).foreach(new Foo$$anonfun$1(Foo.this, d$1))
}
```

and the anonymous function is expanded to the following closure class:

```
final <synthetic> class Foo$$anonfun$1 extends runtime.AbstractFunction1 {
  final def apply(v1: Int): Unit = Predef.println(Int.box(v1.+(this.d$1)))
  final <bridge> def apply(v1: Object): Object = {
    apply(Int.unbox(v1));
    runtime.BoxedUnit.UNIT
  }
  <synthetic> private[this] val d$1: Int = _;
  def this($outer: Foo, d$1: Int): Foo$$anonfun$moo$1 = {
    this.d$1 = d$1;
```

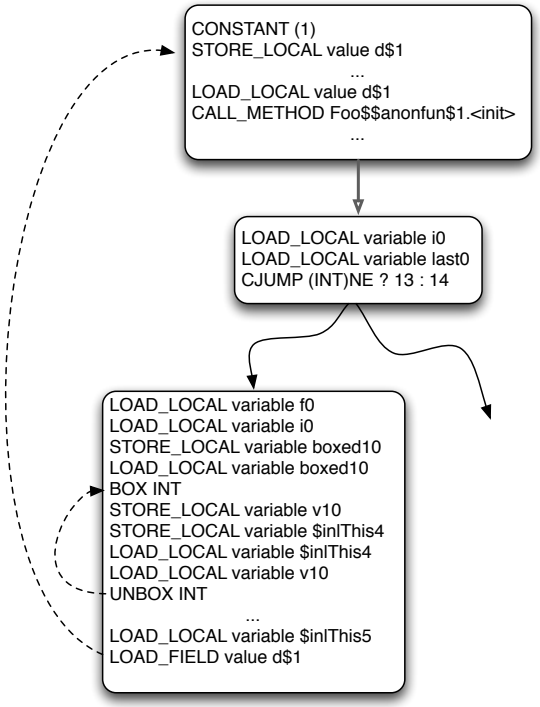


Figure 3.7: Closure elimination: box and unbox pairs and field loads can be replaced by a LOAD_LOCAL

```
      super.this();
    }
  }
```

Notice how the closure has captured local value `d` from the method environment, and how the environment is constructed by passing the outer class and the captured variables to the anonymous function constructor. Suppose that after inlining, most of these methods have been inlined, including `foreach` and `Function1.apply` (remember that foreach is defined in terms of the `Function1` interface). The closure phase should be able to infer that `v1 + this.d$1` can be rewritten to `v1 + d$1`.

The analysis infers copy relations between local variables and other locals, fields and constants. Figure 3.7 shows part of the ICode of method `sample` after inlining. The dotted arrows show interesting relations between costly operations. The field access in the closure can be traced back to the local variable assignment, through the constructor call of the anonymous function. The unbox operations can be paired with the earlier box, and notice that variable `v10` contains a boxed representation of variable `boxed1`, and the unbox operation is per-

L	::=	Local(v)	local variable
	|	Field(r, sym)	field in record r with name sym
	|	This	the special local variable this
v	::=	Deref(L)	value at location L
	|	Boxed(L)	boxed value of L
	|	Const(k)	constant
	|	Record(cls, b)	record with type cls and bindings b
	|	$\top$	top value

$$State ::= \langle \{\overline{L \mapsto v}\}, [\overline{v}] \rangle$$

Figure 3.8: Abstract values for copy propagation

$$lub([v_1, rest_1], [v_2, rest_2]) = \begin{cases} [v_1, lub(rest_1, rest_2)] & \text{if } v_1 = v_2 \\ [\top, lub(rest_1, rest_2)] & \text{otherwise} \end{cases}$$

$$lub(\overline{L_1 \mapsto v_2}, \overline{L_2 \mapsto v_2}) = L \mapsto v_1 \text{ for } L \in \overline{L_1} \cap \overline{L_2}, \text{ if } v_1 = v_2$$

Figure 3.9: Least upper bounds for copy-propagation

formed on `v10`. The unbox operation can be replaced by a `LOAD_LOCAL boxed10`.

Copy Propagation transformation

We extend the classical copy propagation algorithm [36] to handle constants, boxed values and simple records. We compute at each program point the available bindings between abstract locations and abstract values. We model the stack as a list of abstract values, and locals as a map from locations to values. Figure 3.8 defines locations and values of our analysis. Values can be stored in local variables or in record fields. A value is either a dereference of a location, a boxed representation of a value found at another location, a constant (literal) or a record. Records are very simple models of heap objects which carry a type (the class from which they are instantiated) and a binding of their fields. We do not deal with aliasing, instead we get around it by merging together all instances of a class, so assigning to a field destroys all bindings in all records that contain that field. Local variables cannot be aliased on the JVM, so this is enough to ensure correctness.

The analysis proper is a forward data-flow analysis, and the least upper

bound is computed as shown in Figure 3.9. The least upper bound of two stacks is computed pairwise, and use $\top$ at stack positions where the two values are not the same. Variable bindings are computed similarly, keeping only mappings that appear in both states, and that map the same location to the same value.

We model the effect of each instruction on the abstract state:

- `LOAD_LOCAL L`, `THIS` place on top of the stack $Deref(L)$
- `CONSTANT k` places $Const(k)$
- `BOX` places $Boxed(L)$ if the top of the stack is $Deref(L)$.
- `NEW` places a new empty $Record(cls, \epsilon)$
- `LOAD_FIELD f` places the corresponding binding, when the top of the stack is $Record(cls, b)$
- `STORE_LOCAL L`, `STORE_THIS` add a binding between L and the value on top of the stack (if it is not $\top$). It removes all bindings referring to L (or **`this`**) from both the stack and bindings, including inside records.
- `UNBOX` places v, if the top of the stack is $Boxed(v)$
- `STORE_FIELD f` adds a binding to the corresponding field, if the top of the stack is $Record(cls, b)$. Removes all bindings and values referring to f from both the stack and other bindings, including inside records.
- `CALL_METHOD m` If m is a known constructor (see below), places a $Record(cls, b)$ with determined bindings. Unless m is pure (see below), removes all bindings of mutable fields in all records.

Whenever the side condition is not true (and for all other instructions), the operation is simulated by consuming and replacing the required number of stack locations, producing the top value $\top$.

Operations are straight-forward, except for method calls. If the call is a constructor, we may be able to infer the shape of the object after the call. In the case of a case class, constructors are predetermined to fill its fields with their arguments, and there is a one-to-one mapping between arguments and fields. This is especially useful for anonymous function classes, which use the same technique. We approximate the effect of a `CALL_METHOD` to conservatively modify all fields, and therefore lose all information gained so far. A side-effect analysis would help to greatly increase the precision of this analysis. Currently we consider getter methods (as generated by the scala compiler) to be side-effect free.

Optimization pass

The previous analysis provides for each program point the values on the stack, and bindings between local variables. It is now easy to perform one pass through the code, replacing each instruction that loads a location with the value found by copy propagation (another location, a constant or an unboxed value). We follow the longest chain through the bindings, hoping that intermediate nodes in the chain are local variables that become dead after this phase. This is a common case in practice, since inlining introduces many local variables to hold the parameters of the call, which later can be proven to be aliases of existing locals, and therefore eliminated.

There are a few cases to consider:

- `LOAD_LOCAL L` if we have a binding for L, is replaced by a load of the corresponding value (another local, **`this`**, or a constant – effectively achieving constant propagation)
- `LOAD_FIELD F` if the top of the stack can be resolved to a record, and that field is bound by the record, is replaced by a `DROP` and a load of the target value (unless `volatile`).
- `UNBOX` if the top of the stack is $Boxed(L)$, is replaced by a `DROP` and a load of the corresponding value

Volatile fields on the JVM have special multithreaded semantics. A read from a field is a synchronization point, and even if the value is not used, its effect may be visible, therefore optimizations cannot remove them.

In order to maximize the chance of having available an unboxed version of a value, the code generator introduces special local variables for each `BOX` operation. These synthetic locals are dead otherwise and if the copy propagation phase does not use them (to eliminate an `UNBOX`), they are collected in the next optimization pass, dead-code elimination.

3.3.3 Dead code elimination

Inlining and copy propagation leave behind code that is no longer needed. Many local variables introduced by the inliner to hold parameters to the inlined method are later dead because copy propagation can prove they are aliases of other local variables. Objects allocated to hold the state of an anonymous function may become dead once the `apply` method is inlined, and all references to its own field have been rewritten to use the existing method environment. Local classes (classes defined inside a method) that are never instantiated are no longer needed and can be completely removed from generated code.

We use a mark & sweep algorithm for finding the instructions that are needed for the correct execution of a method. In a first pass, we mark instructions that are useful regardless of their result. This consists of:

- `RETURN` – the value returned by a method is by definition *live*.

- STORE_ARRAY_ITEM, STORE_FIELD – arrays and fields may be aliased, and the effects of these instructions may be visible outside the current method.
- MONITOR_ENTER/EXIT – synchronization cannot be removed.
- control-flow instructions – we do not remove jumps and throws. Technically it would be possible, but we chose to keep this phase simple.
- CALL_METHOD m – if method *m* is side-effecting. A pureness analysis would help this analysis as well. We consider getters and closure constructors to be side-effect free, together with the Rich* objects in the Scala runtime.
- DROP – if reaching definitions for the dropped value contain a side-effecting method, it is marked useful. Otherwise, the usefulness of a DROP cannot be determined by itself. Instead, it is useful only if one of the instructions that produce the value to be dropped is useful (see discussion below).

Once the initial instructions are marked, we precess each one in turn using a worklist. Each instruction in the worklist is marked as useful and its reaching definitions are added to the worklist, iterating until the list becomes empty. Some care has to be taken for DROP and NEW:

- If an instruction is useful, and we recorded that its value flows into a DROP, the DROP is marked as useful. This corresponds to the case when an instruction produces a useful value on one branch, and that is dropped on another. Notice that this is follows a *def-use* chain, as opposed to the other cases which follow *use-def*.
- If a NEW instruction is marked as useful, the corresponding constructor call has to be marked as well.

Once marking is done, we perform a *sweep* over the code, keeping only instructions marked in the previous step. This requires some stack bookkeeping, basically adding the necessary amount of DROPs for each of the values produced by a dead instruction.

3.3.4 Peephole optimizations

A last pass through the code eliminates simple patterns by more efficient (or simply more pleasant to look at) operations. The peephole optimizer is limited to replacing a pair of consecutive instructions by another sequence, possibly empty. Some of the combinations we tackle are:

- A load (local variable, field, or constant) followed by a drop are both eliminated (unless the field is volatile)
- A load followed by a store to the same local are both removed.

- A store followed by a local of the same local are replaced by `DUP; STORE_LOCAL`. We implemented a variant of this optimization by using liveness information. If the variable is dead after the assignment, both instructions are eliminated. This situation appears fairly frequently in inlined code.
- A box followed by an unbox are both removed.

We did not notice significant performance improvements due to the peephole optimizer, but it helps understanding generated code and results in cleaner bytecode.

Chapter 4

Opportunistic Specialization

> As much as I love boxing, I hate it.
> And as much as I hate it, I love it.
>
> Budd Schulberg

4.1 Introduction

Parametric polymorphism has become a standard feature in statically-typed programming languages. The ability to write code that operates uniformly on values of different types increases expressiveness and leads to shorter, clearer programs. Polymorphic[1] code is able to manipulate values of unknown types, and does so in a generic way. Unfortunately, run-time efficiency is achieved through optimal use of primitive operations, and current day processors operate on values such as double precision floating point values, in other words require precise type information to select the right instruction.

There are two ways to compile polymorphic code [35]: use a single version of the compiled code, operating on a uniform representation of values (*homogeneous* approach); or generate a specialized version of the generic code, as a function of the concrete instantiated type (*heterogeneous* approach). One way of implementing an homogeneous approach is through *type erasure*, a translation technique which replaces type variables with their upper bound (or `Any` in Scala, `Object` in Java). When the code enters/exits generic code, the compiler inserts casts to maintain type correctness. As explained in detail in § 2.5.1, the main shortcoming of erasure is performance for primitive values: primitives need to be *boxed* into heap-allocated objects to fit the uniform representation. Each access to a primitive needs an extra indirection.

[1] In this section we use the term polymorphism to mean parametric polymorphism. When we refer to subtyping polymorphism it will become clear from the context.

In the heterogeneous approach polymorphic code generates a number of variants which are specialized to work on the given concrete type instantiation. In this case, values use their natural representation, and code runs at full speed. In addition, run-time type information is usually available. The downside is a risk of code bloat (and long compilation times), as there may be any number of different instantiations of a generic definition. Furthermore, this implies a lack of true *separate compilation*, as the compiler needs to generate new code at each instantiation. C++ templates are the most famous example of generics that are statically compiled through specialization [10]. They notoriously exhibit long compilation time and may lead to code explosion [17].

Some of these adverse effects can be alleviated by link-time specialization, the approach taken by the .NET Common Language Runtime [27]. By raising the abstraction level of the compiled code, polymorphic code is instantiated by the VM, and compilers generate code in terms of type variables. Since the VM has the whole program at hand, it can perform instantiation and specialization at runtime, and it may decide to share code between instantiations when it is safe to do so.

There seems to be a tension between tight, efficient code and predictable code size. Moreover, the ideal compilation model depends on the program at hand: a scientific library needs every bit of performance and programmers would gladly sacrifice compilation speed and code size, while a generic UI container library is very likely to favor fast compilation times and small code size over performance. So far language designers chose one compilation model that is set in stone per language, and programmers had no choice but to switch to another language when the model does not suit their application. We propose a hybrid model in which programmers annotate generic definitions that should be specialized.

In this chapter we present a new way of compiling generic code that can be both fast and compact, and that does not require special support from the runtime system. Based on the observation that significant performance is lost only for a small number of type instantiations, we design a solution that is a combination of the two alternatives described above: most generic code uses a common representation, but when performance is critical the programmer may require certain classes or methods to be specialized. Our solution supports separate compilation and allows mixing specialized and generic code.

The rest of this section is structured as follows: Section 4.2 introduces specialization in an intuitive way. Section 4.3 presents a formal description of the specialization translation and proves type preservation, while the last section discusses the implementation in the context of the full Scala language.

4.2 Generic versus specialized code

To illustrate our approach we use a generic matrix definition and a function that implements matrix multiplication on integers. Figure 4.1 shows a simple matrix class defined in Scala. The annotation on type parameter `A` instructs the

```
class Matrix[@specialized A: ClassManifest](val rows: Int, val cols: Int) {
  private val arr: Array[Array[A]] = new Array[Array[A]](rows, cols)

  def checkBounds(i: Int, j: Int) {
    if (i < 0 || i >= rows || j < 0 || j >= cols)
      throw new NoSuchElementException("Indexes out of bounds: " + (i, j))
  }

  def apply(i: Int, j: Int): A = {
    checkBounds(i, j)
    arr(i)(j)
  }

  def update(i: Int, j: Int, e: A) {
    checkBounds(i, j)
    arr(i)(j) = e
  }
}
```

Figure 4.1: A generic Matrix class

compiler to specialize code that depends on A. By default, the matrix will be specialized for all primitive types.

The matrix class provides only two operations, retrieving and updating an element at position i, j. The type parameter is bounded by a `ClassManifest`, which is not essential for this example, but we chose to include it in order to show compilable code[2].

4.2.1 The gist

A generic class definition with specialized type parameters generates a set of specialized classes, deriving each one from the original class using a specific combination of specialized type parameters. When generic code is used in a context where more type information is available, and there exists a specialized version, the compiler rewrites method calls and class instantiations to the specialized version.

For specialized and non-specialized code to work together, a specialized class has to be a subtype of the generic one. This fact allows the compiler to safely replace an instantiation of a generic class with a specialized class when one exists, and the code around it to work with a specialized instance even when the code is not specialized itself. In a context where type information is more specific, for instance the matrix is known to be `Matrix[Int]`, the compiler inserts calls to its specialized variants. We call this approach *opportunistic*

[2] A class manifest is a runtime representation of type A, and is needed for instantiating arrays of an unknown type.

because correctness does not depend on the ability of the compiler to replace generic calls with specialized variants, instead it is a best-effort in the quest for better performance.

We achieve separate compilation by specializing at the definition-site on a finite number of types. When requested, specialization is performed for all primitive types but the user may indicate that only a subset of those should be considered. Eager specialization may generate classes for instantiations that are never needed, but this will not impact performance too much: the Java Virtual Machine relies on lazy class loading, so these additional definitions won't be loaded. The increase in code size we deem tolerable, and leave it to the library designer to find a good balance between performance and code size. The key difference from traditional specialization is that code size is bounded and completely determined by the generic definition, regardless of the number of instantiations.

Figure 4.2 shows how the `Matrix` class is specialized. For clarity, we consider only type `Int` and only one method. In addition to the original methods, the generic class has additional specialized methods whose signature was changed to use concrete, primitive types. These provide a "fast path" for code using matrices at a specialized type, like `Matrix[Int]`, who can call them without a cast or instance check. These methods have a default implementation that simply delegates to the original method, boxing and unboxing values as needed. This way we ensure that specialized code can use generic instances.

Each combination of specializable type parameters generates a specialized subclass. Each inherited member that needs specialization (a more precise definition follows in the next section) is overridden in the subclass, and the generic/specialized delegation is rewired in the opposite sense: the generic method delegates to the specialized variant and takes care of boxing/unboxing as needed. This rewiring ensures that calls through the generic methods end up executing the same payload. If the overridden method is concrete, a specialized implementation is added in the subclass, rewriting the code to take advantage of the new type information. In Figure 4.2 the update method is rewritten to directly access the array of integers, instead of going through a runtime method that uses instance tests (see §2.4.1).

So far we have looked at how definitions are specialized. To close the loop we need to rewrite existing code to use specialized code whenever possible. There are two transformations that together lead to improved performance:

- method calls, or more generally, any member selection, should be rewritten to use the most specific member available. This can be done whenever the type on which a selection is made is a generic type that is instantiated with primitive types.
- **`new`** should create the most specific type available. As before, this can be done whenever the type passed to **`new`** is a generic type that is instantiated with primitive types.

Figure 4.3 shows a straight-forward function for multiplying two matrices.

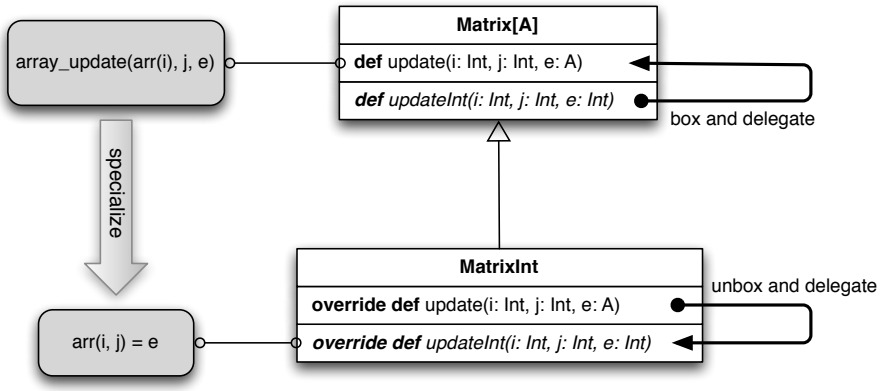


Figure 4.2: Matrix specialization

The code is desugared to show explicit calls to `apply` and `update`. Inside method `mult` the type of `m`, `n` and `p` is known to be `Matrix[Int]`, so a specializing compiler can safely rewrite method calls to more specific variants. Similarly, the instantiation is rewritten to `MatrixInt`.

The whole approach is based on the ability to rewire generic methods towards specialized variants. The question arises what happens for other class members, like fields or types. Type members are the easy case since they have no representation at runtime, so they need not be specialized. On the other hand, fields do, and their representation is important: we do not wish to store primitive types into generic fields, relying on boxing. Moreover, fields are not overridable from the point of view of the virtual machine. The solution is to access all fields through getters and setters, and rely on the same mechanism to rewire accessors. Fortunately, Scala already does so (another example of the uniformity principle, which dictates that all members should be overridable).

So far we have presented an intuitive translation that glossed over details such as method type parameters, bounded type parameters, inner classes, inheritance or object initialization. This omission will be addressed in the following two sections, which introduce a general translation procedure and describe solutions to the problems mentioned above.

4.3 A formal description of specialization

In the following sections we use the term *specializable class* or *specializable method* instead of the exact but cumbersome "a class/method that has specialized type parameters". *Specialized class/method* refers to the result of specialization applied to a class or method.

```
def mult(m: Matrix[Int],
         n: Matrix[Int]) {
  val p = new Matrix[Int](m.rs,n.cs)

  for (i <- 0 until m.rs;
       j <- 0 until n.cs) {
    var sum = 0
    for (k <- 0 until n.rs)
      sum+=m.apply(i,k) * n.apply(k,j)
    p.update(i, j, sum)
  }
}
```

```
def mult(m: Matrix[Int],
         n: Matrix[Int]) {
  val p = new Matrix$Int(m.rs, n.cs)

  for (i <- 0 until m.rs;
       j <- 0 until n.cs) {
    var sum = 0
    for (k <- 0 until n.rs)
      sum += m.apply$Int(i, k)
           * n.apply$Int(k, j)
    p.update$Int(i, j, sum)
  }
}
```

Figure 4.3: Matrix multiplication example.

4.3.1 BabyScala

We formalize our translation using a calculus in the spirit of Featherweight Generic Java [25] and Baby IL with Generics (BILG) [60]. BILG was designed in an effort to formalize the compilation of generics on the .NET platform, and therefore is closer in purpose to our goal. Figure 4.4 shows Baby Scala, a calculus that captures the main aspects of the object system of Scala. Except for the syntax, which we adapted to be more like Scala, the language is almost identical to BILG.

Types can be primitive types, instantiated types or type variables. For simplicity, we consider only one primitive type, integer. Classes can be instantiated at any type, including primitive types, and similar to BILG, we do not have bounds on type parameters. Constructors are completely determined by the class definition, so we choose to omit them. We add one syntactic construction to BILG, type ascription. The specialization translation is much easier to define if we have the explicit type of the receiver in method calls, and we require that the input to specialization is a well-typed program. The type-checker adds all ascriptions for method calls. Internal types include method types, which cannot appear in the source program, but which are needed in the translation.

Figure 4.5 describes well-formed types and the rules for subtyping. We use the standard rules, with transitive and reflexive subtyping. Notice that primitive types are in a subtype relationship only with themselves. We departed from the unified hierarchy of types in Scala in order to be closer to the generated code, that requires boxing and unboxing for this relationship to hold. Typing is defined with reference to an environment Γ, and a class table $\mathcal{D}$. The environment binds variables to their type, and also contains all type variables currently in scope: $\Gamma = \overline{X}, \overline{x : T}$ with all free type variables in T appearing in $\overline{X}$. The class table contains all classes defined in a BabyScala program.

T	::=	$P \mid X \mid I$	types
P	::=	Int	primitive types
I	::=	$C[\overline{T}]$	instantiated type
L	::=	class $C[\overline{X}]$ extends I $\{\overline{f : T}; \overline{M}\}$	class definition
M	::=	def $m[\overline{X}]\ (\overline{x : T}) : T = e$	method definition
e	::=	n	integer constant
	$\mid$	x	variable
	$\mid$	$e.f$	field selection
	$\mid$	$e.m[\overline{U}](\overline{e})$	method call
	$\mid$	new $I(\overline{e})$	instance creation
	$\mid$	e.as$[T]$	type cast
	$\mid$	$e : T$	type ascription
v	::=	n $\mid$ I$(\overline{f} \mapsto \overline{v})$	values
IT	::=	T	internal types
	$\mid$	$\forall \overline{Y}.\overline{T} \to T$	method types

Figure 4.4: BabyScala syntax

Figure 4.6 defines the typing rules for BabyScala. Typing is straightforward, and to keep the rules clean we use two helper functions: $\mathit{fields}(I)$ returns all fields of an instantiated type, including the inherited ones; $\mathit{mtype}(m, C[\overline{T}])$ returns the type of a method *seen* from the given instantiated type. A type seen from an instantiated type $C[\overline{T}]$ is a type whose occurrences of C's type parameters have been replaced with the corresponding types. We use the standard notation for arrow types, $\forall \overline{X}.\overline{T} \to T$, meaning the type of a method taking type parameters $\overline{X}$, value parameters of types $\overline{T}$ and returning a value of type T. For a class to be well-formed we require that its superclass and its methods are well-formed. Fields cannot be overridden (unlike Scala). Methods cannot be overloaded and overriding is valid only when the types of the two methods are exactly the same. The predicate *override* checks that when there is an inherited method with the same name as the defining method, they have the same type (modulo method type parameters).

The evaluation rules are shown in Figure 4.8. Values are integers and object instances, each field in an object instance being in turn a value. We use a big-step semantics, and the only difference compared to BILG is that we have failing casts. BILG uses a type-test expression with a default value if the tests fail, while in BabyScala a failing cast causes a stuck term. We chose explicit casting in order to stay closer to Scala, and because casts to primitive types/-type variables is the way Scala represents unboxing/boxing operations at the source level. A cast from a primitive type to a type variable represents a box operation (and conversely for a cast to a primitive type). Other casts may issue

$$\boxed{\Gamma \vdash T\ ok}$$

$$\Gamma \vdash \text{Int}\ ok. \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ ok.} \qquad \frac{\Gamma \vdash \overline{T}\ ok. \qquad C[\overline{X}] \in \mathcal{D}}{\Gamma \vdash C[\overline{T}] ok.}$$

$$\boxed{\Gamma \vdash S <: T}$$

$$\Gamma \vdash S <: S \qquad \frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \qquad \frac{C[\overline{X}] \text{ extends } I\ \{...\} \in \mathcal{D}}{\Gamma \vdash C[\overline{T}] <: [\overline{T}/\overline{X}]I}$$

Figure 4.5: Well-formed types and subtyping.

warnings when erasure may prevent a correct answer.

We split the formal transformation in three, which makes it easier to explain but also follows our implementation. The main translation procedure is called *spec*, and generates specialized variants with regard to class type parameters. Method type parameters are specialized using a specific translation called *norm*, which normalizes method definitions by generating variants for their type parameters. The output of *norm* is then used by *spec*. Besides the two translations for definitions, there is a third translation called TS, which rewrites terms to instantiate specialized classes and use specialized methods whenever it is safe to do so.

We only consider well-typed programs, and all method calls are ascribed with the receiver type. This sort of type elaboration can be done during type-checking, which produces both a type and a new term.

4.3.2 Method expansion

We begin by defining a normalization transformation which takes care of generic methods, generating a number of specialized variants for each specialization of its type parameters. This first step takes into consideration only method type parameters, specialization of the enclosing class being left for the *spec* translation, presented in the following section.

Normalization adds a number of method definitions derived from the original method, called *variants* (Figure 4.9). Each specialization of its type parameters generates one variant. The original method definition is unchanged, and the additional methods differ only in that all type annotations, both in its type and in its implementation, have been mapped through their respective specialization.

Specializations, denoted by s are functions from type variables to primitive types, and $\mathcal{P}$ generates all specializations of the given type parameters. We use $s_1 \oplus s_2$ to denote extending a specialization with another specialization, when the domains of s_1 and s_2 are disjoint. A specialization applied to a type, denoted by $|T|_s$, recursively replaces occurrences of type variables with the

$$\frac{\Gamma \vdash e : T \qquad \Gamma \vdash T <: T'}{\Gamma \vdash e : T'} \quad \text{(T-SUBSUMPTION)}$$

$$\Gamma \vdash n : \text{Int} \quad \text{(T-INT)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e : I \qquad fields(I) = \overline{T\ f} \qquad f_i \in \overline{f}}{\Gamma \vdash e.f_i : T_i} \quad \text{(T-FIELD)}$$

$$\frac{\Gamma \vdash e : I \qquad mtype(\mathrm{m}, \mathrm{I}) = \forall \overline{Y}.\overline{T} \to T \qquad \Gamma \vdash \overline{e} : [\overline{U}/\overline{Y}]\overline{T}}{\Gamma \vdash e.m[\overline{U}](\overline{e}) : [\overline{U}/\overline{Y}]T} \quad \text{(T-INVOKE)}$$

$$\frac{\Gamma \vdash I \text{ ok} \qquad fields(I) = \overline{T\ f} \qquad \Gamma \vdash \overline{e} : \overline{T}}{\Gamma \vdash \text{new } I(\overline{e}) : I} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash e : T' \qquad \Gamma \vdash T\ ok.}{\Gamma \vdash e.\text{as}[T] : T} \quad \text{(T-TEST)} \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash (e : T) : T} \quad \text{(T-ASCRIPTION)}$$

Class and method typing $\qquad \vdash md\ ok\ in\ C[\overline{X}]$ $\qquad \vdash cd\ ok\ in\ \mathcal{D}$

$$\frac{mtype(m, C[\overline{T}]) = \forall \overline{X}.\overline{S} \to S \implies [\overline{Y}/\overline{X}]\overline{S} = \overline{U}\ and\ [\overline{Y}/\overline{X}]S = U}{override(m, C[\overline{T}], \forall \overline{Y}.\overline{U} \to U)}$$

$$\frac{C[\overline{X}] \text{ extends } I\{...m..\} \in \mathcal{D} \qquad override(m, I, \forall \overline{Y}.\overline{T} \to T) \qquad \overline{X}, \overline{Y}, \overline{x : T}, \text{this} : C[\overline{X}] \vdash e : T}{\vdash \text{def } m[\overline{Y}](\overline{x : T}) : T = e\ ok} \quad \text{(T-METHOD)}$$

$$\frac{\overline{X} \vdash I, \overline{T}\ ok. \qquad fields(I) = \overline{g : U} \qquad \overline{f}, \overline{g}\ distinct \qquad \overline{md}\ ok\ in\ C[\overline{X}]}{\vdash \text{class } C[\overline{X}] \text{ extends } I\ \{\overline{f : T}; \overline{md}\}\ ok} \quad \text{(T-CLASS)}$$

Figure 4.6: Typing Rules

$$\frac{C[\overline{X}] \text{ extends } I\ \{\overline{f_1 : U_1}; \overline{md}\} \qquad fields([\overline{T}/\overline{X}]I) = \overline{f_2 : U_2}}{fields(C[\overline{T}]) = \overline{f_1} : [\overline{T}/\overline{X}]\overline{U_1}, \overline{f_2 : U_2}}$$

$$\frac{\begin{array}{c} C[\overline{X}] \text{ extends } I\ \{\overline{f : V}; ...\text{def } m[\overline{Y}](\overline{x : U}) : U = e...\} \\ \overline{U_1} = [\overline{T}/\overline{X}]\overline{U} \qquad U_1 = [\overline{T}/\overline{X}]U \end{array}}{\begin{array}{c} mype(m, C[\overline{T}]) = \forall \overline{Y}.\overline{U_1} \to U_1 \\ mbody(m, C[\overline{T}]) = \langle \overline{x}, [\overline{T}/\overline{X}]e \rangle \end{array}}$$

$$\frac{C[\overline{X}] \text{ extends } I\ \{\overline{f : V}; \overline{md}\} \qquad m \notin \overline{md}}{\begin{array}{c} mtype(m, C[\overline{T}]) = [\overline{T}/\overline{X}]mtype(m, I) \\ mbody(m, C[\overline{T}]) = [\overline{T}/\overline{X}]mbody(m, I) \end{array}}$$

Figure 4.7: Helper functions for BabyScala type checking

$$\frac{(x = v) \in E}{E \vdash x \Downarrow v} \text{ (E-VAR)} \qquad \frac{E \vdash e \Downarrow I(\overline{f} \mapsto \overline{v})}{E \vdash e.f_i \Downarrow v_i} \text{ (E-FIELD)}$$

$$\frac{\begin{array}{c} E \vdash e \Downarrow I'(\overline{f} \mapsto \overline{v}) \qquad E \vdash \overline{e} \Downarrow \overline{w} \\ \text{mbody}(m, I') = \langle \overline{x}, e' \rangle \qquad \text{this} = I'(\overline{f} \mapsto \overline{v}), \overline{x = w}, E \vdash e' \Downarrow w' \end{array}}{E \vdash e.m[\overline{U}](\overline{e}) \Downarrow w'} \text{ (E-CALL)}$$

$$\frac{E \vdash \overline{e} \Downarrow \overline{v} \qquad \text{fields}(I) = \overline{f} : \overline{T}}{E \vdash \text{new } I(\overline{e}) \Downarrow I(\overline{f} \mapsto \overline{v})} \text{ (E-NEW)}$$

$$\frac{E \vdash e \Downarrow n}{E \vdash e.\text{as}[Int] \Downarrow n} \text{ (E-ASINT)}$$

$$\frac{E \vdash e \Downarrow I'(\overline{f} \mapsto \overline{v}) \qquad I' <: T}{E \vdash e.\text{as}[T] \Downarrow I'(\overline{f} \mapsto \overline{v})} \text{ (E-AS)}$$

Figure 4.8: Evaluation rules for BabyScala

$$norm\,[\![\text{def } m[\overline{Y}](\overline{x : T}) = e]\!] \quad ::= \quad \{\text{def } m[\overline{Y}](\overline{x : T}) = e\} \cup \{\text{def } m_s(\overline{x} : |T|_s) = [\![e]\!]_s \mid s \in \mathcal{P}(\overline{Y})\}$$

Figure 4.9: Normalization of BabyScala methods

type they are mapped to. The same transformation applied to a term is denoted by $[\![e]\!]_s$. Their precise definitions are listed in Figure 4.10.

For example:

$$\begin{aligned} norm\,[\![\text{def } m[X,Y](x : X, y : List[Y]) = (y : List[Y]).head()]\!] \quad = \\ \{\text{def } m[X,Y](x : X, y : List[Y]) = (y : List[Y]).head()\} \\ \cup\, \{\text{def } m_s(x : Int, y : List[Int]) = (y : List[Int]).head()\} \end{aligned}$$

Given that Int is our only primitive type, there is only one specialization that can be formed. For clarity, we do not give an explicit name mangling scheme, and name all specialized variants m_s, but in the following section we describe a necessary criteria for choosing one.

4.3.3 Class specialization

The next translation scheme for definitions takes into account only class type parameters. For each class, *spec* generates specialized subclasses for each specialization s. The original class definition is augmented with *specialized variants* of each method[3] and *specialized overrides*. Each variant delegates to the original method. Figure 4.11 shows the formal definition of *spec*. The interesting bit to note is how fwd_s implements delegation: because the parameter types may be different from the original types, it has to add casts. For example:

$$\begin{aligned} fwd_s[\![\text{def } m(x : T) : T = x]\!] ::= \\ \text{def } m_s(x : Int) : Int = (\text{this} : C[\overline{X}]).m(x.\text{as}[T]).\text{as}[Int] \end{aligned}$$

The call to m expects an argument of type T, while x has type Int. These casts are guaranteed to succeed, as specialized variants are only called when the static type of the receiver is compatible with s.

A specialized override for method m is necessary when it overrides m' and m' has specialized variants. In order to keep the implementations 'in sync', any overriding has to occur both for the generic and specialized method. We use a helper macro *overriddenFrom*, which given a method m and a class type,

[3]The implementation is optimized to not generate variants if the method type does not use any class type parameters. In that case, all variants would be identical to the original method.

$$\mathcal{P}(X) = \{s = \{X \mapsto T\} \,|\, T \in P\}$$
$$\mathcal{P}(X, \overline{X}) = \left\{s_1 \oplus s_2 \mid s_1 \in \mathcal{P}(X), s_2 \in \mathcal{P}(\overline{X})\right\}$$

$$|T|_s = \begin{cases} s(X) & \text{when } T = X, \text{ s is defined at } X \\ X & \text{when } T = X, \text{ s is not defined at } X \\ P & \text{when } T = P \\ C[s(\overline{X})] & \text{when } T = C[\overline{X}] \\ \forall \overline{Y}.(|\overline{T}|_s \to |T|_s) & \text{when } T = \forall \overline{Y}.(\overline{T} \to T) \text{ and } s \text{ not defined at } \overline{Y} \end{cases}$$
$$|x : T, \Gamma|_s = x : |T|_s, |\Gamma|_s$$
$$|X, \Gamma|_s = |\Gamma|_s \qquad \text{if s is defined at } X$$
$$|X, \Gamma|_s = X, |\Gamma|_s \qquad \text{if s is not defined at } X$$

$$[\![n]\!]_s = n$$
$$[\![x]\!]_s = x$$
$$[\![e.f]\!]_s = [\![e]\!]_s .f$$
$$[\![e.m[\overline{U}](\overline{e})]\!]_s = [\![e]\!]_s .m[|\overline{U}|_s]([\![\overline{e}]\!]_s)$$
$$[\![\text{new } I(\overline{e})]\!]_s = \text{new } |I|_s([\![\overline{e}]\!]_s)$$
$$[\![e.\text{as}[T]]\!]_s = [\![e]\!]_s .\text{as}[|T|_s]$$
$$[\![e : T]\!]_s = [\![e]\!]_s : |T|_s$$

Figure 4.10: Additional definitions used for the specialization translation

$$spec\Big[\!\Big[\text{class } C[\overline{X}] \text{ extends } I\{\text{val } \overline{f:T}; \overline{M}\}\Big]\!\Big] ::=$$
$$\Big\{\text{class } C[\overline{X}] \text{ extends } I\{\text{val } \overline{f:T};\ spec[\![\overline{M}]\!]_{C[\overline{X}]}\ ;\ ovs[\![\overline{M}]\!]_{C[\overline{X}]}\}\Big\}$$
$$\cup\ \{\ \text{class } C_s \text{ extends } \left|C[\overline{X}]\right|_s\ \ \{specimp[\![\overline{M}]\!]_s\}\ \mid s \in \mathcal{P}(\overline{X})\}$$
$$spec[\![M]\!]_{C[\overline{X}]} ::= \{fwd_s(M) \mid s \in \mathcal{P}(\overline{X})\}$$

$$fwd_s\ [\![\text{def } m[\overline{Y}](\overline{x:T}) = e]\!] ::=$$
$$\text{def } m_s[\overline{Y}](\overline{x} : \left|\overline{T}\right|_s) : |T|_s = (\text{this} : C[\overline{X}]).m(x_i.\text{as}[T_i]).\text{as}[|T|_s]$$
$$specimp[\![\text{def } m[\overline{Y}](\overline{x:T}) = e]\!]_s ::= \text{def } m_s[\overline{Y}](\overline{x} : \left|\overline{T}\right|_s) : |T|_s = [\![e]\!]_s$$

Figure 4.11: The *spec* translation

$$\frac{C[\overline{X}] \text{ extends } I \qquad D[\overline{P}] = overriddenFrom(m, I) \qquad D[\overline{Z}] \in \mathcal{D} \qquad s = \{\overline{Z \mapsto P}\}}{ovs[\![\text{def } m[\overline{Y}](\overline{x:T}) : T = e]\!]_{C[\overline{X}]} ::= \{\text{def } m_s[\overline{Y}](\overline{x:T}) : T = [\![e]\!]_s\ , \text{def } m[\overline{Y}](\overline{x:T}) : T = (\text{this} : C[\overline{X}]).m_s[\overline{Y}](\overline{x})\}} \text{(S-OVERRIDES)}$$

$$\frac{m \in C[\overline{X}]}{overriddenFrom(m, C[\overline{T}]) = C[\overline{T}]}$$

$$\frac{m \notin C[\overline{X}] \qquad C[\overline{X}] \text{ extends } D[\overline{U}]}{overriddenFrom(m, C[\overline{T}]) = [\overline{T}/\overline{X}]overriddenFrom(m, D[\overline{U}])}$$

Figure 4.12: Specialized overrides

$$
\begin{aligned}
TS[\![n]\!] &::= n \\
TS[\![x]\!] &::= x \\
TS[\![e.f]\!] &::= TS[\![e]\!].f \\
TS[\![(e : C[\overline{P_1}]).m[\overline{P_2}](\bar{e})]\!] &::= (TS[\![e]\!] : C[\overline{P_1}]).m_s(TS[\![\bar{e}]\!]) \\
&\quad \text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D}, \\
&\quad s = \{\overline{X} \mapsto \overline{P_1}, \overline{Y} \mapsto \overline{P_2}\} \\
TS[\![(e : C[\overline{P}]).m[\overline{T}](\bar{e})]\!] &::= (TS[\![e]\!] : C[\overline{P}]).m_s[\overline{T}](TS[\![\bar{e}]\!]) \\
&\quad \text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D}, s = \{\overline{X} \mapsto \overline{P}\} \\
TS[\![(e : C[\overline{T}]).m[\overline{P}](\bar{e})]\!] &::= (TS[\![e]\!] : C[\overline{T}]).m_s(TS[\![\bar{e}]\!]) \\
&\quad \text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D}, s = \{\overline{Y} \mapsto \overline{P}\} \\
TS[\![\text{new } C[\overline{P}](\bar{e})]\!] &::= \text{new } C_s(TS[\![\bar{e}]\!]) \\
&\quad \text{where } C[\overline{X}]...\{...\} \in \mathcal{D}, s = \{\overline{X} \mapsto \overline{P}\} \\
TS[\![e.\text{as}[T]]\!] &::= TS[\![e]\!].\text{as}[T] \\
TS[\![e : T]\!] &::= TS[\![e]\!] : T
\end{aligned}
$$

Figure 4.13: Specialized term translation

returns the instantiated type of the defining class D. The instantiated type of D is the type "as seen from" the subclass where overriding occurs. All type parameters of D are substituted for the types with which D was instantiated when inherited by C (transitively) (Figure 4.12).

Special overrides are created only if the overriding method inherits a fully specialized supertype. This constraint is formalized in S-OVERRIDES by requiring that the instantiated type returned by *overriddenFrom*, $D[\overline{P}]$, mentions only primitive types. If this condition doesn't hold, *ovs* is the identity. When *ovs* applies, it changes the overriding method m to call the specialized variant m_s, and a new method m_s is created in C, whose body is the specialized body of m. The forwarding call does not need any casts in this case, since the type of m_s is the same as m's.

4.3.4 Term rewriting

So far we have showed how generic definitions are translated to have specialized variants. The last step in specialization is to rewrite method calls and instantiations to use these specialized definitions, shown in Figure 4.13. The only interesting terms are method calls and object creation. Method calls can be completely specialized, when both the class and method type parameters

are primitive types. The call is rewritten towards the specialized variant using a specialization of all type parameters. Partial specializations are also possible, when only class or method type parameters are instantiated at primitive types. For a new object instance when all type parameters are primitive, TS rewrites it to its corresponding specialized subclass.

To sum everything up, we need to define what a BabyScala program is and extend the various definitions to operate on programs.

A BabyScala program is a collection of class definitions C and a term t. A well-formed program $P = (C, t)$ satisfies $\vdash$ *c is ok* for all classes in C, and $\vdash_{\mathcal{D}} t : T$ where D is the corresponding class table for C.

We extend the previous translations to programs in the natural way.

$$norm\left[\!\left[\text{class } C[\overline{X}] \text{ extends } I\{\overline{\text{val } f : T}; \overline{md}\}\right]\!\right] ::= \\ \text{class } C[\overline{X}] \text{ extends } I\{\overline{\text{val } f : T}; \overline{norm[\![md]\!]}\}$$

$$TS[\![\text{def } m[\overline{Y}](\overline{x : T}) : T = e]\!] ::= \text{def } m[\overline{Y}](\overline{x : T}) : T = TS[\![e]\!]$$

$$TS\left[\!\left[\text{class } C[\overline{X}] \text{ extends } I\{\overline{\text{val } f : T}; \overline{md}\}\right]\!\right] ::= \text{class } C[\overline{X}] \text{ extends } I\{\overline{\text{val } f : T}; \overline{TS[\![md]\!]}\}$$

$$spec[\![(\overline{c}, t)]\!] ::= (\overline{TS[\![spec[\![norm[\![c]\!]]\!]]\!]}, TS[\![t]\!])$$

The last line shows the order in which the various steps are performed: method expansion, class specialization and lastly term specialization.

4.3.5 Specialization preserves typing

In this section we give a formal proof that the transformation introduced previously preserves typing. More formally, given a well-typed BabyScala program, we prove that *spec* does not introduce any badly-typed definitions nor terms.

Theorem 1. *Given a BabyScala program P and a class table D, and that* $\vdash_{\mathcal{D}} P : ok.$, $spec(P) : ok.$

Before we delve into the proofs proper, we need a couple of lemmas on how type specialization interacts with *fields*, *mtype* and substitution.

Lemma 1 (Type Substitution)**.** *Given a specialization s not defined at any type variable in* $\overline{X}$, $|[\overline{T}/\overline{X}]U|_s = [|\overline{T}|_s/\overline{X}]|U|_s$.

Proof. See Appendix A. □

Lemma 2 (Field Specialization)**.** *Given a specialization s, a well-formed type* $C[\overline{T}]$, *and* $fields(C[\overline{T}]) = \overline{U\ f}$, *we have that*

$$fields\left(\left|C[\overline{T}]\right|_s\right) = \overline{|U|_s\ f}.$$

Proof. See Appendix A. □

Lemma 3 (Term Specialization)**.** *Given a valid class table D, a specialization s and a term e such that* $\Gamma \vdash e : T$*, we have*

$$|\Gamma|_s \vdash [\![e]\!]_s : |T|_s$$

Proof. See Appendix A. □

First we prove that normalization introduces only well-typed methods.

Theorem 2 (Normalization preserves typing)**.** *Given a well-formed class* $C[\overline{X}]$ $norm[\![C]\!]$ *is still well formed.*

Proof. We prove that $norm[\![M]\!]$ preserves typing.

We have that

$$M = \text{def } m[\overline{Y}](\overline{x : T}) : T = e, \text{ is well-formed}$$

and that

$$norm[\![M]\!] = \text{def } m_s(\overline{x : |T|_s}) : |T|_s = [\![e]\!]_s, \text{ for some } s = \{\overline{Y} \mapsto \overline{P}\}.$$

We prove that

$$\Gamma \vdash \text{def } m_s(\overline{x : |T|_s}) : |T|_s = [\![e]\!]_s \text{ is well-formed.}$$

According to the type checking rule

$$\frac{C[\overline{X}] \text{ extends } D[\overline{V}]\{...m..\} \in \mathcal{D} \qquad override(m, D[\overline{V}], \forall \overline{Y}.\overline{T} \to T) \qquad \overline{X}, \overline{Y}, \overline{x : T}, \text{this} : C[\overline{X}] \vdash e : T}{\vdash \text{def } m[\overline{Y}](\overline{x : T}) : T = e \; ok}$$

we need to first prove that $override(m_s, I, [\overline{Y}]\overline{T} \to T)$ holds. In other words, if there is a method with the same name m_s in C's superclass, it has the same type as m_s. If there is no such method, the proof is trivial.

Assume there is a method m_s in class D, the superclass of C. By construction, such a method exists iff there is a method m in D which is specialized using a specialization s'. Let that method be

$$\text{def } m[\overline{Y'}](\overline{x : U}) : U = e$$

The type of m is then $\forall \overline{Y'}.(\overline{U} \to U)$ and the type of $m_{s'}$ is $\left|\overline{U} \to U\right|_{s'}$.

Let

$$s = \left\{\overline{Y} \mapsto \overline{P}\right\}, \quad s' = \left\{\overline{Y'} \mapsto \overline{P}\right\}$$

Because the two specializations give rise to methods with the same name, both s and s' map to the same primitive types. They differ however in the name of

Let $s = \{X_i \mapsto P_i | i \in [1..n]\}$. Then

$$[\overline{Y}/\overline{X}]s = \left\{Y_i \mapsto P_i \mid \forall i, j \in [1..n], Y_i = Y_j \implies P_i = P_j\right\}$$

Substitution is defined only if all types $\overline{Y}$ are type variables, and there are no conflicting mappings.

$$s_1 = s_2 \text{ iff } s_1(x) = s_2(x)\ \forall x \in \text{dom}(s_1) \cup \text{dom}(s_2)$$

Figure 4.14: Substitution and equality on specializations

the type variables, as the overridden method may have named its own type parameters differently.

We need to prove that the type of $m_{s'}$ is the same as the type of m_s, seen from C.

$$\begin{aligned}
m_s : & \left|\overline{T} \to T\right|_s \\
&= \left|\forall\overline{Y}.(\overline{T} \to T)\right|_s && \text{by } spec \text{ definition} \\
&= \left|[\overline{V}/\overline{X}]\forall\overline{Y'}.(\overline{U} \to U)\right|_{s'} && \text{by } mtype \text{ definition} \\
&= [|\overline{V}|_{s'}/\overline{X}]\left|\forall\overline{Y'}.(\overline{U} \to U)\right|_{s'} && \text{by Lemma 1} \\
&= [|\overline{V}|_{s'}/\overline{X}]\left|(\overline{U} \to U)\right|_{s'} && \text{by definition of specialization}
\end{aligned}$$

$\overline{V}$ are the types used to instantiate the supertype of C and therefore cannot mention any type variable in $\overline{Y'}$. It follows that $\left|\overline{V}\right|_{s'} = \overline{V}$ and that

$$\left|\overline{T} \to T\right|_s = [\overline{V}/\overline{X}]\left|\overline{U} \to U\right|_{s'}$$

which is exactly what we needed to prove: the type of m_s is the same as the type of $m_{s'}$, when seen from C, therefore overriding is legal.

The last thing we need to prove is that the body of m_s is well-typed

$$\overline{X}, \overline{x : |T|_s}, this : C[\overline{X}] \vdash [\![e]\!]_s : |T|_s$$

This is follows directly from Lemma 3. □

Before we move to the next theorem, we need to explain overriding of specialized methods. We did not specify how method names are derived from a specialization s, and we are going to leave that as an implementation detail. However, we need to specify when two methods m_{s_1} and m_{s_2} have the same name for the purpose of overriding.

Definition 1. Assume m_{s_1} is defined in a class $C[\overline{X}]$, and m_{s_2} in $D[\overline{Y}]$ and $D[\overline{Y}]$ extends $C[\overline{V}]$. Then m_{s_2} overrides m_{s_1} iff $[\overline{V}/\overline{X}]s_1 = s_2$.

The definitions of substitution on specializations and equality between specialization is shown in Figure 4.14.

Lemma 4 (Substitution on specialization)**.** *Given a specialization s, $\overline{X}$ and $\overline{Y}$ type variables such that s is not defined anywhere in $\overline{Y}$, and $[\overline{Y}/\overline{X}]s$ is defined, we have*

$$\left|[\overline{Y}/\overline{X}]T\right|_{[\overline{Y}/\overline{X}]s} = |T|_s$$

Proof. See Appendix A. □

Theorem 3 ($spec[\![]\!]$ preserves typing)**.** *Given a well-typed program $P = (\overline{C}, T)$, $\vdash spec[\![C]\!]$ ok, $\forall C \in \overline{C}$.*

Proof. We begin by proving that $spec[\![M]\!]_{C[\overline{X}]}$ is well-typed. By definition, we have that

$$spec\left[\!\left[\text{def } m[\overline{Y}](\overline{x : T}) = e\right]\!\right]_{C[\overline{X}]} ::= \left\{fwd_s(M) \mid s \in \mathcal{P}(\overline{X})\right\}$$

$$\begin{aligned} fwd_s\left[\!\left[\text{def } m[\overline{Y}](\overline{x : T}) = e\right]\!\right] ::= \\ \text{def } m_s[\overline{Y}](\overline{x} : \left|\overline{T}\right|_s) : |T|_s = (\text{this} : C[\overline{X}]).m(x_i.\text{as}[T_i]).\text{as}[|T|_s] \end{aligned}$$

We prove that the new method definition is well-typed. We do so by using T-METHOD, and we need to prove that overriding is sound, and the body of m_s is well-typed. We begin with the last goal, as it is simpler:

$$\Gamma = \overline{X}, \overline{Y}, \overline{x : |T|_s}, this : C[\overline{X}] \vdash (\text{this} : C[\overline{X}]).m(x_i.\text{as}[T_i]).\text{as}[|T|_s] : |T|_s$$

We have

$$\begin{aligned} \Gamma \vdash \text{this} : C[\overline{X}] && \text{trivially} \\ mtype(m, C[\overline{X}]) = \forall \overline{Y}.\overline{T} \to T && \text{by definition of } fwd_s \\ \Gamma \vdash x_i.\text{as}[T_i] : T_i && \text{trivially} \end{aligned}$$

and the method call is well-typed.

In the following we assume that method type parameters do not change name through overriding. This has no impact on generality but keeps things simple, as there is one less step of alfa-renaming needed when looking at type equality.

We now prove that overriding is correct. If m_s overrides an inherited method $m_{s'}$, then they have the same type.

Let $C[\overline{X}]$ extend $B[\overline{V}]$ and $B[\overline{Z}]$ be the declaration of B. Let m' the original method for m'_s. We have that m overrides m' correctly (the initial program is well-typed, and they have the same name). Let the type of m' be $\overline{Y}.\overline{T'} \to T'$.

We have

$$
\begin{aligned}
[\overline{V}/\overline{Z}]s' &= s && \text{by overriding } m_s \quad (*)\\
[\overline{V}/\overline{Z}]\forall\overline{Y}.\overline{T'} \to T' &= \forall\overline{Y}.\overline{T} \to T && \text{by overriding } m'\\
\left|[\overline{V}/\overline{Z}]\forall\overline{Y}.\overline{T'} \to T'\right|_s &= \left|\forall\overline{Y}.\overline{T} \to T\right|_s && \text{specializing with } s\\
\left|[\overline{V}/\overline{Z}]\forall\overline{Y}.\overline{T'} \to T'\right|_{[\overline{V}/\overline{Z}]s'} &= \left|\forall\overline{Y}.\overline{T} \to T\right|_s && \text{by } (*)\\
\left|\forall\overline{Y}.\overline{T'} \to T'\right|_{s'} &= \left|\forall\overline{Y}.\overline{T} \to T\right|_s && \text{by Lemma 4}
\end{aligned}
$$

We have proved that the specialized methods have the same type, therefore overriding is legal.

The next step is to prove that $ovs[\![M]\!]_{C[\overline{X}]}$ preserves typing. We have

$$
\begin{aligned}
ovs\left[\!\left[\text{def } m[\overline{Y}](\overline{x:T}) : T = e\right]\!\right]_{C[\overline{X}]} ::= \\
\{&\text{def } m_s[\overline{Y}](\overline{x:T}) : T = [\![e]\!]_s\,,\\
&\text{def } m[\overline{Y}](\overline{x:T}) : T = (\text{this} : C[\overline{X}]).m_s(\overline{x})\}
\end{aligned}
\quad (4.1)
$$

and that

$$
C[\overline{X}] \text{ extends } I \qquad D[\overline{P}] = overriddenFrom(m, I) \qquad D[\overline{Z}] \in \mathcal{D} \qquad s = \{\overline{Z \mapsto P}\}
$$

Let m' be the method overridden by m, defined inside class $D[\overline{Z}]$. We have

$$
mtype(m', D[\overline{Z}]) = \forall\overline{Y}.\overline{T'} \to T'
$$

and by overriding rules

$$
[\overline{P}/\overline{Z}](\forall\overline{Y}.\overline{T'} \to T') = mtype(m, C[\overline{X}]) = \forall\overline{Y}.(\overline{T} \to T) \qquad (**)
$$

We notice that $[\overline{P}/\overline{Z}]$ and s have exactly the same effect on any type (they both substitute type variables in $\overline{Z}$ with the same primitive types).

$$
\begin{aligned}
mtype(m'_s, D[\overline{Z}]) = \left|\forall\overline{Y}.\overline{T'} \to T'\right|_s &= [\overline{P}/\overline{Z}](\forall\overline{Y}.\overline{T'} \to T')\\
&= \forall\overline{Y}.\overline{T} \to T && \text{by } (**)
\end{aligned}
$$

Since the type of m_s is $\forall\overline{Y}.\overline{T} \to T$, we conclude that m_s correctly overrides m'_s. Using T-METHOD, we need to prove that the body of m_s is well typed:

$$
\overline{X}, \overline{Y}, \overline{x:T}, this : C[\overline{X}] \vdash [\![e]\!]_s : T
$$

Because the initial method is well-typed, we have that

$$
\overline{X}, \overline{Y}, \overline{x:T}, this : C[\overline{X}] \vdash e : T
$$

and by Lemma 3, the body is well typed.

The second method generated by $ovs[\![]\!]$ keeps the original method type, but changes the body. Therefore, the overriding condition is trivially satisfied and we only need to prove that the body has type T

$$\Gamma = \overline{X}, \overline{Y}, \overline{(x : T)}, this : C[\overline{X}] \vdash (this : C[\overline{X}]).m_s[\overline{Y}](\overline{x}) : T$$

We trivially have that

$$\Gamma \vdash (this : C[\overline{X}]) : C[\overline{X}]$$
$$\Gamma \vdash \overline{x : T}$$

and

$$mtype(m_s, C[\overline{X}]) = \forall \overline{Y}.\overline{T} \to T$$

Using T-INVOKE we conclude that

$$\Gamma = \overline{X}, \overline{Y}, \overline{(x : T)}, this : C[\overline{X}] \vdash (this : C[\overline{X}]).m_s[\overline{Y}](\overline{x}) : T$$

and furthermore, that $ovs[\![]\!]$ preserves typing.

Let's have a look at the class definition where these translations are used:

$$spec\Big[\!\Big[\text{class } C[\overline{X}] \text{ extends } I\{\text{val } \overline{f : T}; \overline{M}\}\Big]\!\Big] ::=$$
$$\Big\{\text{class } C[\overline{X}] \text{ extends } I\{\text{val } \overline{f : T};\ spec[\![\overline{M}]\!]_{C[\overline{X}]}\ ;\ ovs[\![\overline{M}]\!]_{C[\overline{X}]}\}\Big\} \cup \{\ \ldots\} \quad (4.2)$$

We have proved so far that all method definitions in class $C[\overline{X}]$ are well typed. In order to prove that the class if well-typed, we still have to prove (according to T-CLASS):

$$\overline{X} \vdash I, \overline{T}\ ok. \qquad fields(I) = \overline{g : U} \qquad \overline{f}, \overline{g}\ distinct$$

$spec[\![]\!]$ does not change the supertype, nor the field names or types. Since the original class was well-typed, these conditions are trivially satisfied.

The last thing we need to prove is that the specialized implementation classes are also well typed:

$$\{\ \text{class } C_s \text{ extends } \left|C[\overline{X}]\right|_s\ \ \{specimp[\![\overline{M}]\!]_s\}\ \mid s \in \mathcal{P}(\overline{X})\}$$

According to T-CLASS, we have to prove

$$\vdash |C[\overline{X}]|s\ ok. \qquad \overline{md}\ ok\ in\ C_s$$

and since C_s has no fields, we only have to prove the superclass is a well-formed type, and that methods are well-typed.

A specialization maps type variables to primitive types, which are trivially well-formed. It is immediate that, if $\overline{X} \vdash C[\overline{X}]\ ok$, $|C[\overline{X}]|_s$ is also well-formed.

We prove that

$$specimp[\![\text{def } m[\overline{Y}](\overline{x : T}) = e]\!]_s ::= \text{def } m_s[\overline{Y}](\overline{x} : \left|\overline{T}\right|_s) : |T|_s = [\![e]\!]_s$$

is well typed. We first prove that overriding is correct. We need to prove

$$mtype(m_s, C[\overline{X}]) = mtype(m_s, C_s)$$

We have that s takes all values from $\mathcal{P}(\overline{X})$, and that $spec[\![M]\!]$ generates a forwarding method m_s for each $s \in \mathcal{P}(\overline{X})$. Therefore, there must be a method m_s in $C[\overline{X}]$ using the same specialization s.

$$\begin{aligned} mtype(m_s, C[\overline{X}]) &= \forall \overline{Y}.|\overline{T}|_s \to |T|_s && \text{by definition of } spec[\![]\!] \\ mtype(m_s, C_s) &= \forall \overline{Y}.|\overline{T}|_s \to |T|_s && \text{by definition of } specimp[\![]\!]. \end{aligned}$$

Next we prove that the body of m_s is well typed

$$\overline{Y}, \overline{x : |T|_s}, this : C_s \vdash [\![e]\!]_s : |T|_s$$

We know that the original method, m, is well-typed and we have

$$\Gamma = \overline{X}, \overline{Y}, \overline{x : T}, this : C[\overline{X}] \vdash e : T$$

Using Lemma 3 we get

$$|\Gamma|_s \vdash [\![e]\!]_s : |T|_s$$

This is almost what we need, but we have to take care of the environment Γ.

$$|\Gamma|_s = \overline{Y}, \overline{x : |T|_s}, \text{this} : |C[\overline{X}]|_s$$

We omitted type parameters $\overline{X}$, because s replaces them by primitive types, and the environment contains only type variables and bindings. The type environment under which we have to type-check m_s's body differs in the type of *this*: instead of $|C[\overline{X}]|_s$ it has type C_s. However, we have that

$$\text{class } C_s \text{ extends } \left|C[\overline{X}]\right|_s$$

And by using T-SUBSUMPTION we have this : $|C[\overline{X}]|_s$. Therefore, by Lemma 3 we can prove that $[\![e]\!]_s : |T|_s$ under *weaker* assumptions than we are given. Therefore,

$$\overline{Y}, \overline{x : |T|_s}, this : C_s \vdash [\![e]\!]_s : |T|_s$$

and $specimp[\![]\!]$ preserves typing.

We have proven that all classes generated by $spec[\![\text{class}C[\overline{X}] \ldots]\!]$ are well typed. □

We can now turn to the last step of the specialization translation, $TS[\![]\!]$.

Lemma 5. *Given two specializations* $s_1 = \left\{\overline{X \mapsto P_1}\right\}$ *and* $s_2 = \left\{\overline{Y \mapsto P_2}\right\}$, $\overline{X} \cap \overline{Y} = \emptyset$, *we have*

$$\left||T|_{s_1}\right|_{s_2} = |T|_{s_1 \oplus s_2} = [\overline{P_2}/\overline{Y}][\overline{P_1}/\overline{X}]T = [\overline{P_2}, \overline{P_1}/\overline{Y}, \overline{X}]T$$

Proof. Specializations are just a special case of substitutions. If the domains are distinct, they can be applied in any order (they are commutative). The conclusion is proven by straight-forward induction on the structure of types. □

Theorem 4. *Given a well-typed program $P = (\overline{c}, e)$, $spec[\![P]\!]$ is well-typed, and $TS[\![e]\!]$ has the same type as e.*

Proof. We have

$$spec[\![(\overline{c}, t)]\!] ::= (\overline{TS[\![spec[\![norm[\![c]\!]]\!]]\!]}, TS[\![t]\!])$$

and we prove that if $\Gamma \vdash e : T$,

$$\Gamma \vdash TS[\![e]\!] : T$$

We use induction on the structure of terms.

- n Trivial.
- x Trivial.
- $e.f$
  We have
  $$\Gamma \vdash e.f : T$$
  We prove
  $$\Gamma \vdash TS[\![e]\!].f : T$$
  By Induction Hypothesis we have
  $$\Gamma \vdash TS[\![e]\!] : T_1$$
  where T_1 is the type of e. By T-FIELD we have that
  $$\Gamma \vdash TS[\![e]\!].f : T$$
- $(e : I).m[\overline{U}](\overline{e})$ We have the following subcases, according to the definition of $TS[\![]\!]$:
  - $\overline{U} = \overline{P}$
    We have
    $$TS[\![(e : I).m[\overline{P}](\overline{e})]\!] ::= (TS[\![e]\!] : I).m_s(TS[\![\overline{e}]\!])$$
    $$\text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D},$$
    $$s = \{\overline{Y} \mapsto \overline{P}\}$$

Let $mtype(m, I) = \forall\overline{Y}.(\overline{T} \to \overline{T})$, and the type at the call be $[\overline{P}/\overline{Y}]\overline{T} \to \overline{T} = \overline{T_1} \to T_1$. The initial call is well-typed, and by the Induction Hypothesis we have

$$\Gamma \vdash TS[\![e]\!] : I \qquad \Gamma \vdash TS[\![\bar{e}]\!] : \overline{T_1}$$

We prove that

$$\Gamma \vdash (TS[\![e]\!] : I).m_s(TS[\![\bar{e}]\!]) : T_1$$

We have the receiver and argument types, and we need to prove that the call is well-typed according to T-INVOKE, in other words, that the type of m_s is $\overline{T_1} \to T_1$.
By definition,

$$\begin{aligned} &norm[\![\text{def } m[\overline{Y}](\overline{x : T}) = e]\!] ::= \\ &\quad \text{def } m[\overline{Y}](\overline{x : T}) = e \cup \left\{\text{def } m_s(\overline{x} : |T|_s) = [\![e]\!]_s \mid s \in \mathcal{P}(\overline{Y})\right\} \end{aligned} \tag{4.3}$$

therefore there is one method for specialization s in the same class, whose type is $|\overline{T}|_s \to |T|_s$.
Specializations are just a special case of type substitutions, so we have

$$\begin{aligned} |\overline{T}|_s \to |T|_s &= [\overline{P}/\overline{Y}]\overline{T} \to [\overline{P}/\overline{Y}]T \\ &= \overline{T_1} \to T_1 \end{aligned}$$

Therefore, $mtype(m_s, I) = \overline{T_1} \to T_1$.

– $\overline{T} = \overline{P}$

We have

$$\begin{aligned} TS[\![(e : C[\overline{P}]).m[\overline{U}](\bar{e})]\!] ::= (TS[\![e]\!] : C[\overline{P}]).m_s[\overline{U}](TS[\![\bar{e}]\!]) \\ \text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D}, \\ s = \{\overline{X} \mapsto \overline{P}\} \end{aligned}$$

We prove

$$\Gamma \vdash (TS[\![e]\!] : C[\overline{P}]).m_s[\overline{U}](TS[\![\bar{e}]\!]) : T_1$$

where $\overline{T_1} \to T_1$ is the type of m_s at application point, $[\overline{P}, \overline{U}/\overline{X}, \overline{Y}]\overline{T} \to T$.
The initial term is well-typed, and by the Induction Hypothesis we have

$$\Gamma \vdash TS[\![e]\!] : C[\overline{P}] \qquad \Gamma \vdash TS[\![\bar{e}]\!] : \overline{T_1}$$

Using T-INVOKE, we are left to prove that m_s exists and

$$mtype(m, C[\overline{P}]) = mtype(m_s, C[\overline{P}]) = [\overline{P}/\overline{X}](\forall\overline{Y}.\overline{T} \to T)$$

We have

$$spec[\![\text{def } m[\overline{Y}](\overline{x:T}) = e]\!]_{C[\overline{X}]} ::= \{fwd_s(M) \mid s \in \mathcal{P}(\overline{X})\}$$

$$fwd_s\,[\![\text{def } m[\overline{Y}](\overline{x:T}) = e]\!] ::= \\ \text{def } m_s[\overline{Y}](\overline{x} : |\overline{T}|_s) : |T|_s = (\text{this} : C[\overline{X}]).m(x_i.\text{as}[T_i]).\text{as}[|T|_s]$$

therefore, $spec[\![]\!]$ generates a method for each possible specialization, including s:

$$\begin{aligned} mtype(m_s, C[\overline{X}]) &= \forall\overline{Y}.|\overline{T}|_s \to |T|_s \\ &= \forall\overline{Y}.([\overline{P}/\overline{Y}]\overline{T} \to [\overline{P}/\overline{Y}]T) \\ &= mtype(m, C[\overline{P}]) \\ &= mtype(m_s, C[\overline{P}]) \end{aligned}$$

– $\overline{T} = \overline{P_1}, \overline{Y} = \overline{P_2}$
We have

$$\begin{aligned} TS[\![(e : C[\overline{P_1}]).m[\overline{P_2}](\overline{e})]\!] ::= &\ (TS[\![e]\!] : C[\overline{P_1}]).m_s(TS[\![\overline{e}]\!]) \\ &\text{where } C[\overline{X}]...\{..\text{def } m[\overline{Y}](\overline{x} : \overline{T}) : T = e..\} \in \mathcal{D}, \\ &s = \{\overline{X} \mapsto \overline{P_1}, \overline{Y} \mapsto \overline{P_2}\} \end{aligned}$$

Let $s = s_1 \oplus s_2$ and $s_1 = \{\overline{Y} \mapsto \overline{P_2}\}$, $s_2 = \{\overline{X} \mapsto \overline{P_1}\}$. We prove that

$$\Gamma \vdash (TS[\![e]\!] : C[\overline{P_1}]).m_s(TS[\![\overline{e}]\!]) : [\overline{P_1}/\overline{X}][\overline{P_2}/\overline{Y}]T$$

Let $mtype(m, C[\overline{P_1}]) = \forall\overline{Y}.\overline{T_1} \to T_1$ and the type at the application be $[\overline{P_2}/\overline{Y}](\overline{T_1} \to T_1) = \overline{T_2} \to T_2$. By Induction Hypothesis we have that

$$\Gamma \vdash TS[\![e]\!] : C[\overline{P_1}] \qquad \Gamma \vdash TS[\![\overline{e}]\!] : \overline{T_2}$$

We need to prove that the type of m_s in $C[\overline{P_1}]$ is $\overline{T_2} \to T_2$. We have from the definition of $norm[\![]\!]$ and $spec[\![]\!]$:

$$norm\,[\![\text{def } m[\overline{Y}](\overline{x:T}) = e]\!] ::= \quad \left\{\ldots \text{def } m_{s_1}(\overline{x} : |T|_{s_1}) = [\![e]\!]_{s_1}\right\}$$

$$spec[\![\text{def } m_{s_1}(\overline{x:T_1}) = e]\!]_{C[\overline{X}]} ::= \{fwd_s(M) \mid s \in \mathcal{P}(\overline{X})\}$$

$$fwd_{s_2}\,[\![\text{def } m_{s_1}(\overline{x:T_1}) : T_1 = e]\!] ::= \\ \text{def } m_s(\overline{x} : |\overline{T}|_s) : |T|_s = (\text{this} : C[\overline{X}]).m(x_i.\text{as}[T_i]).\text{as}[|T|_s]$$

where we use the fact that $s = s_1 \oplus s_2$ and that $T_1 = |T|_{s_1}$. Using the fact that s_1 and s_2 are disjunct because they are in different scopes, we have that

$$\begin{aligned} mtype(m_s, C[\overline{X}]) &= |\overline{T}|_s \to |T|_s \\ &= \left||\overline{T}|_{s_1}\right|_{s_2} \to \left||T|_{s_1}\right|_{s_2} \\ &= [\overline{P_1}/\overline{X}][\overline{P_2}/\overline{Y}]\overline{T} \to [\overline{P_1}/\overline{X}][\overline{P_2}/\overline{Y}]T \quad = \overline{T_2} \to \overline{T_1} \end{aligned}$$

Using T-INVOKE we conclude that the method call is still well-typed.

- new $C[\overline{P}](\bar{e})$

  We have
  $$\Gamma \vdash \text{new } C[\overline{P}](\bar{e}) : C[\overline{P}]$$
  We prove
  $$\Gamma \vdash TS[\![\text{new } C[\overline{P}](\bar{e})]\!] : C[\overline{P}]$$
  Using the definition of $TS[\![]\!]$ we have that
  $$TS[\![\text{new } C[\overline{P}](\bar{e})]\!] ::= \text{new } C_s(TS[\![\bar{e}]\!])$$
  and we prove
  $$\Gamma \vdash \text{new } C_s(TS[\![\bar{e}]\!]) : C[\overline{P}]$$
  Using T-NEW, we need to prove that
  $$\Gamma \vdash C_s \textit{ is ok} \qquad fields(C_s) = \overline{f : T} \qquad TS[\![\bar{e}]\!] : \overline{T}$$
  C_s *is ok* follows directly from Theorem 3. From the definition of $spec[\![]\!]$ we know that C_s does not define any new fields, therefore $fields(C_s) = fields(C[\overline{P}]) = \overline{f : T}$. By Induction Hypothesis we have
  $$\Gamma \vdash TS[\![\bar{e}]\!] : \overline{T}$$
  Using T-NEW we have
  $$\Gamma \vdash \text{new } C_s(TS[\![\bar{e}]\!]) : C_s$$
  From the definition of $spec[\![]\!]$ we have that
  $$\begin{aligned} &spec\left[\!\left[\text{class } C[\overline{X}] \text{ extends } I\{\text{val } \overline{f : T}; \overline{M}\}\right]\!\right] ::= \\ &\quad \left\{\ldots \text{ class } C_s \text{ extends } \left|C[\overline{X}]\right|_s \ \left\{specimp\left[\!\left[\overline{M}\right]\!\right]_s\right\} \mid s \in \mathcal{P}(\overline{X})\right\} \end{aligned}$$
  therefore $C_s <: C[\overline{P}]$, where $s = \{\overline{X} \mapsto \overline{P}\}$. Using T-SUBSUMPTION we have
  $$\Gamma \vdash \text{new } C_s(TS[\![\bar{e}]\!]) : C[\overline{P}]$$

```
class C[T] {
  private var f: T

  def f: T
  def f_=(x: T): Unit
}
```

```
class C$Int extends C[Int] {
  private var f$: Int

  def f: Int = f$Int
  def f_=(x: Int): Unit = f_=$Int(x)
  def f$Int: Int = this.f$
  def f_=$Int(x: Int): Unit = this.f$ = x
}
```

Figure 4.15: Field Specialization

- $e.\text{as}[T]$. Trivially using the Induction Hypothesis.
- $e : T$. Trivially using the Induction Hypothesis. □

In this section we have shown that specialization preserves types. The natural next question is whether specialization preserves semantics as well. While this may be the case for BabyScala, in the presence of type erasure (and the full Scala language) this is not the case. For example, before specialization, and because of type erasure, a program cannot distinguish between `List[Int]` and `List[String]`. After specialization, the two types have different classes, and a type test would distinguish them. It would be interesting to explore for what subset of the Scala language semantics are preserved. We leave a detailed discussion of this topic for future work.

4.4 Implementation

The formal treatment introduced so far describes well how specialization interacts with the core of the Scala language. However, as any calculus, BabyScala glosses over many language features that may nevertheless interact with the transformation we propose. Inheritance, field overriding and (arbitrary) constructors need to be properly treated when implementing specialization for the Scala compiler. In the following sections we describe the challenges of treating these additional language features.

4.4.1 Field specialization

We have shown how methods are specialized, but class fields benefit from specialization as well. Objects may store generic data, and having a specialized representation for them may save numerous boxing operations.

Specialization is fundamentally based on overriding, and Scala allows field overriding through a translation that turns all field accesses in getter/setter operations. Specialization on getters and setters proceeds as usual method specialization. The only additional operation is to add a field of the specialized

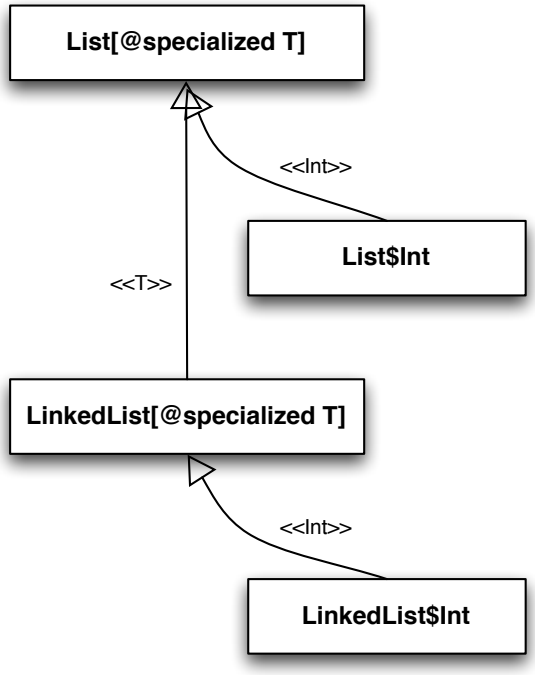


Figure 4.16: Specialized inheritance

type in the specialized subclass, and rewire the specialized getters to operate on the new field. The original field in the generic version exists but it is inaccessible in a specialized class. Figure 4.15 shows how this happens in practice.

4.4.2 Specialized inheritance

A specializable class may extend another specializable class. When the supertype is instantiated using a specialized type variable, we deal with *specialized inheritance*. The intention is that the subclass is specialized itself, and inherits the specialized representation of its superclass. Consider the following example:

```
class List[@specialized A] {
..
}
class LinkedList[@specialized A] extends List[A] {
..
}
```

Figure 4.16 shows the generated classes and their relationship. Each class generates a specialized subclass, `LinkedList$Int` and `List$Int`. The problem becomes clearer when we ask what should be the superclass of `LinkedList$Int`: according to our transformation, it is the generic `LinkedList` (to allow mixing specialized and unspecialized code). If we follow the inheritance chain, we notice the superclass of `LinkedList` is `List`, not `List$Int`, therefore it does not inherit the specialized representation of plain lists. Moreover, inherited methods would require boxing, even in the case of a specialized instance such as `LinkedList$Int`. While this does not break anything, it is not what a programmer expects, and it implies a certain performance penalty.

This limitation stems from our use of overriding for rerouting existing methods and field accesses. Whenever a specialized instance is created, the original methods are overridden to use the specialized representation. However, there is one thing that cannot be overridden on the Java Virtual Machine: the inheritance relationship.

One solution is to override all specialized variants inherited from `List` in the context of `LinkedList$Int` It turns out that exactly the same effect can be achieved using Scala's mechanism for multiple inheritance (mix-in composition in Scala). Whenever a specialized type parameter is used in a supertype of a specialized class, we mix-in again that type in a specialized subclass. In our example, `LinkedListInt` becomes:

```
class LinkedList$Int
  extends LinkedList[Int] with List$Int {
..
}
```

Scala mix-in composition requires that `List` is a trait. The Scala compiler issues a warning when it encounters specialized inheritance involving a class instead of a trait, since inherited members in the specialized subclass require boxing.

4.4.3 Specialized instance initialization

BabyScala constructors are hard wired, and consist solely of field initialization. However, a class definition in Scala contains member definitions, freely interspersed with arbitrary statements. They are evaluated top to bottom when a new instance is created, and field values are available immediately after their definition point:

```
abstract class Stack[@specialized T](size: Int) {
  val data = new Array[T](size)

  println("created array of size " + data.length)

  def push(x: T)
  def pop: T
}
```

```
abstract class Stack$Int extends Stack[Int] {
  val data$Int: Array[Int] = _
  override def data = data$Int

  def this(size: Int) {
    super.this(size)
    this.data$Int = new Array[Int](size)
  }
  // ...
}
```

Figure 4.17: Specialized instance creation

The compiler groups all initialization code in one method, consisting of expressions and field initializers. Each constructor method first calls the super class constructor, thus making sure it is itself a valid instance of the super class. Here's the stack class again, this time with an explicit constructor, as created by the compiler in later phases:

```
abstract class Stack[@specialized T] {
  val data: Array[T] = _

  def this(size: Int) {
    super.this()
    this.data = new Array[T](size)
    println("created array of size " + data.length)
  }

  def push(x: T)
  def pop: T
}
```

Figure 4.17 shows one specialized subclass of `Stack`. Unsurprisingly, it has its own constructor, that initializes its own (specialized) fields. In this example, `data` is the only field specialized in `Stack$Int`. Similar to method specialization, a field is specialized by overriding its accessors towards its specialized representation (all Scala fields are accessed through getters and setters, regardless of specialization). The specialized constructor calls the superclass constructor before it initializes its own fields. The problem is that the super class constructors expects that `data` has been initialized already, and tries to get its length. At the same time, `Stack$Int` has overriden the getter for `data` towards the specialized variant, resulting in an unitialized field access.

There is another, less obvious problem with this example: the initializer for `data` is evaluated twice: once in the superclass constructor, for the generic version of the field, and another time in the specialized subclass. This is a problem, since the right-hand side of a value definition can have arbitrary code.

```
class Stack[@specialized T] {
  def specInstance$: Boolean = false

  def this(size: Int): Stack = {
    super.this();
    if (!this.specInstance$) {
      Stack.this.data = new Array[T](size)
      println("created array of size "
        + data.length)
    }
  }
//..
}
```

```
class Stack$Int extends Stack[Int] {
  override def specInstance$: Boolean = true

  def this(size: Int): Stack = {
    super.this(size);
    Stack.this.data$Int = new Array[Int](size)
    println("created array of size "
      + data.length)
  }
  // ..
}
```

Figure 4.18: Specialized initialization solution

We need to solve two problems: field initializers should be evaluated only once, and instance initialization should not see uninitialized fields introduced by specialization. We notice that the uninitialized field problem may arise only during object construction, when expressions depend on fields of the object being initialized.

The first problem is easier to solve: we simply need to distinguish between specialized and generic instances, and guard the constructing code with the specific test. To solve the second problem, we move the constructor code to the specialized subclass. The generic and specialized classes are two sides of the same definition, so we can freely move code from one to the other. By copying all the constructor code from the superclass to the subclass we are guaranteed to have the same behavior regarding data dependencies between fields and other statements. The only case in which behavior is changed as a result is if the constructor uses reflective calls. In such a case, it could for instance reveal that the current class name is different from the source-level name.

The Scala compiler generates the code shown in Figure 4.18. Notice the additional method `specInstance$` that allows the superclass to decide whether to evaluate or not its own constructor. The specialized subclass overrides it to signal that the current instance is specialized. We could have used reflection, or `isInstance` calls, but we opted for this solution as it is more efficient.

The specialized constructor merges the superclass expressions with specialized field initialization:

$$c_i = \begin{cases} \text{this}.f_s = [\![e_i]\!]_s & \text{if } g_i \text{ is this}.f = e_i \text{ (initialization of a specialized field)} \\ g_i & \text{otherwise} \end{cases}$$

The specialized constructor statement at position i is denoted by c_i, a specialized field by f_s and a generic field by f. We use the initialization of a specialized

field f_s when the original statement in the generic constructor is an initialization of a generic field f. In the other cases (normal field initialization or plain statements), the specialized constructors simply copies over the corresponding statement.

4.4.4 Type bounds

Type parameters in Scala may have lower and upper bounds which restrict the applicable types when the corresponding class or method is instantiated. Type parameters of classes are simple to handle: we only generate specialized subclasses for primitive types that fall between the bounds. Things get interesting when we look at method type parameters, in particular when their bounds involve type parameters of the enclosing class.

Consider the following:

```
def m[@specialized B >: Lo <: Hi, C](x: B, y: C): B
```

This definition has two type parameters, `B` and `C`. Parameter `B` has type bounds `Lo` and `Hi`, which means that all instantiations of `B` have to be supertypes of `Lo` and subtypes of `Hi`. We can derive the expanded method definitions of some method by iterating over all combinations of its specialized parameters (only `B` in this example), keeping only those that fall between the bounds. If we assume that `Int` falls between the bounds, this gives

```
def m[B >: Lo <: Hi, C](x: B, y: C): B
def mInt[C](x: Int, y: C): Int
```

The question is what to do otherwise, and here we distinguish two cases:

- satisfiable: The bounds of a type parameter mention a specialized type parameter of the enclosing class. We cannot conclude that any concrete type satisfies its bounds until the enclosing class is instantiated.
- conflicting: The bounds of a type parameter clearly forbid a concrete type combination.

To understand why we need this distinction, we look at the way linked lists are defined in the standard library

```
class List[@specialized A] {
def ::[@specialized B >: A](x: B): List[A] =
  new ::(x, this)
..
}
```

We notice that `Int` is not a valid specialization for the cons operation (`::`), because `Int` is not a supertype of `A`, for all types `A`. However, a cons operation specialized for `Int` makes sense when working on a `List[Int]`. By noticing that `A` is also specialized, and that the constraint may be fulfilled when specializing `List`, we let expansion generate a specialized variant for `Int`.

Expansion generates only satisfiable variants. Conflicting combinations give a compile-time warning, since most probably this is not intended.

What should go into the body of expansions? We distinguish again between two cases: a valid expansion is implemented by rewriting the original body with the valid type bindings. By contrast, a satisfiable expansion cannot be implemented the same way, as that would yield type-incorrect code (remember that satisfiable methods are instantiated at types that do *not* fall between the bounds). Therefore it is implemented by delegation to the original method.

4.4.5 Selection

Specialization is not needed on all members. A class may have many methods that do not operate on generic types, and whose specialization would be a waste of space and time, with no obvious gain. In the implementation we chose to specialize only those members that mention a specialized type parameter in their signature. For example,

```
abstract class Foo[@specialized T, U] {
  def apply(x: T): U
  def get(x: Int): Array[T]
  def foreach(x: T => Unit)
  val a: Array[T]

  def bar1(x: U): Unit
  def bar2(x: List[T]): U
  val b: List[T]
}
```

In class `Foo` it makes sense to specialize the first four members, since calling any of them involves boxing. However, it makes little sense to specialize `bar2`, for example, whose parameter is `List[T]`, even though `T` is specialized. The reason is that the argument to `bar2` is an object, therefore no boxing is involved.

The important observation is that, besides naked specialized type parameters, we need to specialize members that mention specialized type parameters in specialized positions inside a generic class. Such is the case for method `foreach`, that does not have a naked type parameter in its signature, but would still benefit from specialization. The reason is that the parameter `T => Unit` is in fact a generic type, `Function1[Unit, T]`, and `T` appears inside `Function1` at a specialized type parameter position. Therefore, by specializing `foreach`, we enable specialized use of the argument `f`.

In general, a member is specialized if at least one of the following is true:

- its return type or one of the parameter types is a specialized type variable.
- its return type or one of the parameter types is a polymorphic type `C[T1,..]` with specialized type parameters, and at least one specialized type parameter `T` is used in C's instantiation at a specialized position

Arrays are considered to be defined as `Array[@specialized T]`, so the last rule selects for specialization members `get` and `a` in the above example.

Chapter 5

Evaluation

> Measure a thousand times and cut once
>
> Turkish proverb

5.1 Introduction

Language features such as closures and generics make code more expressive at the expense of performance. In Chapter 2 we have identified and presented several areas where performance can be improved: boxing and unboxing, higher-order functions and generic arrays. We have implemented the techniques described in Chapter 4 in the current version of the Scala compiler[1]. In the following sections we use a series of micro-benchmarks in order to assess how well the techniques described so far are actually improving the performance of Scala programs.

5.2 Methodology

We use two criteria for evaluating how well the compiled program performs. Naturally, the first thing we are interested in is speed: how fast is the resulting program compared to a baseline (unoptimized/unspecialized) program. The second criteria is code size: specialization generates new class definitions thus increasing code size, while optimization may remove unused classes completely.

Measuring performance of the Java Virtual Machine is notoriously difficult [20, 19]. Garbage collection, just-in-time compilation and dynamic optimization interact in non-obvious ways with the program under inspection. The VM

[1] At the time of this writing, the Scala compiler version 2.8.0 is not yet released.

loads the program and starts executing bytecode in interpreter mode. At some point during execution the VM may decide that some method should be compiled to native code to improve performance (usually, the method is deemed a "hotspot" if it has been executed a large number of times, e.g. 10,000 times). At this moment the VM compiles and optimizes the code, using everything it knows about the code currently loaded in the VM. Some of these assumptions, for instance that a certain method is never overridden (even though it's not marked **`final`**), may be invalidated in the future. For example, the VM loads a new class that overrides that method, triggering de-optimization of existing compiled code. Measuring the execution time under this scenario would be misleading, since we'd be really measuring the interpreter, the just-in-time compiler and the de-optimizer, in addition to the real benchmark program.

We can roughly caracterize the VM state as being either "cold" or "warm": the VM starts cold, loading classes, compiling and optimizing hot spots, possibly de-optimizing code based on too optimistic assumptions. At the point where these activities are down to a minimum, the VM reaches maximum performance: we say the VM is "warm". It is meaningful to measure performance in both states, as long as the times we compare are measured in the same state.

Micro-benchmarks are designed to exercise a small set of features, like higher-order functions. Typically, they have a tight loop performing some computation whose result is not really needed. The problem is that the VM may be smart enough to dead-code eliminate the computation if the result is not used in any way. However, printing or writing to a file may distort the measurements in a significant way, so they should not be part of the actual benchmark. All of the benchmarks presented in this chapter have a step that prints the result, outside the measuring loop, thus making sure we are actually measuring what we are interested in.

Each measurement is repeated 5 times, and we use the mean as the single-value performance metric. There have been several metrics used in literature,e.g. best, second-best or worst-time [31], but in Georges et al. conclude that the mean gives the most realistic picture of how the program will behave [18] . Unless otherwise stated, all benchmarks are run on an Intel Core 2 Duo 2.5 GHz with 4 GB of RAM, running Mac OS X 10.6.4 and Java 1.6.0_20, 64-bit server VM.

5.2.1 Steady-state performance

Steady-state performance measures how well the program performs on a warm VM. We warm up the VM by running the benchmark payload 2 times, and garbage collect between runs. We run the program another 5 times in the same VM, this time measuring the execution time. Before each iteration we force a garbage collection run. To check that no additional compilation takes place, we enable a VM flag to log when methods are jit-compiled[2]. Note that even

[2]We use `-XX:+PrintCompilation`.

though it is very likely that the garbage collector runs *during* the benchmark payload, it is not a problem: the GC cost is part of the overall performance.

5.2.2 Start-up performance

Start-up performance measures how well the program performs on a cold VM. In this scenario we measure the time needed to execute the test *in one invocation of the VM*, over 7 runs. We do not consider the first two measurements, which are a warm-up of the operating system (loading from disk, paging, etc). We average over the following 5 times and provide the standard deviation as a measure of the range we witnessed. Even though this measurement includes dynamic compilation, class loading and optimization, it gives an overall picture of how the start-up time of a program is affected by our optimizations. For instance, specialization may generate a large number of methods and classes, and class-loading time may increase significantly. This metric shows whether this is the case, and the assumption is that the baseline numbers include the same components beside actual program execution.

5.2.3 Code size

We are interested in the effect on compiled code size. The Java bytecode format [32] is verbose: each class has its own file, with its own symbol table and linking is done symbolically, using fully qualified names. We check the impact of new classes and methods by measuring the size on disk.

A number of tools alleviate the cost of the classfile format by applying several specific compression techniques, including symbol table sharing. We use `pack200`, the tool distributed with the standard Java environment, to give a more realistic view of the the class size when loaded by the VM (we can assume that the VM shares symbol table entries).

5.3 Specialization

This section presents the results we obtain by using only code specialization.

5.3.1 Benchmark suite

The benchmark suite consists of 6 programs, each exercising some specific aspect of the Scala language:

matrix A straight-forward implementation of matrix multiplication. The matrix class is generic in the element type. We multiply two random matrices of 250x150 integers.

fft A simple implementation of the fast Fourrier transform. Complex numbers are represented as pairs of double precision numbers. The `Pair` class is generic in its element type. The input is 65,536 data points.

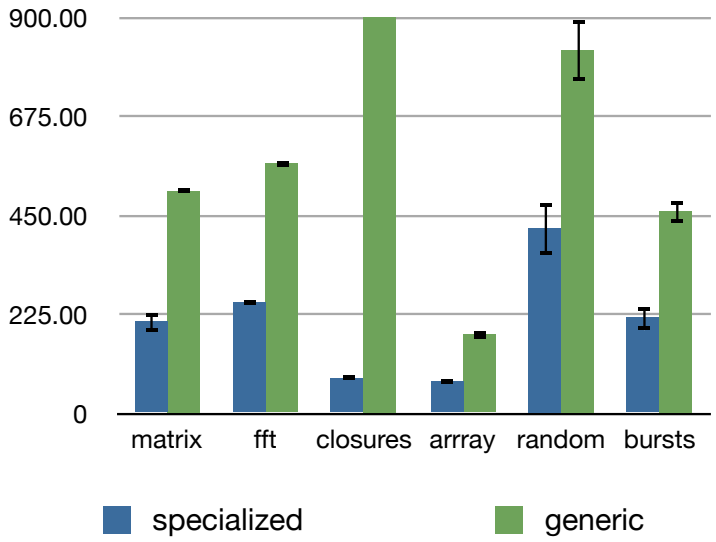


Figure 5.1: Steady-state performace of specialization

closures A program that applies a function literal computing $x^2 + 1$ to each element of a 10,000 element array and then summing up all the elements.

arraybuf The next three benchmarks are all based on a specialized implementation of `ArrayBuffer`. The buffer is backed by an array and can shrink and grow as needed. It provides higher-order operations like `map` and `fold`.

- ops The first test creates a buffer of 1,000,000 integers and then performs an in-place map, a reverse and a fold-left to sum up all its elements.
- reads Creates a 1,000,000 element buffer and performs 10 million random reads.
- burst Same as before, but the 10 million random reads are distributed in 1 million bursts of 10 consecutive location reads.

Steady-state

Figure 5.1 shows how performance of each benchmark compares to the baseline (same code, without specialization). The speed is measured in milliseconds and the error bars show one standard deviation of the data set. As described in § 5.2, each data point is the mean of the last 5 runs out of 7 runs per JVM invocation.

Figure 5.2 shows the average running times and speedup. We notice that all programs are around 2 to 2 1/2 faster with specialization, with one notable outlier. The closures benchmark is a massive 35 times faster when specialized! Most of the time spent by the "closures" benchmark is in boxing (around 200

	Generic	Specialized	Speedup (Nx)
matrix	506.00	209.20	2.42
fft	568.20	252.60	2.25
closures	2,865.40	79.80	35.91
arraybuf.Ops	179.80	72.40	2.48
arraybuf.Reads	826.00	421.80	1.96
arraybuf.Burst	459.80	218.40	2.11

Figure 5.2: Steady-state execution time (ms) and speedup

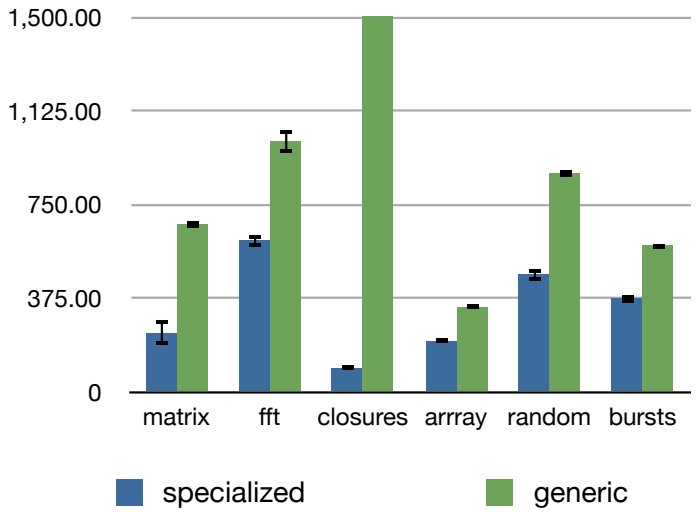


Figure 5.3: Startup performace of specialization

million operations). We also believe that the cache performance was greatly improved when there was no boxing, amplifying the benefits of direct integer manipulation.

Start-up time

Figures 5.3 and 5.4 show roughly the same tendency as for steady-state performance. The speedups are slightly lower than before, as specialized programs have more classes to load/verify.

Code size

Figure 5.5 shows the results of measuring code size in the Scala standard library. The first line shows the increase in size of the standard library when

	Generic	Specialized	Speedup (Nx)
matrix	671.83	236.67	2.84
fft	1,005.33	608.17	1.65
closures	2,679.00	95.83	27.95
arraybuf.Ops	340.17	204.00	1.67
arraybuf.Reads	874.50	469.50	1.86
arraybuf.Burst	585.67	374.50	1.56

Figure 5.4: Startup execution time (ms) and speedup

	Generic	Specialized	Increase
stdlib	10.7M	12.5M	17%
specific	142K	426K	300%
specific[pack200]	32.1K	51.8K	161%

Figure 5.5: Code size in the standard library

specializing array-backed collections, functions up to two parameters and lists. The cost is relatively small, 17%. However, when we look at the modified classes alone the picture is rather bleak: the specialized code is around three times larger. This is the exact size on disk, and we believe the effect of specialization is seriously aggravated by the class file format of the JVM. Most specialized classes are relatively small, and pay the price of a full constant pool and structure information. In the last line we show the results after using the pack200 jar file compressor, which is known to share constant pool information between classfiles; the relative cost went down to almost half of what it was before.

When we turn to the previous benchmarking suite, the code size follows the same pattern: specialized library code takes a hit, but application code using such libraries is roughly the same size. However, things are much worse for the array buffer class, whose code increases over 20 times. The reason is that `ArrayBuffer` is specialized for all 9 primitive types, and one closure defined inside `foldLeft` is specialized for all primitive types of the *method* type parameter, bringing in 81 specialized classes for the relatively small closure class.

The cost of extra classes is alleviated by using a jar compressor, but it is definitely a concern for library writers. This is not unexpected, and we believe that library developers are in a good position to make decisions about performance-critical areas and where specialization is truly needed. For instance, the cost can be brought down by requiring specialization only on a few primitive types (for instance, `Unit` is most likely useless as a specialization).

	Generic	Specialized	Increase
matrix	120	160	33.5%
fft	88	88	0%
closures	48	48	0%
arraybuf	64	1448	22x
arraybuf[pack200]	16	48	3x

Figure 5.6: Code size in the benchmark suite

5.4 Optimizations

In this section we look at the performance improvements we can get from compiler optimization alone. The benefits of optimization come from closure and boxing elimination. Since specialization is guaranteed to remove boxing on specialized code paths, we turn it off for the following benchmarks. We are interested to see if optimization can entirely eliminate anonymous functions, and if this truly improves program execution times. The JVMs have improved greatly in terms of the runtime optimizations they perform, and the space left for static optimizations has been consequently reduced (or so it is thought).

5.4.1 Benchmark suite

As described in Chapter 2, we are mainly interested in reducing the cost of language extensions through libraries. We have identified boxing and higher-order functions as prime candidates for optimization, and we tailor our suite to this purpose. General purpose optimizations, like strength reduction, loop unrolling, code motion, redundant expression elimination [36, 29, 44, 28] are not of prime interest to this work. The JVM is very good at doing it already, and most of these optimizations are intra-procedural, meaning they have little to no effect when using compiled libraries. We focus instead of several patterns of usage of language extensions, and show that the JVM does not match the static optimizations that we perform. This is in part due to the speed restrictions of the just-in-time compiler, and to the limited type information available for the JVM runtime.

The benchmark suite we use for optimizations is a slightly modified version of the original suite. We have added a new use-case that shows a language extension, Java-like asserts, and have removed the `ArrayBuffer` benchmarks. The matrix multiplication show-cases for-comprehension on integers.

fft, closures Fast-Fourier Transform and closure test. They are the same benchmarks as in § 5.3.1.

matrix Matrix multiplication. Essentially the same algorithm as in § 5.3.1, but instead of using a matrix class it manipulates arrays directly. It mainly measures the effectiveness of for-loop optimizations (see § 2.3.3).

```
def assert(cond: => Boolean, message: => Any) {
  if (assertionsEnabled && !cond)
    throw new AssertionError("assertion failed: " + message)
}
```

Figure 5.7: Predef.assert

```
def sqrt(x: Int): Int = {
  assert(x >= 0, "Expected positive integer: " + x)
}

def test(xml: Node) {
  assert(xml.label == "persons", "Wrong Xml root element: " + xml)
}
```

Figure 5.8: Using assert

asserts A language extension that implements Java-like asserts, in two scenarios: enabled and disabled. See discussion below.

The `assert` keyword from Java can be implemented in Scala as a library function. The tricky part consists in the requirement that the condition and message should not be evaluated unless assertions are enabled, and the assertion fails, respectively. In a strict language like Java, and in the absence of macros, this is impossible to express. Scala offers call-by-name parameters, which turn arguments at each call site into nullary functions. The body of the method applies the function each time it needs to obtain the value.

The matrix multiplication example is centered around a triple nested for loop:

```
def multiply(n: Int, a: Array[Array[Int]], b: Array[Array[Int]]) = {
  val c: Array[Array[Int]] = new Array(n, n)
  for (i <- 1 until n; j <- 1 until n; k <- 1 until n)
    c(i)(j) += a(i)(k) * b(k)(j)
  c
}
```

As explained in § 2.3.3, for comprehensions are expanded to nested calls to `foreach`, which in turns gets a closure argument. The purpose of this benchmark is to show that we can keep the library implementation of plain loops on integer values, and still get good performance.

Steady-state performance

Figure 5.9 shows the execution times for the benchmark suite. For asserts we measure both enabled and disabled asserts, to show that the evaluation of the

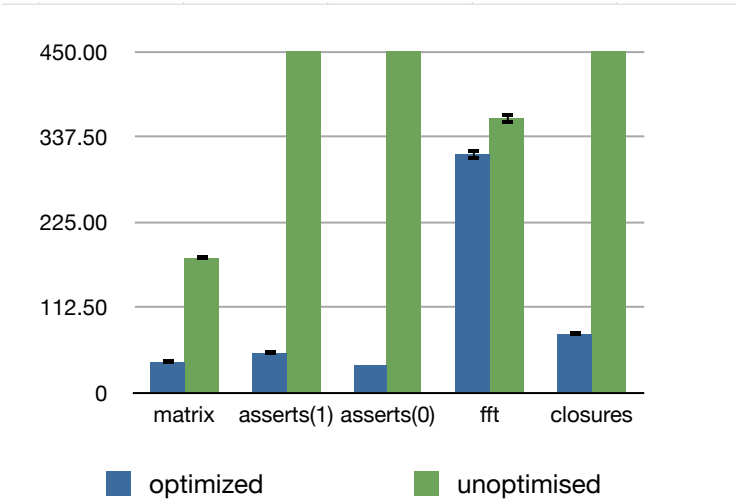


Figure 5.9: Optimized steady-state execution times

condition is optimized as well. Matrix multiplication is 4.3 times faster, and the two assert benchmarks are an order of magnitude faster. The execution time for the two assert benchmarks is dominated by closure application, and the fact that the compiler eliminated the overhead leads to a large improvement It is interesting to note that for matrix multiplication, static optimization gives better speedup than specialization alone, showing that optimizations can remove more than just the boxing cost, and that the JVM does not perform the same level of inlining and cross-method optimizations.

The `closures` benchmark shows the same 35x speedup, which is hard to beat. The amount of boxing that optimizations alone can remove seems to be in line with specialization. Comparatively, The Fast-Fourier transformation is not improved by much: it is only 15% faster. In this case the optimizer is not able to remove boxing and unboxing that goes on when using the generic `Pair` class, so specialization is outperforming the optimizer. Why does the optimizer fail to eliminate pairs? There are several reasons at play:

access modifiers In order to inline methods, like accessors to the tuple fields, their body has to use only public members (otherwise, code would not pass verification). Pairs keep their fields private.

redundancy Boxing can be eliminated when the boxed value exists in unboxed form and is guaranteed to have the same value. This usually happens with closures that capture an environment: the boxed value is passed to the closure, but the unboxed value still exists in the caller's environment. However, in this benchmark pairs are stored in an array, and their elements exist only as `Doubles`. The only way out is to specialize the holder

	Optimized	Unoptimized	Speedup (Nx)
matrix	40.80	177.20	4.34
asserts(1)	52.60	785.00	14.92
asserts(0)	36.00	768.60	21.35
fft	314.60	362.40	1.15
closures	76.60	2,666.40	34.81

Figure 5.10: Execution time (ms) and speedup for steady-state.

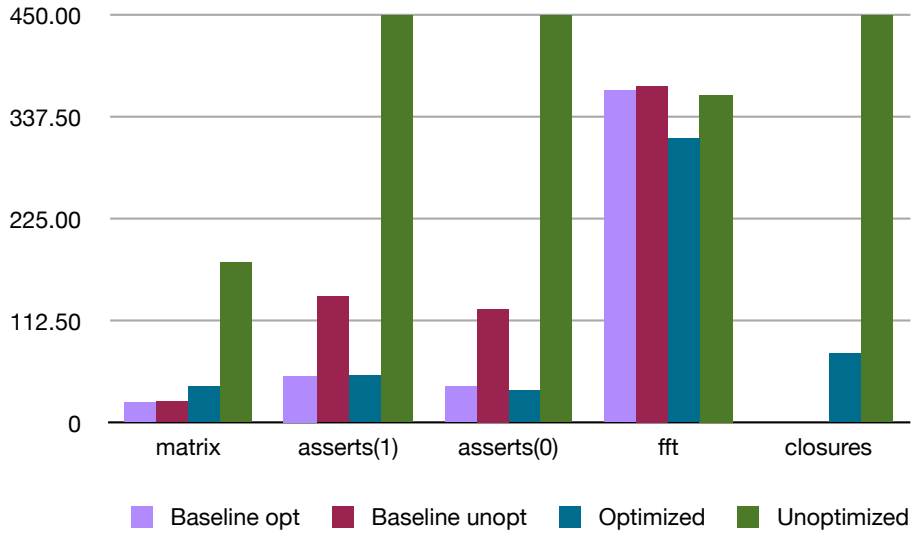


Figure 5.11: Execution times compared to a hand-written baseline. Loops are rewritten to **`while`**, asserts are inlined.

class to have `double` fields.

Still, we get a 15% speedup, which is non-negligible in performance computing. It again shows that the JVM is not able to inline/optimize across methods as well our optimizer.

Figure 5.11 shows how execution times compare to hand-written versions of the benchmarks, which we call *baseline*. We measure both optimized and unoptimized performance in the baseline and original setting. Matrix multiplication and `fft` have been rewritten to use **`while`** instead of for-loops, and the two assert benchmarks have been inlined by hand. The `closures` benchmark misses the hand-written versions as it made little sense to transform it. We notice that in the matrix case, the optimized version gets very close to the baseline performance, while for the two asserts, the unoptimized baseline performance is actually worse than in the optimized case. However, the optimized baseline

	Optimized	Unoptimized	Speedup (Nx)
matrix	117.17	262.17	2.24
asserts(1)	61.83	850.33	13.75
asserts(0)	41.67	822.17	19.73
fft	517.33	503.83	0.97
closures	88.83	2,666.50	30.02

Figure 5.12: Execution time (ms) and speedup for startup.

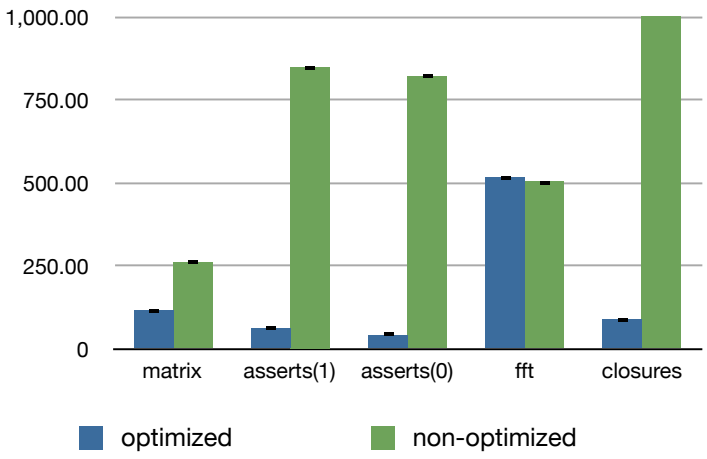


Figure 5.13: Optimized startup execution times

version performs as well as the original, optimized benchmark. For `fft` the baseline version does not perform better than the unoptimized version, probably because most of the work is done inside the loop.

Start-up time

Figures 5.13 and 5.12 show the results for startup times. We see a different picture here: only the assertion benchmarks have comparable speedups to the previous results. The worst case is for Fast-Fourier transform, which actually gets 3% slower. We blame this slowdown on the increased method size in the optimized program. Once the methods are jit-compiled (steady-state), the program is indeed faster, but larger methods increase the compilation time, and decrease the likelihood of the jit-compiler to inline them.

	Optimized	Unoptimized	Ratio
matrix	40	56	0.71
asserts(1)	24	72	0.33
asserts(0)	24	72	0.33
fft	80	82	0.97
closures	32	40	0.80

Figure 5.14: Code size for optimized programs

Code Size

The optimizer may affect code size in two different manners: increase the code size by inlining methods, and decrease by completely eliminating closure classes that have been inlined. Table 5.14 shows the code size before and after optimizations

Overall, the optimizer does a pretty good job at removing closures, and none of the examples show any increase in the code size. For the `matrix` benchmark, the optimizer removes 3 out of 5 closure classes. The ones that are not eliminated are due to hitting a limit on the overall inlining that is allowed per method, in order to keep method sizes and compilation times under control. More inlining can generally be forced by annotating methods with `@inline`, which skips this check. In the `assert` examples, all 7 closures are successfully inlined and their classes eliminated. In the `FFT` benchmark, only 3 out of 8 closures were successfully eliminated, which may also explain the relatively low speedup observed on this benchmark. On the last benchmark, only one closure out of two has been removed, but interestingly, it is the right one (the one in the tight loop). That explains the great speedup we observed.

5.4.2 The Scala compiler

In order to asses how the optimizer handles large programs, and if the performance is indeed improved for non-trivial programs, we benchmarked the Scala compiler itself. The compiler has 88,000 lines of Scala code, and the standard library adds another 62,000. There over 450 calls to the `assert` method (defined using call-by-name parameters for both the condition and message). In addition, it calls about the same number of times to the `log` method, which we modified to take a call-by-name parameter:

```
@inline final def log(msg: => AnyRef) {
  if (settings.log contains phase.name)
    inform("[log " + phase + "] " + msg)
}
```

Logging may be expensive, since the message may contain pretty-printed tree fragments, so it is important that the argument to `log` is not evaluated unless logging is enabled.

	Execution time		Speedup
	unoptimized	optimized	
steady-state	3,432.88	3,317.75	1.035
start-up	13,519.67	12,995.50	1.040

Closures		Code size	
unoptimized	optimized	unoptimized	optimized
4,445	3,579	9,377,945	8,688,690

Figure 5.15: Benchmark results for the Scala compiler

We measured the time it takes `scalac` to compile the Scala runtime (41 files, 1,344 lines of code). Results are presented in Figure 5.15. We notice a slight improvement of 3.5% for steady-state performance, and 4% for start-up. The results are consistent for larger inputs. The relative small improvement can be explained by the fact that the compiler is written in a very cautious style, avoiding constructs that might slow down the compiler or increase memory usage. Additionally, it uses very few primitive values, so boxing and unboxing is almost not present in a compiler run. Overall, we were pleasantly surprised that the optimized version of `scalac` is actually faster.

In terms of code size, the optimized compiler is indeed smaller by almost 8%, owing to almost 900 closure classes that were completely eliminated.

5.5 Conclusions

We have presented a set of micro-benchmarks that asses the effectiveness of compiler optimization and specialization. We have been interested in seeing how performant code can be achieved using high-level constructs, like higher-order functions and generic container classes, without giving up optimizations on library code and genericity.

We have showed that specialization achieves speedups between 2 and 3 on generic code operating at primitive types. In extreme cases, this can go as high as an order of magnitude faster. The speedups are a bit less pronounced for startup time, which is a combination of more classes to load and more methods to jit-compile. The optimizer proved to be less predictable, giving very high gains when it can remove boxing and closure creation (5 to 6 times speedups), but it may fail to give high speedups when the structure of the code becomes too involved (for instance, relying on private fields, or storing containers of boxed values in other containers).

Chapter 6

Related Work

> Every person I work with knows something better than me. My job is to listen long enough to find it and use it.
>
> Jack Nichols

6.1 Specialization

Polymorphic code is code that can operate on a variety of types, without depending on the actual type of the values it manipulates. Parametric polymorphism (*generics* in object-oriented circles) allows code to be type checked using type variables instead of concrete types, and to instantiate type variables with actual types when the code is used.

There are two ways in which polymorphic code can be compiled (using the terminology from [11]): *homogeneous* translations use a single representation for polymorphic data, and *heterogenous* translations generate specialized version of the code for each instantiation. Homogeneous translations are compact and allow separate compilation, but they penalize performance when the concrete types are known. They rely on using one common representation for all values (usually one machine-word), and representing integers or double precision floating point numbers as pointers to *boxed* objects on the heap. Heterogeneous translations suffer from *code bloat* [35] and lack of true separate compilation, but offer good performance.

6.1.1 Homogeneous translations

Compilation of parametric polymorphism has been studied initially in the context of functional languages [30, 35, 26, 23]. All these approaches start from a

homogeneous scheme and look for opportunities to use unboxed representations when the concrete types allow it. Leroy proposes that values use their natural representation whenever the types are concrete, and add coercion operations (which he calls *wrap* and *unwrap*) at the points where concrete types enter/exit generic code [30]. His approach is similar to how Java handles generics [11]. Peyton-Jones and Launchbury take a complementary approach, starting with a language where all values use a uniform representation and showing a number of source-to-source translations that make boxing explicit [26]. Algebraic laws allow to further simplify and fuse operations on primitive types, leading to more efficient code. Similar to Leroy, Morrison et al. propose to use the natural representation for values whenever the types are concrete, and a uniform representation for polymorphic code [35]. Each polymorphic function is wrapped by an *envelope* function which converts the arguments from a concrete representation to the boxed form. Each value is tagged, and the envelope function decides at runtime how to convert the value to a uniform representation.

All of these approaches adhere to a single version of the compiled code. They are motivated by performance, but they only tackle the case when the code is truly monomorphic, resorting to boxing whenever a definition is polymorphic. This limits their use to fully concrete programs, penalizing the use of a generic container instantiated at a primitive type.

In the context of object-oriented languages there is less of a consensus on homogeneous translations. While Java and Scala use an erasure-based translation [40, 42], Ada, C++ and to some extent C# use heterogeneous translations.

There have been several proposals to extend Java with generics [11, 12, 2, 9]. The design finally chosen is based on [11], and uses a homogeneous translation. Classes and methods may have bounded type parameters. All types are checked during compilation, but they are not available at runtime. Casts (that are guaranteed to succeed) are added around the entry and exit points to/from generic code (casts are required by the JVM, which uses typed bytecode but is generics-agnostic).

6.1.2 Heterogeneous translations

C++ is one of the first object-oriented languages to have generics and brought them to the mainstream [10] (Ada had generics before C++, but the object-oriented extensions were added later [51]). C++ templates allow definitions (classes, functions, methods) to be parameterized by types and values. A template is instantiated by providing concrete types (and values) for its formal parameters. A new definition is derived by substituting the arguments for the template parameters, and the newly formed definition is type-checked at the point of application. Each distinct instantiation results in a new definition, and there is no uniform representation for values. The programmer may provide a specialized implementation for specific template instantiations. Together with the macro-like semantics of template instantiation, it leads to a Turing-complete language at the type level [56] and a technique based on tem-

plate meta-programming [4]. Despite their popularity in the C++ community and the good performance they entail, templates have been criticized for code bloat, lack of separate compilation, and obscure error messages.

Agesen et al.

One of the first proposals for Java generics is described by Agesen, Freund and Mitchell [2]. They propose a heterogeneous translation in which generic classes are compiled to an extension of Java bytecode, and specialized at load time. The modular architecture of the JVM allows users to write a *class loader*, which may perform load-time code modification. In this proposal, a special class loader pre-processes the extended bytecode and replaces all occurrences of type variables with their actual types, resulting in a specialized class for each generic class instantiation. This gives good performance, exact runtime types and a small code footprint, though this is misleading: new classes are generated on the fly, as the program instantiates generic code. Compared to other approaches, this allows type variables in more positions in the source code, for instance classes may extend type variables. However, it does not discuss how to handle method type parameters, since they would require an unlimited number of methods per class, and the JVM requires that classes are immutable once loaded.

Odersky et al. describe and evaluate their implementation of a heterogeneous translation based on Agesen et al for the Pizza compiler [41]. Polymorphic methods are translated using a homogeneous scheme, while parameterized classes are specialized at load time. They compare the heterogeneous translation to one based on type erasure and find that, contrary to their expectations, the performance of the specialized code was on average 26% slower on the pizza compiler itself. The slowdown came mostly from class loading (many more classes to load) and the specialization translation performed at load time.

Myers et al.

In [9], Myers, Bank and Liskov propose an extension of Java with generics at the VM level. Classes (but not methods) may be parameterized with types, which may be characterized by *where* clauses. A *where* clause lists required methods and constructors on a type parameter, together with their signatures. When a class is instantiated, the actual types have to conform to the *where* clauses in the generic definition. The translation is mostly homogeneous, but primitive type instantiation may require class specialization. The VM is extended with new instructions for invoking methods on type parameters, and new data structures for instantiation-specific class data, such as method pointers for *where* methods. Homogeneity holds as long as all types have the same size, but breaks for longs and doubles. In that case the authors propose class specialization, but it is unclear when that would be performed.

NextGen

Perhaps the closest to our approach in realization, if not in intention, is the approach proposed by Cartwright and Steele [12]. NextGen adds genericity to Java with full run-time type information, without changes to the JVM. They use a mixed translation strategy, sharing between all instantiations code that does not depend on types variables, and turning to a heterogeneous translation for the parts that do.

Each generic class is translated to an erased base class, plus one interface and one *wrapper* class per instantiation. The interface is simply a marker of the instantiated type, while the wrapper class extends the base class and implements the interface. Type tests are carried in terms of the interface, which is name mangled to represent the type instantiation. The wrapper class carries type-dependent operations grouped in *snippets*: whenever the base class uses type variables in code (such as creating an instance of a type parameter), the operation is extracted in an abstract method called a *snippet*. The wrapper class implements these snippets, as it is generated when the instantiation is known.

Unlike the other approaches so far, NextGen supports covariant type parameters. In order to reconcile the subtype relationship in the language with the one at the JVM level, each wrapper interface extends the wrapper interfaces of all supertypes instantiated at the immediate supertype of the argument. This can lead to many classes being generated, especially when using more than one type parameter. Unfortunately, this scheme breaks down for contravariant parameters, as the JVM does not allow the introduction of new supertypes.

Our approach is similarly using a scheme based on type-erasure, and adds specialized variants for instantiations. In our approach we aim to improve performance, and specialize only a bounded number of types (primitive types), while interestingly, NextGen does not allow primitive type instantiations. This keeps code bloat under control, but does not give precise run-time type information. Our specialization is optional and opportunistic, meaning the correctness of the translation does not depend on specializing all instantiations. We support both covariant and contravariant type parameters, which is a result of the limitation to primitive types (which are final and have only one super type, `AnyRef`). We do not need explicit snippets or wrapper interfaces because all type-dependent code is re-implemented in the specialized subclasses, which play the role of wrapper classes.

.NET Generics

Kennedy and Syme present an extension of the .NET Common Language Runtime that supports generics [27]. Similar to Myers et al., they extend the VM and the instruction set to accommodate polymorphic classes and methods.

The system preserves exact run-time types and uses both specialization and code sharing. Specialized classes are created lazily, as generic classes are instantiated. Whenever the code layout permits, an existing instantiation is reused. In practice this means that primitive types and user-defined value

classes are specialized, while reference types share the same code. The implementation does not rely on boxing, leading to good performance and efficient space usage.

To provide run-time types the implementation uses *dictionaries*, similar in intention to Cartwright's snippets. Dictionaries are lazily computed and contain type handles for all open-type expressions used inside (code-shared) generic code. Class dictionaries are co-located with *vtables*, but polymorphic methods may need an additional parameter to carry the dictionary corresponding to its own type parameters.

Our approach is similar in the way we mix homogeneous and heterogeneous translations. Primitive types trigger specialization, in the quest to save the cost of boxing, while reference types go through the erased representation. We differ in that we specialize opportunistically, and specialized and generic instantiations are interoperable: an instance of `List$Int` is also an instance of `List[Int]` and `List[Any]`.

Furthermore, it is not clear how covariance can be integrated without code sharing in this approach: suppose `List[+A]` is covariant in type `A`, and `Int` is a subtype of the top type, `Any`. It follows that `List[Int]` is a subtype of `List[Any]`. Unless the same representation is used for values of type `A`, the layout of the two classes in incompatible. We circumvent this problem by subclassing the erased version of `List`, and paying an extra cost in the additional fields. Kennedy et al. solve this problem by restricting covariance to interface and delegate types [1, 33].

Together with Yu, Kennedy and Syme have formalized and proven sound the specialization transformation [60]. We use a smilar language and technique to describe our transformation in Chapter 4.

6.2 Optimizations

Compiler optimizations have been extensively studied for the past 40 years. In this work we focus on optimizations for object-oriented languages for the Java Virtual Machine. More precisely, we aim to bring down the cost of abstraction in Scala, mostly through optimizing higher-order functions and eliminating the cost of boxing whenever possible.

Based on the moment when optimizations are performed, we can roughly divide them in *dynamic* and *static* optimizations. Dynamic optimizations are performed at run time, and they are usually directed at the points that get executed the most (hot-spots). Static optimizations are performed by the compiler and are done ahead-of-time. Based on the scope of optimizations, we further divide them in *whole program* and *modular*. The former assume the whole program is available for analysis, and it is usually implemented by virtual machine optimizers, while the latter has a limited view of the program and consequently works with less precise information.

6.2.1 Virtual method call resolution

The first step in optimizing a program is to build a call-graph, linking call sites to the methods that might be executed. Object-oriented programs make extensive use of dynamic dispatch, meaning the method that is called depends on the dynamic type of the receiver object. In other words, on the value that flows to the receiver. All of the following approaches assume the whole program is available for analysis.

Control Flow Analysis

Even though initially described as an analysis for dynamically typed functional languages (Scheme), Shivers is the first to describe a principled solution to resolving dynamic function calls [48, 47]. In Scheme, functions are values which can be passed around, bound to variables or stored in data structures. The code that is executed when a function is applied depends on the values that flow into the call site.

Shivers describes a framework to compute the set of possible values that may flow at a program location, and that may be bound to a variable. The analysis is called CFA, and in its simplest form it identifies variables and functions with their syntactic expression (0-CFA). This form does not distinguish between different instances of a function call, effectively merging all closures that may be created at a program point (and the same for variables). In other words, the analysis is not contextual. The idea is to build a set of constraints on the possible functions that may reach a variable or program point. For example, a function definition adds itself to the set that could flow to that point, and a function application adds the constraint that all possible functions flowing in an argument have to be included in the set of possible functions of its corresponding function parameter. A solution to the constraint system gives the required information about call sites.

The analysis may be extended to keep track of the context of each function application. The information computed at program points is qualified with a context, essentially a stack of program points representing the call-path to that point. The context depth is bound by a constant, hence the name k-CFA.

Class Hierarchy Analysis

In the context of the Vortex compiler, Dean, Grove and Chambers [14, 22] developed a series of techniques for optimizing object-oriented programs. Virtual calls are resolved using Class Hierarchy Analysis (CHA), which is a very natural and simple technique using the static type of the receiver object: build a whole program hierarchy of classes, and at each call site retain only the methods that are actually implemented in a subclass of the static type of the receiver. They report good results for the Cecil language, results that were later confirmed by Sundaresan et el. for Java [50].

Rapid Type Analysis

A simple and effective refinement of CHA is rapid type analysis [7]. The observation is that many classes are never instantiated, for instance when the program uses only a small part of an existing library. RTA builds a set of all instantiated classes, and prunes the call graph by removing methods defined in classes that are not in this set. Bacon and Sweeney report that 71% of the virtual calls in C++ can be resolved by RTA, and only 51% by CHA alone. The results seem to carry over to Java, as reported by Sundaresan in [50], where he suggests that the biggest improvement comes from unused library code (the percentage of resolved calls drops from 77% to 7% when they consider the benchmark code alone).

Variable Type Analysis

In the larger context of the Soot project [55, 54], Sundaresan et al. describe a more precise algorithm for resolving method calls in Java [50]. The algorithm starts from a conservative call graph built using CHA, on which it builds a propagation graph. The idea is to push types from allocation (calls to `new`) to their use, through fields, method arguments and return values: edges connect the two sides of an assignment, and actual to formal method arguments. They report improvements over RTA, especially when considering only the call-sites appearing in the benchmark code (thus excluding library code).

Even if we did not mind the whole program assumption, both CHA and RTA are not precise enough for successfully eliminating closure calls in Scala. As explained in § 2.6.1, closures are called through an interface that has hundreds of implementations in a typical Scala program, and it is practically impossible that only one concrete implementation is ever instantiated (as required by RTA). On the other hand, VTA can resolve a very limited class of closure calls: when the call site is truly monomorphic, even though the static type of the receiver is still the `FunctionN` interface, it may be able to propagate the instantiated closure class to the application point.

6.2.2 Java static optimizations

Optimization efforts for Java have been directed either at runtime optimizations at the virtual machine level, or static optimizations at the bytecode level. Bytecode optimizers translate compiled bytecode to an intermediate representation, perform various optimizations and then rewrite the program either back to bytecode or to native code. Both approaches assume the whole program is available for analysis.

Soot

Soot is a framework for optimizing Java bytecode [55]. It offers three intermediate representations, in increasing order of abstraction: Baf, Jimple and

Grimp. Baf is a simple, straight line, stack based representation of bytecode. It abstracts over the constant pool and provides a typed instruction set. Jimple is a three-address code intermediate representation more suitable for intra-procedure optimizations. The stack is replaced by additional temporary variables, and the `jsr` instruction is removed by inlining its target. Lastly, Grimp is a high-level intermediate language, closer to decompiled Java. For instance, it uses arbitrary expressions instead of the flat sequence of operations dictated by three-address codes. A fourth representation, Shimple, has been added later [53] and it provides a static single-assignment form IL.

Different optimizations have been implemented using Soot. In [54] Vallée-Rai et al. describe and evaluate a number of optimizations for Java. They have implemented inlining and a number of traditional intra-procedural optimizations, like copy propagation, constant folding, conditional and unconditional branch folding, dead-code elimination and dead assignment elimination. They report speedups of 4 to 8%. The inlining strategy is based on CHA and starts with leaf methods, inlining as much as possible, without increasing the size of the caller above a threshold, and only if the callee is below a fixed size. Being focused on Java, they do not attempt to eliminate object allocation, as Java did not have closure objects (and still doesn't) nor boxing at the time. We believe our approach is a natural extension of this work.

Transforming bytecode into a typed representation poses some challenges. Local variables and stack location are untyped at the VM level, and type recovery is not always possible without splitting variables. Gagnon et al. describe several extensions to Plevyak's work on type inference in object-oriented programs [16, 45]. We have found that the problematic cases are very rare in practice.

Marmot

Marmot is an optimizing Java compiler developed by Fitzgerald et al. at Microsoft Research [15]. Marmot is a bytecode to native code compiler, so it necessarily is whole program. The goal of this project was to produce code as efficient as C++, keeping the original Java semantics as much as possible. For instance, Marmot does not support dynamic class loading, and only supports a subset of reflection.

The intermediate language is a three-address code that uses additional temporary variables instead of the stack, called JIR. Basic blocks differ from the classical definition [3] in the same way as in our work: besides the normal control-flow exit points at the end of the block, exception handlers add special edges that may originate at any instruction in the block. Handlers are paired with the type of the exception they may catch and may cover several basic blocks. JIR is in static single-assignment form.

Marmot performs an impressive list of optimizations, ranging from standard (common sub-expression elimination, constant/copy propagation, loop invariant code motion, induction variable elimination, etc) to object-oriented and Java specific (synchronization elimination, stack allocation of objects, null

check removal, etc). Calls are resolved using a lazy variation of CHA. Perhaps surprisingly, inlining is guided by a very simple strategy, and performed only when the inlined method gives shorter code than the original call sequence.

Their results show speeds comparable to other Java systems, and within 3 to 38% slower than the C++ version of the program.

6.2.3 Dynamic optimizations

The picture of compiler optimizations for the Java platform would be incomplete without mentioning the intense research area of runtime optimizations. Indeed, most research efforts in compiler optimization are focused on just-in-time compilation. Providing a complete picture of this area is beyond the scope of this work, but we highlight some of the features of the leading JVM, Sun's[1] HotSpot virtual machine.

Dynamic optimizations are attractive because they provide several important pieces of information to the compiler: *feedback* – what regions of the code are executed the most, hence where optimization effort should be directed; a *complete* view on the program – whole program techniques can be applied. In addition, it frees compiler writers from restrictions in the bytecode abstraction level. Some optimizations can only be performed when the target language is sufficiently close to the processor, like array bounds check elimination, register allocation, or instruction scheduling.

The downside is that optimizations have to be very fast, since the compilation cost is paid at runtime. This usually forbids algorithms that need many passes over the code, and leads to a tradeoff between optimized code and fast compilation. To ease the decision, two compilers have been implemented in the HotSpot VM, the *client* compiler which is very fast, but implements only a few optimizations, and the *server* compiler, which spends more time optimizing but whose running time has to be amortized by long running processes. The user is called on to decide which one to run by passing a command-line option.

The HotSpot server compiler

The recommended configuration for long running applications is the HotSpot server compiler [44]. The compiler described by Paleczny et al. uses an SSA-based representation that blurs the distinction between basic blocks and instructions [13]. Each instruction is a node, and it represents the value that it computes. Similarly, the operands of an instruction are pointers to their definition node, meaning that use-def chains are part of the representation.

The server compiler performs constant propagation, inlining, global value numbering, graph-coloring register allocation and instruction scheduling. Interestingly, inlining is performed at parse time, when compiling a method.

[1] Sun has been recently acquired by Oracle, but we chose to attribute this technological achievement to the original company.

When the target of the call can be determined (using CHA, or through receiver profiling), the target method bytecode is merged into the caller and optimized together. If some of the assumptions on which inlining has been performed are invalidated later, the method is de-optimized and relegated to interpreted mode.

Most of the compiler time is spent in the register allocator (49%), followed by the optimizer (20%) and the parser (14%). The server compiler applies many of the traditional compiler optimizations, ensuring that Java compilers can remain relatively simple and let the runtime deal with the platform specifics.

The HotSpot client compiler

Applications are not always long-running, performance critical programs. Desktop applications favor fast response times and usually run for shorter time, augmenting the relative cost of expensive optimizations. The client compiler has been developed as an alternative to the performant (but somewhat slow to start up) server compiler. The two compilers do not share any code, and the intermediate representation is different [21]. The code generator is very simple, and initially the register allocator was unnecessarily pessimistic.

More recently, Kotzmann et al. describe advances in the client compiler for Java 1.6 [29]. The intermediate representation is now SSA-based sea-of-nodes [13], the register allocator uses linear scan, and various optimizations have been implemented, like inlining, global and local value numbering, null-check elimination and conditional expression elimination.

In the same paper Kotzmann et al. describe a number of optimizations that may be included in future versions. Of prime interest to our work is *scalar replacement*: when an object is proven not to *escape* its defining scope, its fields can be replaced by local variables, and the object construction completely elided. In the context of closure elimination, this means that the closure object is created and consumed in the same method, depending essentially on the ability to inline both the higher-order function (like `foreach`), and the function application inside. Differently to their work, we never add local variables to act as fields of the eliminated closure; instead we rely on proving that there is a mapping between each field and a local variable prior to object elimination.

Escape analysis opens the way to a number of other interesting transformations, like stack allocation and thread synchronization removal. Stack allocation can be employed when the object escapes only to methods called from the current method. The object is then allocated on the thread stack, and a pointer is passed to callees. When the code leaves the method, the object is deallocated automatically.

Other JVMs

Besides the reference implementation, a number of JVMs have been developed by others. Most of them rely on a similar approach to adaptive optimization.

The IBM implementation relies on three levels of optimizations, and a controller that decides when to recompile a method using a more aggressive level [49]. BEA JRockit is a production-level JVM that uses ahead-of-time compilation. Every method is compiled before it is executed, and based on sampling profiles hot methods are recompiled using an optimizing compiler. Jikes Research Virtual Machine is a JVM developed by IBM and written in Java [5], using adaptive optimization, and it is the basis of a large number of research ideas (over 188 papers on the project's website).

Chapter 7

Conclusions and Future Work

> Finally, in conclusion, let me say just this.
>
> Peter Sellers

We have identified and described several areas of impact for efficient Scala programs: higher-order functions and closures, and generic definitions used at primitive types. We further identified *boxing elimination* as essential for tackling both problems.

This thesis presented two solutions for improving the performance of compiled Scala code: optimization and specialization. We have implemented both proposals in the Scala compiler, and are part of the current release of Scala (version 2.8.0)[1]. To evaluate our approach, we used a set of benchmarks and have shown that they are viable.

7.1 Optimizations

Closures and higher-order functions can perform better when they are optimized together. We proposed closure elimination through aggressive inlining and an extended algorithm for copy-propagation.

We have implemented a series of static optimizations that operate across compilation units and shown how inlining followed by closure elimination and dead-code elimination can reduce the cost of closures by up to 5 times. Inlining uses a flow-sensitive analysis to find more precise types at call sites, and for closure elimination we extend copy-propagation with a simple model for objects that can flatten closures and remove boxing.

Separate compilation can be achieved without giving up on the optimization of library code. We have implemented a *bytecode reader* that can bring

[1] Some improvements have been made after the 2.8.0 release. They are available in the nightly distribution, and will be part of the next Scala release.

library code back to the same representation used by the Scala compiler for its own optimizations.

We can improve upon the proposed optimizations by having more precise analyses. Side effect analysis is of prime interest, and could improve the effectiveness of closure elimination. Escape analysis could allow more interesting optimizations, like scalar replacement, and both are natural continuations of this work.

7.2 Specialization

We proposed a new approach to compiling parametric polymorphism for performance. We mix a homogeneous translation scheme with user-directed specialization for primitive types. Specialized classes are compatible with unspecialized code, and specialization agnostic code can work with specialized instances. We achieve separate compilation by limiting specialization to primitive types.

We have implemented and evaluated the approach on several Scala benchmarks. Generic collections and closures benefit the most from this scheme, and we showed that execution speed can increase more than two times with our technique.

So far specialization is an implementation technique, keeping the current semantics of Scala programs. Specialized classes are derived automatically by the Scala compiler, for different combinations of specialized type parameters. We can extend the current mechanism by allowing users to specify a specialization for a certain combination of type parameters, similarly to C++ specialized templates. This would allow a programmer to give a bit set implementation of generic sets, for instance

```
class Set[@specialized T] {
  def apply(x: T): Boolean
}
```

and specialize for `Int`:

```
@specialized class Set[Int] {
  def apply(x: Int) = //...
}
```

We have shown two complementary approaches for improving performance of Scala programs. Specialization is more predictable and well-suited for library designers who can balance between code size and performance. Code size can be a problem, especially when the number of specialized type parameters goes beyond two. On the other hand, optimizations deal better with code size, but may miss opportunities. They are in general less predictable, but may give very good results when they apply.

The abundance of optimizing JVMs leads to the conclusion that Java compilers may safely omit traditional optimization techniques like value number-

ing, constant propagation and folding, or code motion. However, there are areas where JVMs today do not perform as well as static compilers, mostly because of the restrictions on execution times and lack of type information. We have shown that significant improvements can be achieved by static compilation techniques, even when targeting highly-optimizing VMs.

Appendix A

Proofs

Lemma 1 (Type Substitution)**.** *Given a specialization s not defined at any type variable in $\overline{X}$, $|[\overline{T}/\overline{X}]U|_s = [|\overline{T}|_s/\overline{X}]|U|_s$.*

Proof. We prove by induction on the structure of types.

- $U = Y$ (type variable)
  - $Y \in \overline{X}$.

$$|[\overline{T}/\overline{X}]Y|_s = |T_i|_s$$
$$[|\overline{T}|_s/\overline{X}]|Y|_s = [|\overline{T}|_s/\overline{X}]Y = |T_i|_s$$

  - $Y \notin \overline{X}$.

$$|[\overline{T}/\overline{X}]Y|s = |Y|_s = s(Y)$$
$$[|\overline{T}|_s/\overline{X}]|Y|_s = |Y|_s = s(Y)$$

- $U = P$ (primitive type)

$$|[\overline{T}/\overline{X}]U|_s = |[\overline{T}/\overline{X}]P|_s \qquad = |P|_s = P$$
$$[|\overline{T}|_s/\overline{X}]|U|_s = [|\overline{T}|_s/\overline{X}]|P|_s \qquad = |P|_s = P$$

- $U = C[\overline{V}]$ (instantiated type)

$$|[\overline{T}/\overline{X}]U|_s = |[\overline{T}/\overline{X}]C[\overline{V}]|_s \qquad = |C[[\overline{T}/\overline{X}]\overline{V}]|_s = C[|[\overline{T}/\overline{X}]\overline{V}|_s]$$
$$[|\overline{T}|_s/\overline{X}]|U|_s = [|\overline{T}|_s/\overline{X}]|C[\overline{V}]|_s \quad = [|\overline{T}|_s/\overline{X}]C[|\overline{V}|s] = C[[|\overline{T}|_s/\overline{X}]|\overline{V}|_s]$$

  and by the Induction Hypothesis,

$$|[\overline{T}/\overline{X}]\overline{V}|_s = [|\overline{T}|_s/\overline{X}]|\overline{V}|_s$$

- $U = \forall\overline{Y}.\overline{V} \to V$

  We assume $\overline{Y}$ is distinct from $\overline{X}$ and $\mathrm{dom}(s)$ without loss of generality.

$$\begin{aligned}|[\overline{T}/\overline{X}]\forall\overline{Y}.\overline{V} \to V|_s &= |\forall\overline{Y}.([\overline{T}/\overline{X}]\overline{V} \to [\overline{T}/\overline{X}]V|_s \\ &= \forall\overline{Y}(|[\overline{T}/\overline{X}]\overline{V}|_s \to |[\overline{T}/\overline{X}]V|_s)\end{aligned}$$

$$\begin{aligned}[|\overline{T}|_s/\overline{X}]|\forall\overline{Y}.\overline{V} \to V|_s &= [|\overline{T}|_s/\overline{X}]\forall\overline{Y}.|\overline{V}|_s \to |V|_s \\ &= \forall\overline{Y}.\left([|\overline{T}|_s/\overline{X}]|\overline{V}|_s \to [|\overline{T}|_s/\overline{X}]|V|_s\right)\end{aligned}$$

  By Induction Hypothesis

$$\forall\overline{Y}(|[\overline{T}/\overline{X}]\overline{V}|_s \to |[\overline{T}/\overline{X}]V|_s) = \forall\overline{Y}.\left([|\overline{T}|_s/\overline{X}]|\overline{V}|_s \to [|\overline{T}|_s/\overline{X}]|V|_s\right)$$

  and immediately follows that

$$|[\overline{T}/\overline{X}]\forall\overline{Y}.\overline{V} \to V|_s = [|\overline{T}|_s/\overline{X}]|\forall\overline{Y}.\overline{V} \to V|_s \qquad \square$$

Lemma 2 (Field Specialization)**.** *Given a specialization s, a well-formed type $C[\overline{T}]$, and $fields(C[\overline{T}]) = \overline{f : U}$, we have that*

$$fields\left(|C[\overline{T}]|_s\right) = \overline{f : |U|_s}.$$

Proof. By induction on the inheritance relationship.

- $C = Object$. Trivial, since *Object* has no fields
- $C[\overline{X}]$ extends I $\{\overline{f_1 : U_1}; \overline{md}\} \in \mathcal{D}, fields([\overline{T}/\overline{X}]I) = \overline{f_2 : U_2}$

  We have that $fields(C[\overline{T}]) = \overline{f_1 : [\overline{T}/\overline{X}]U_1}, \overline{f_2 : U_2}$

$$\begin{aligned}fields(|C[\overline{T}]|_s) &= fields(C[|\overline{T}|_s]) && \text{by definition of } s \\ &= \overline{f_1 : [|\overline{T}|_s/\overline{X}]U_1}, fields([|\overline{T}|_s/\overline{X}]I) && \text{by definition of } fields\end{aligned}$$

  All type variables mentioned in the declared field types U_1 are drawn from $\overline{X}$ (because of T-CLASS, the environment Γ used for typing fields contains only $\overline{X}$). The type $[\overline{T}/\overline{X}]U_1$, contains no type variables in $\overline{X}$, and the only place where s can 'fire' is inside the types $\overline{T}$. It follows that

$$|[\overline{T}/\overline{X}]U_1|_s = [|\overline{T}|_s/\overline{X}]U_1 \tag{A.1}$$

  by the same line of reasoning,

$$|[\overline{T}/\overline{X}]I|_s = [|\overline{T}|_s/\overline{X}]I$$

therefore

$$fields([|\overline{T}|_s/\overline{X}]I) = fields(|[\overline{T}/\overline{X}]I|_s) \tag{A.2}$$
$$= \overline{f_2 : |U_2|_s} \qquad \text{by the Induction Hypothesis} \tag{A.3}$$

From A.1 and A.2 follows that

$$fields(|C[\overline{T}]|_s) = \overline{|[\overline{T}/\overline{X}]U_1|_s}, \overline{f_2 : |U_2|_s} \qquad \square$$

Lemma 3 (Term Specialization)**.** *Given a valid class table D, a specialization s and a term e such that* $\Gamma \vdash e : T$*, we have*

$$|\Gamma|_s \vdash [\![e]\!]_s : |T|_s$$

Proof. We prove by induction on the structure of terms.

- $e = n$

  $$\Gamma \vdash n : Int$$

  Trivial.

- $e = x$

  We have

  $$\Gamma \vdash x : T.$$

  From T-VAR we have that $x : T \in \Gamma$. By definition of $|\Gamma|_s$ we have that $x : |T|_s \in |\Gamma|_s$, therefore

  $$|\Gamma|_s \vdash x : |T|_s$$

- $e = e.f$

  We have

  $$\Gamma \vdash e.f : V_i$$

  We prove

  $$|\Gamma|_s \vdash [\![e]\!].f : |V_i|_s$$

  From T-FIELD we have

  $$\Gamma \vdash e : U \qquad fields(U) = \overline{f : V}$$

  By Induction Hypothesis we have

  $$|\Gamma|_s \vdash [\![e]\!] : |U|_s$$

  and by Lemma 2

  $$fields(|U|_s) = \overline{f : |V|_s}$$

  It follows directly that

  $$|\Gamma|_s \vdash [\![e]\!].f : |V_i|_s$$

- $e = e.m[\overline{U}](\bar{e})$

  We have
  $$\Gamma \vdash e.m[\overline{U}](\bar{e}) : T$$
  We prove
  $$|\Gamma|_s \vdash [\![e]\!]_s .m[|\overline{U}|_s]([\![\bar{e}]\!]_s) : |T|_s$$
  From T-INVOKE we have
  $$\Gamma \vdash e : I \qquad mtype(m, I) = \forall \overline{Y}.\overline{T} \to T \qquad \Gamma \vdash \bar{e} : [\overline{U}/\overline{Y}]\overline{T}$$
  By Induction Hypothesis we have
  $$|\Gamma|_s \vdash [\![e]\!]_s : |I|_s \qquad |\Gamma|_s \vdash [\![\bar{e}]\!]_s : \left|[\overline{U}/\overline{Y}]\overline{T}\right|_s$$
  s is not defined at any of the method type parameters $\overline{Y}$, so we can use Lemma 1 and get
  $$|\Gamma|_s \vdash [\![\bar{e}]\!]_s : [|\overline{U}|_s/\overline{Y}]|\overline{T}|_s$$
  To prove our goal we need to apply the same typing rule, T-INVOKE. The only missing ingredient is the method type, so we will prove that
  $$mtype(m, |I|_s) = \forall \overline{Y}.|T|_s \to |T|_s$$
  Let $I = C[\overline{V}]$. Using the definition of *mtype* we have
  $$\begin{aligned} mtype(m, |C[\overline{V}]|_s) &= mtype(m, C[|\overline{V}|_s]) \\ &= \forall \overline{Y}.[|\overline{V}|_s/\overline{X}]\overline{U_1} \to [|\overline{V}|_s/\overline{X}]U_1 \end{aligned}$$
  where $\overline{X}$ are the type parameters of class C and $\overline{U_1}, U_1$ are the declared types of method m in class C ($[\overline{V}/\overline{X}]\overline{U_1} = \overline{T}$ and $[\overline{V}/\overline{X}]U_1 = T$).

  Similarly to the argument in Lemma 2, all free type variables in $\overline{U_1}, U_1$ are from $\overline{X}, \overline{Y}$. Furthermore, s is not defined at any type variable in $\overline{Y}$, therefore
  $$[|\overline{V}|_s/\overline{X}]\overline{U_1} = \left|[\overline{V}/\overline{X}]\overline{U_1}\right|_s = |\overline{T}|_s$$
  We have
  $$mtype(m, |C[\overline{V}]|_s) = \forall \overline{Y}.|\overline{T}|_s \to |T|_s$$
  We can now use T-INVOKE to conclude that
  $$|\Gamma|_s \vdash [\![e]\!]_s .m[|\overline{U}|_s]([\![\bar{e}]\!]_s) : |T|_s$$

- $e = \text{new } I(\bar{e})$

  We have
  $$\Gamma \vdash \text{new } I(\bar{e}) : I$$
  We prove
  $$|\Gamma|_s \vdash \text{new } |I|_s([\![e]\!]_s) : |I|_s$$

From T-NEW we have

$$fields(I) = \overline{f : T} \qquad \Gamma \vdash \overline{e : T}$$

By Induction Hypothesis we have

$$|\Gamma|_s \vdash [\![e]\!]_e : |T|_s$$

and by Lemma 2

$$fields(|I|_s) = \overline{f : |T|_s}$$

Using T-NEW we conclude that

$$|\Gamma|_s \vdash \text{new } |I|_s([\![e]\!]_s) : |I|_s$$

- $e = e.\text{as}[T]$

  We have

  $$\Gamma \vdash e.\text{as}[T] : T$$

  We prove

  $$|\Gamma|_s \vdash [\![e]\!]_s .\text{as}[|T|_s] : |T|_s$$

  Using T-AS we have

  $$\Gamma \vdash T \; ok. \qquad \Gamma \vdash e : T'$$

  then

  $|\Gamma|_s \vdash [\![e]\!]_s : |T'|_s$ by Induction Hypothesis
  $|\Gamma|_s \vdash |T|_s \; ok.$ by Lemma 4

  It follows immediately that

  $$|\Gamma|_s \vdash [\![e]\!]_s .\text{as}[|T|_s] : |T|_s$$

- $e = (e : T)$

  Trivially by Induction Hypothesis:

  $$|\Gamma|_s \vdash [\![e]\!]_s : |T|_s$$ □

Lemma 4 (Well-formed Specialization)**.** *Given a well formed type $\Gamma \vdash T \; ok$ and a specialization s, $|T|_s$ is well formed under $|\Gamma|_s$.*

Proof. By induction on the structure of types.

- $T = Int$. Trivial
- $T = X$. Either $s(X) = P$ and all primitive types are well-formed, or s is undefined at X, and $|T|_s$ is well-formed by the hypothesis.

- $T = C[\overline{T}]$. By Induction hypothesis, $|\Gamma|_s \vdash |\overline{T}|_s$ *ok*, and immediately follows that $|\Gamma|_s \vdash |C[\overline{T}]|_s$ *ok*.
- $T = \forall\overline{Y}.\overline{T} \to T$.

  We have that $\overline{Y}, \Gamma \vdash \overline{T}, T$ *ok*. By Induction Hypothesis, we have

  $$|\overline{Y}, \Gamma|_s \vdash |\overline{T}|_s, |T|_s \text{ ok}.$$

  It follows immediately that

  $$|\Gamma|_s \vdash \forall\overline{Y}.|\overline{T}|_s \to |T|_s \text{ ok}. \qquad \square$$

Lemma 5 (Substitution on specialization)**.** *Given a specialization s, $\overline{X}$ and $\overline{Y}$ type variables such that s is not defined anywhere in $\overline{Y}$, and $[\overline{Y}/\overline{X}]s$ is defined, we have*

$$\left|[\overline{Y}/\overline{X}]T\right|_{[\overline{Y}/\overline{X}]s} = |T|_s$$

Proof. By induction on the structure of T.

- $T = X_i$

  $$\begin{aligned} |[\overline{Y}/\overline{X}]X_i|_{[\overline{Y}/\overline{X}]s} = |Y_i|_{[\overline{Y}/\overline{X}]s} &= s(X_i) \\ |X_i|_s &= s(X_i) \end{aligned}$$

- $T = X$, s is not defined at X

  $$\begin{aligned} |[\overline{Y}/\overline{X}]X|_{[\overline{Y}/\overline{X}]s} = |X|_{[\overline{Y}/\overline{X}]Xs} &= X \\ |X|_s &= X \end{aligned}$$

- $T = P$ Trivial
- $T = C[\overline{T}]$

  $$\begin{aligned} [\overline{Y}/\overline{X}]C[\overline{T}] &= C[[\overline{Y}/\overline{X}]\overline{T}] \\ \left|[\overline{Y}/\overline{X}]C[\overline{T}]\right|_{[\overline{Y}/\overline{X}]s} &= \left|C[[\overline{Y}/\overline{X}]\overline{T}]\right|_{[\overline{Y}/\overline{X}]s} \\ &= C[\left|[\overline{Y}/\overline{X}]\overline{T}\right|_{[\overline{Y}/\overline{X}]s}] \end{aligned}$$

  By Induction Hypothesis we have that

  $$C[\left|[\overline{Y}/\overline{X}]\overline{T}\right|_{[\overline{Y}/\overline{X}]s}] = C[|\overline{T}|_s] = |C[\overline{T}]|_s$$

  therefore

  $$\left|[\overline{Y}/\overline{X}]C[\overline{T}]\right|_{[\overline{Y}/\overline{X}]s} = |C[\overline{T}]|_s$$

- $T = \forall \overline{Z}.(\overline{T} \to T)$

  We prove

  $$\left|[\overline{Y}/\overline{X}]\forall \overline{Z}.(\overline{T} \to T)\right|_{[\overline{Y}/\overline{X}]s} = \left|\forall \overline{Z}.(\overline{T} \to T)\right|_s$$

  We assume $\overline{X}, \overline{Y}, \overline{Z}$ are all distinct type variables. We have

  $$\begin{aligned}
  [\overline{Y}/\overline{X}]\forall \overline{Z}.(\overline{T} \to T) &= \forall \overline{Z}.([\overline{Y}/\overline{X}]\overline{T} \to [\overline{Y}/\overline{X}]T) \\
  \left|[\overline{Y}/\overline{X}]\forall \overline{Z}.(\overline{T} \to T)\right|_{[\overline{Y}/\overline{X}]s} &= \left|\forall \overline{Z}.([\overline{Y}/\overline{X}]\overline{T} \to [\overline{Y}/\overline{X}]T)\right|_{[\overline{Y}/\overline{X}]s} \\
  &= \forall \overline{Z}.(\left|[\overline{Y}/\overline{X}]\overline{T}\right|_{[\overline{Y}/\overline{X}]s} \to \left|[\overline{Y}/\overline{X}]T\right|_{[\overline{Y}/\overline{X}]s})
  \end{aligned}$$

  And by Induction Hypothesis we have

  $$\left|[\overline{Y}/\overline{X}]\forall \overline{Z}.(\overline{T} \to T)\right|_{[\overline{Y}/\overline{X}]s} = \forall \overline{Z}.(\left|\overline{T}\right|_s \to |T|_s) \qquad \square$$

Bibliography

[1] Ecma international, standard ECMA-335 Common Language Infrastructure, June 2006.

[2] AGESEN, O., FREUND, S. N., AND MITCHELL, J. C. Adding type parameterization to the Java language. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1997), ACM, pp. 49–65.

[3] AHO, A., SETHI, R., AND ULLMAN, J. *Compilers, Principles, Techniques and Tools*. Pearson Education Singapore, 1986.

[4] ALEXANDRESCU, A. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[5] ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. Adaptive optimization in the Jalape no JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2000), ACM, pp. 47–65.

[6] AYCOCK, J. A brief history of just-in-time. *ACM Comput. Surv. 35*, 2 (2003), 97–113.

[7] BACON, D., WEGMAN, M., AND ZADECK, K. Rapid type analysis for C++. Tech. Rep. pending, IBM Thomas J. Watson Reseach Center, 1996.

[8] BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1996), ACM, pp. 324–341.

[9] BANK, J. A., MYERS, A. C., AND LISKOV, B. Parameterized types for Java. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 132–145.

[10] BJARNE, S. *The C++ Programming Language*. Addison-Wesley, 1987.

[11] BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM, pp. 183–200.

[12] CARTWRIGHT, R., AND STEELE, JR., G. L. Compatible genericity with run-time types for the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1998), ACM, pp. 201–215.

[13] CLICK, C., AND PALECZNY, M. A simple graph-based intermediate representation. *SIGPLAN Not. 30*, 3 (1993), 35–49.

[14] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science 952* (1995), 77–101.

[15] FITZGERALD, R., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., AND TARDITI, D. Marmot: an optimizing compiler for Java. *Softw. Pract. Exper. 30*, 3 (2000), 199–232.

[16] GAGNON, E. M., HENDREN, L. J., AND MARCEAU, G. *Efficient Inference of Static Types for Java Bytecode*, vol. 1824 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2000, pp. 143–155.

[17] GARCIA, R., JARVI, J., LUMSDAINE, A., SIEK, J. G., AND WILLCOCK, J. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (New York, NY, USA, 2003), ACM, pp. 115–134.

[18] GEORGES, A., BUYTAERT, D., AND EECKHOUT, L. Statistically rigorous Java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (New York, NY, USA, 2007), ACM, pp. 57–76.

[19] GOETZ, B. Java theory and practice: Dynamic compilation and performance measurement. http://www.ibm.com/developerworks/java/library/j-jtp12214/, December 2004.

[20] GOETZ, B. Java theory and practice: Anatomy of a flawed microbenchmark. http://www.ibm.com/developerworks/java/library/j-jtp02225.html, February 2005.

[21] GRIESEMER, R., AND MITROVIC, S. A compiler for the Java HotSpot™virtual machine. In *The School of Niklaus Wirth, "The Art of Simplicity"* (2000), dpunkt.verlag/Copublication with Morgan-Kaufmann, pp. 133–152.

[22] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 1997), ACM, pp. 108–124.

[23] HARPER, R., AND MORRISETT, G. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, 1995), pp. 130–141.

[24] HUDAK, P. Building domain-specific embedded languages. *ACM Computing Surveys 28* (1996).

[25] IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. 23*, 3 (2001), 396–450.

[26] JONES, S. L. P., AND LAUNCHBURY, J. *Unboxed values as first class citizens in a non-strict functional language*, vol. 523 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1991, pp. 636–666.

[27] KENNEDY, A., AND SYME, D. Design and implementation of generics for the .NET Common language runtime. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM, pp. 1–12.

[28] KNOOP, J., RÜTHING, O., AND STEFFEN, B. Lazy code motion. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation* (New York, NY, USA, 1992), ACM, pp. 224–234.

[29] KOTZMANN, T., WIMMER, C., MÖSSENBÖCK, H., RODRIGUEZ, T., RUSSELL, K., AND COX, D. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008), 1–32.

[30] LEROY, X. Unboxed objects and polymorphic typing. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1992), ACM, pp. 177–188.

[31] LILJA, D. J. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, September 2005.

[32] LINDHOLM, T., AND YELLIN, F. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[33] MICROSOFT CORPORATION. *C# 4.0 Language Specification*, 2010. see http://msdn.microsoft.com/en-us/vcsharp/.

[34] MIECZNIKOWSKI, J., AND HENDREN, L. J. Decompiling Java bytecode: Problems, traps and pitfalls. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction* (London, UK, 2002), Springer-Verlag, pp. 111–127.

[35] MORRISON, R., DEARLE, A., CONNOR, R. C. H., AND BROWN, A. L. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Program. Lang. Syst. 13*, 3 (1991), 342–371.

[36] MUCHNICK, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, August 1997.

[37] NANDA, S., AND CHIUEH, T.-C. A survey of virtualization technologies. Tech. Rep. ECSL–TR–179, SUNY at Stony Brook, February 2005.

[38] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[39] ODERSKY, M. Scala 2.8 collections, October 2009. http://www.scala-lang.org/sid/3.

[40] ODERSKY, M. The Scala language specification. available at www.scala-lang.org, 2009.

[41] ODERSKY, M., RUNNE, E., AND WADLER, P. *Two Ways to Bake Your Pizza — Translating Parameterised Types into Java*, vol. 1766 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2000, pp. 114–132.

[42] ODERSKY, M., AND WADLER, P. Pizza into Java: translating theory into practice. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1997), ACM, pp. 146–159.

[43] OPENJDK. Project lambda. http://openjdk.java.net/projects/lambda/, 2010.

[44] PALECZNY, M., VICK, C., AND CLICK, C. The java HotSpot^{TM}server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium* (Berkeley, CA, USA, 2001), USENIX Association, pp. 1–1.

[45] PLEVYAK, J., AND CHIEN, A. A. Precise concrete type inference for object-oriented languages. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications* (New York, NY, USA, 1994), ACM, pp. 324–340.

[46] SCHINZ, M. *Compiling Scala for the Java virtual machine*. PhD thesis, EPFL, Lausanne, 2005.

[47] SHIVERS, O. Control flow analysis in Scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (New York, NY, USA, 1988), ACM, pp. 164–174.

[48] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[49] SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. A dynamic optimization framework for a Java just-in-time compiler. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM, pp. 180–195.

[50] SUNDARESAN, V., HENDREN, L. J., RAZAFIMAHEFA, C., VALLÉE-RAI, R., LAM, P., GAGNON, E., AND GODIN, C. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)* (2000), pp. 264–280.

[51] TAFT, S. T., DUFF, R. A., BRUKARDT, R. L., PLOEDEREDER, E., AND LEROY, P. *Ada 2005 Reference Manual. Language and Standard Libraries: International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1 (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[52] TAHA, W. Domain-specific languages. In *International Conference on Computational & Experimental Engineering and Sciences* (2008).

[53] UMANEE, N. Shimple: An investigation of static single assignment form. Master's thesis, McGill University, February 2006.

[54] VALLEE-RAI, R., GAGNON, E., HENDREN, L., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing Java bytecode using the soot framework: Is it feasible? *International Conference on Compiler Construction, LNCS 1781* (2000), 18–34.

[55] VALLÉE-RAI, R., HENDREN, L., SUNDARESAN, V., LAM, P., GAGNON, E., AND CO, P. Soot - a Java optimization framework. In *Proceedings of CASCON 1999* (1999), pp. 125–135.

[56] VELDHUIZEN, T. L. C++ templates are Turing complete. Tech. rep., Indiana University Computer Science, 2003.

[57] VENNERS, B. ScalaTest, 2009. http://www.scalatest.org/.

[58] WIKIPEDIA. List of CLI languages, July 2010. http://en.wikipedia.org/wiki/List_of_CLI_languages.

[59] WIKIPEDIA. List of JVM languages, July 2010. http://en.wikipedia.org/wiki/List_of_JVM_languages.

[60] YU, D., KENNEDY, A., AND SYME, D. Formalization of generics for the .NET common language runtime. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2004), ACM, pp. 39–51.

Curriculum Vitae

Personal Information

Name	Iulian Dragoş
Citizenship	Romanian
Date of birth	2nd of February 1980
Place of birth	Timişoara, Romania

Education

2005 – 2010	Ph.D. program, EPFL
2003	Diploma Thesis, Forschungszentrum Informatik, Karlsruhe
1998–2003	"Politehnica" University of Timişoara, Romania, Faculty of Computer Science

Professional experience

2008	Google (internship)
2005 – 2010	PhD programme at EPFL
2000 – 2003	softNRG, software engineer
1999 – 2000	DNT Timişoara, now Astral Telecom, software engineer