NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM

MASTER THESIS

Faster Scala Collections with Compile-Time Reflection

Georgios Kollias

Supervisor: **Yannis Smaragdakis**, Associate Professor NKUA

ATHENS

MAY 2013

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γρήγορες Δομές Δεδομένων στη Γλώσσα Scala με Στατική Ανάκλαση

Γεώργιος Κόλλιας

Επιβλέπων: **Γιάννης Σμαραγδάκης**, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΙΟΣ 2013

MASTER THESIS

Faster Scala Collections with Compile-Time Reflection

Georgios Kollias

RN: M1049

SUPERVISOR:

Yannis Smaragdakis, Associate Professor NKUA

THESIS COMMITTEE:

Yannis Smaragdakis, Associate Professor NKUA
Panos Rondogiannis, Associate Professor NKUA

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γρήγορες Δομές Δεδομένων στη Γλώσσα Scala με Στατική Ανάκλαση

Γεώργιος Κόλλιας

ΑΜ: Μ1049

ΕΠΙΒΛΕΠΩΝ :

Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ
Παναγιώτης Ροντογιάννης, Αναπληρωτής Καθηγητής ΕΚΠΑ

Περίληψη

Περιγράφουμε την υλοποίηση συγκεκριμένων μεθόδων (`map` και `foreach`) των δομών δεδομένων στη βιβλιοθήκη της γλώσσας Scala, με χρήση λειτουργιών στατικής ανάκλασης. Ο σκοπός είναι η δημιουργία γρηγορότερων δομών μέσω ενσωμάτωσης και βελτιστοποίησης της λειτουργίας στο σημείο κλήσης. Η λειτουργικότητα αυτή περιλαμβάνεται στη βιβλιοθήκη της γλώσσας, έτσι ώστε ταχύτερες λειτουργίες μπορούν να χρησιμοποιηθούν σε όλους τους τύπους δομών της γλώσσας (π.χ. λίστες, πίνακες, κτλ.) χωρίς ανάγκη ορισμού νέων τύπων. Ο μηχανισμός μας έχει υλοποιηθεί κατεύθειαν στη βασική βιβλιοθήκη και το μεταγλωττιστή της γλώσσας. Τα αποτελέσματα είναι ενθαρρυντικά, με πειράματα να εμφανίζουν ταχύτητα βελτιωμένη κατά 40%.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Στατικός Μεταπρογραμματισμός

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ : Scala, μεταπρογραμματισμός, δομές δεδομένων, μεταγλωττιστές, μακροεντολές, ανάκλαση

Abstract

We describe the implementation of specific Scala collections operations (currently the `map` and `foreach` methods) using the Scala 2.10 compile-time reflection facilities. The primary motivation for this work is to create faster collections by inlining operations at the call site. The functionality is available at the standard Scala library level, so that our optimized operations can be used on all plain Scala collection types (e.g., `List`, `Array`, etc.) without the need of creating new specialized types. Our mechanism is implemented directly inside the Scala standard library and by modifying the default compiler. The results are encouraging since benchmarks show a 40% speedup.

SUBJECT AREA: Compile-Time Metaprogramming

KEYWORDS: Scala, metaprogramming, collections, compilers, macros, reflection

Acknowledgements

FiXme
Fatal:
Add me

Contents

List of Figures

List of Tables

List of Corrections

Preface

FiXme
Fatal:
Replace
me

Chapter 1

Introduction

This chapter will introduce our work and the problem it tries to attack. It is a real-world hard problem that affects the whole Java Virtual Machine (JVM) ecosystem.

FiXme Fatal: add abbr

In "Fixing The Inlining Problem" , Cliff Click describes an issue that has emerged the last few years in the JVM ecosystem:

FiXme Fatal: cite

"*The Problem* is simply this: new languages on the JVM (e.g., JRuby) and new programming paradigms (e.g., Fork Join) have exposed a weakness in the current crop of inlining heuristics. Inlining is not happening in a crucial point in hot code exposed by these languages, and the lack of inlining is hurting performance in a major way. AND the inlining isn't happening because The Problem is a hard one to solve; (i.e. it's not the case that we'll wave our wands and do a quick fix & rebuild HotSpot and the problem will go away). John Rose, Doug Lea and I all agree that it's a Major Problem facing JVMs with long and far reaching implications."

Let's see what *The Problem* is with a small example in a Java-like language. The Problem is getting the right amount of context in hot inner loops - which also contain a *megamorphic* virtual call in the loop and not much else. Megamorphic virtual method calls are these whose receiver can have many different runtime types. Figure 1.1 shows a naive implementation of a `map` method that applies a predetermined function to all the elements of a source array and assigns the result to a destination array. In this case we increment all source's elements by one.

```
1 // The function in the inner loop
2 long add1(long a) {return a + 1;}
3 // The iterator function
4 void map(long[] dst, long[] src) {
5   for(int i=0; i < dst.len; i++) // simple loop
6     dst[i] = add1(src[i]); // around a simple loop body
7 }
```

Figure 1.1: Example of naive `map` in Java

```
// A sample iterator function
void map( CallableOneArg fcn1arg, long[] dst, long[] src) {
  for(int i=0; i < dst.len; i++)
    dst[i] = fcn1arg.call(src[i]);
}
```

Figure 1.2: Example of generic `map` in Java

Inlining the function `add1` is crucial to performance here. Without inlining the compiler does not know what the loop body does (because function calls can in general do anything), and with inlining it can understand the entire function completely - and then see it's a simple loop around a stream of array references. At this point the JIT can do range-check elimination, loop unrolling, and prefetching, among other optimizations.

The Problem is that there are multiple variations of `add1` *and* the wrapping iterator gets complex. It's the product of these two parts getting complicated that makes The Problem. In this work, we are mostly interested in the first part, since the Scala collections iterators are not very big or complex.

More often than not, after implementing `map`, we would like to add more functions similar to `add1`. We might also want to add `add2`, `mult3`, `filter` and so on. What we really want is a way to pass in the function to apply on the basic data bits in the innermost loop of our iterator. In Java, we often do this with either a `Callable` or a `Runnable`. Figure 1.2's example uses `Callable`.

We need only one copy of our iterator, and we can apply nearly all kinds of one argument functions. Alas, that inner loop now contains a function call that needs inlining and there are dozens of different functions for `fcn1arg.call`. The JIT does not know which one to inline here, because all these different functions are called at different times. Typically then the JIT does not inline any of them, and instead opts for a virtual call. While the virtual call itself is not too bad, the lack of knowledge of what goes on inside the virtual call prevents all kinds of crucial optimizations: loop unrolling, range-check elimination, all kinds of prefetching and alias analyses.

One solution would be to make the inner function call a static (final) call, which then the JIT can inline. Of course, if we do that we need an iterator for the `add1` version, one for the `add2` version, and one for the `mult3` version, so we need a lot of them. Also, we will need a new one for each new function we can think of; we cannot just name them all up front. So

we will end up with a lot of these iterators each with a custom call in the inner loop. All these iterators will start blowing out the instruction cache on our CPU, and besides it is a pain to maintain dozens of cloned possibly complex iterators.

Several of the Java ecosystem's prominent figures, like Cliff Click, John Rose, Doug Lea, have proposed solutions that range from pure obscure technical to pure educational ones, demonstrating possible *megamorphic inlining friendly* coding styles.

Scala, as most of the modern JVM languages, is no exception and it suffers from The Problem too and, actually, it affects its adoption negatively. At the end of 2011, an email from a YAMMER employee towards the CEO of TYPESAFE, the company backing the Scala ecosystem, about Scala shortcomings, leaked to the public. Most technical issues were related with The Problem and, more specifically, with the Scala standard library collections' performance.

FiXme Fatal: cite http://ww scala

In the next chapters we will see where exactly the problem lies and how our project can help us alleviate it. We have named our project FT-DECLOSURIFY; the *ft* prefix stands for Scala compiler's FastTrack mechanism, as we will see in Chapter 3, and the *declosurify* suffix suggests that the implementation is based on Paul Phillips's original DECLOSURIFY project.

FiXme Fatal: cite github url

ft-declosurify defines two methods, `macroMap` and `macroForeach`, that can be used from all the Scala standard libary collections. They offer the same functionality as their Scala standard library counterparts, `map` and `foreach` methods, but they are much faster because they suffer much less from The Problem. In most cases, `macroMap`/`macroForeach` can be considered as faster and drop-in replacements of the default `map`/`foreach` methods.

Our solution tries to attack The Problem using Scala's new compile-time reflection subsystem. So, the next chapter will give us an overview of the Scala language, its standard library collections and its new compile-time reflection subsystem, in order to prepare us for Chapter 3 where we will explain our project's core functionality and implementation. Chapter 4 provides us with several benchmarks, showing promising performance speedups. Chapter 5 presents some similar work in the area. Finally, Chapter 6 summarizes this work and mentions its drawbacks.

Chapter 2

Background

Scala is a relatively new statically typed programming language that tries to unify the object-oriented and functional programming paradigms into one coherent paradigm, recently called object-functional. Currently, its main implementation runs on the JVM and so its main goal is to provide a more general and uniform superset of Java. Since version 2.8, Scala has a rich collections library and since version 2.10, it has a completely new reflection subsystem . FiXme Fatal: cite FiXme Fatal: cite FiXme Fatal: cite

2.1 Scala Collections Overview

The Scala library systematically distinguishes between mutable and immutable collections. A mutable collection can be updated or extended in place. This means one can change, add, or remove elements of a collection as a side effect. Immutable collections, by contrast, never change. We still have operations that simulate additions, removals, or updates, but these operations will in each case return a new collection and leave the old collection unchanged. We will see in the next chapter how this mutable-immutable separation affects our macro transformation plan.

All collection classes are found in the package `scala.collection` or one of its sub-packages `mutable`, `immutable`, and `generic`. Most collection classes needed by client code exist in three variants, which are located in packages `scala.collection`, `scala.collection.immutable`, and `scala.collection.mutable`, respectively. Each variant has different characteristics with respect to mutability.

A collection in package `scala.collection.immutable` is guaranteed to be immutable for everyone. Such a collection will never change after it is created. A collection in package `scala.collection.mutable` is known to have some operations that change the collection in place.

A collection in package `scala.collection` can be either mutable or immutable. For instance, `collection.IndexedSeq[T]` is a superclass of both `collection.immutable.IndexedSeq[T]` and `collection.mutable.IndexedSeq[T]`. Generally, the root collections

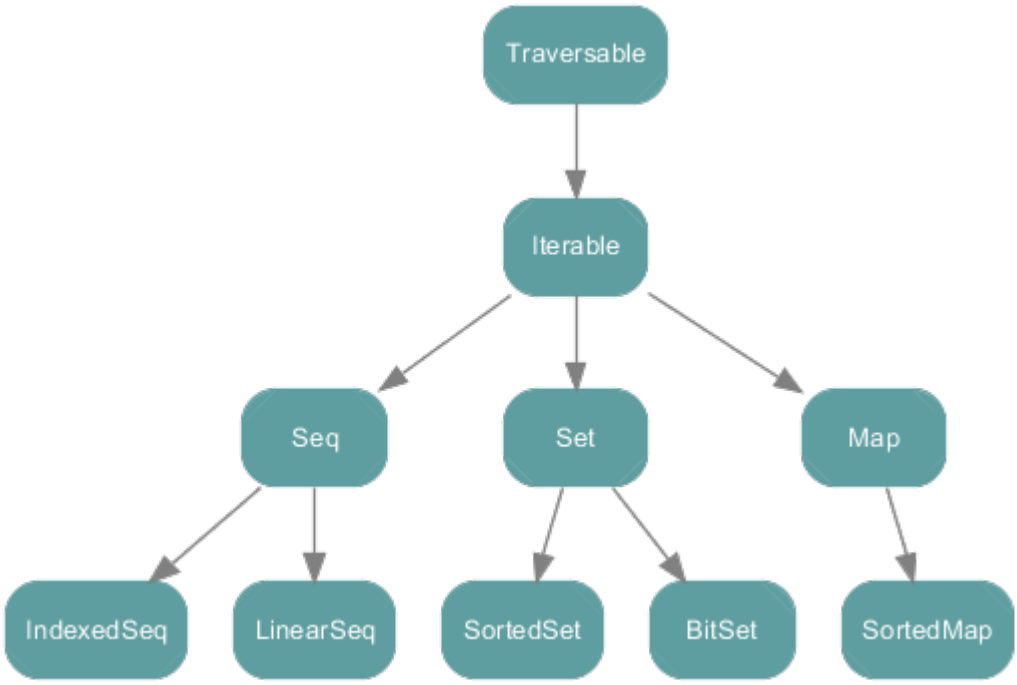


Figure 2.1: Basic collections in `scala.collection`

in package `scala.collection` define the same interface as the immutable collections, and the mutable collections in package `scala.collection.mutable` typically add some side-effecting modification operations to this immutable interface.

By default, Scala always picks immutable collections. For instance, if we just write `Set` without any prefix or without having imported `Set` from somewhere, we get an immutable set because these are the default bindings imported from the scala package. To get the mutable default version, we need to write explicitly `collection.mutable.Set`.

Figure 2.1 shows all collections in package `scala.collection`. These are all high-level abstract classes or traits, which generally have mutable as well as immutable implementations. [1]

This work focuses mainly on `Seq`'s subtree, since it's where we can get the most prominent speedups by exploiting the sequences' properties. A sequence is a kind of iterable that has a length method and whose elements have fixed index positions, starting from 0.

[1] Figure courtesy of Matthias Doenitz

2.2 Scala Compile-Time Reflection Overview

Scala version 2.10, released on , introduced a new reflection subsystem adding both run time and compile metaprogramming capabilities. The new run-time reflection is much more general and feature complete compared to Java's reflection. Compile-time reflection is quite rare in mainstream statically typed programming languages and, currently, it can only be found in more exotic languages like Haskell and Nemerle. Compile-time reflection enabled the introduction of an experimental version of type-safe syntactic macros.

FiXme Fatal: date

FiXme Fatal: cite Template Haskell

FiXme Fatal: cite Nemerle

Syntactic macro systems work at the level of abstract syntax trees and preserve the lexical structure of the original program. Macro systems that work at the level of lexical tokens, like the C preprocessor, cannot preserve the lexical structure reliably. The most widely used implementations of syntactic macro systems are found in Lisp-like languages such as Common Lisp, Scheme. These languages are especially suited for this style of macro due to their uniform, parenthesized syntax (known as S-expressions).

FiXme Fatal: cite Scala Macros, a Technical Report

Compile-time metaprogramming is a valuable tool for enabling such programming techniques as:

- Language virtualization (overloading/overriding semantics of the original programming language to enable deep embedding of DSLs)
- Program reification (providing programs with means to inspect their own code)
- Self-optimization (self-application of domain-specific optimizations based on program reification)
- Algorithmic program construction (generation of code that is tedious to write with the abstractions supported by a programming language)

This work falls in categories three and four, since we use compile-time reflection to generate code programmatically for each `macroMap`/`macroForeach` method, specialized at the call-site for optimization reasons.

Scala's compile-time metaprogramming can be used through Scala's new macro system that allows programmers to write *macro defs*: functions that are transparently loaded by the compiler and executed during compilation.

Our project is implemented directly in the Scala compiler (scalac), so we can use most of the available compile-time metaprogramming capabilities directly, without using macro defs explicitly. In the next chapter we will see how we achieve it.

Compile-time reflection allows us to create new and/or manipulate existing abstract-syntax trees (ASTs) during the compiler's typechecking phase. All the new or changed ASTs are re-typechecked, guaranteeing us type-safe transformations. scalac represents ASTs with objects of type `scala.reflect.api.Tree` or `scala.reflect.api.Exprs`, which is just a typed-wrapper of `scala.reflect.api.Tree`. Through the available compiler APIs, we can create, inspect or change the compiler's `scala.reflect.api.Symbol` and `scala.reflect.api.Type` objects that are related with these ASTs.

FiXme Fatal: abbr

For example, we can create the AST of the Scala expression `x < 10` manually, either with a macro or directly within scalac, with this code `Apply(Select(Ident(newTermName("x")), newTermName("$less"), List(Literal(Constant(10)))))`. `Apply`, `Select`, `Ident`, `Literal`, `Constant` are AST objects themselves of `scala.reflect.api.Tree` type.

Obviously, the AST construction is cumbersome and error-prone. But most probably it is also wrong. If the AST was generated within an internal scalac method or within a macro, the returned AST will be inlined and type-checked at the method/macro call site. But this means that the identifier `x` will be type-checked at a point where it is most likely not visible, or in the worst case they might refer to something else. In the macro literature, this insensitivity to bindings is called non-hygienic. Scala's compile-time reflection solves the non-hygiene problem providing a built-in macro, called `reify`, that produces its tree one stage later.

FiXme Fatal: cite 19, 8

The `reify` macro plays a crucial role in the compile-time metaprogramming. Its definition as a member of `Context` is:

```
1 def reify[T](expr : T): Expr[T] = macro . . .
```

Reify accepts a single parameter expr, which can be any well-typed Scala expression, and creates a tree that, when compiled and evaluated, will recreate the original tree expr. So `reify` is like time-travel: trees get re-constituted at a later stage. If `reify` is called from normal compiled code, its effect is that the AST passed to it will be recreated at run time. Consequently, if `reify` is called from a macro implementation or a method inside scalac, its effect is that the AST passed to it will be recreated at macro-expansion time (which corresponds to run time for macros). This gives a convenient way to create syntax trees from Scala code: pass the Scala code to `reify`, and the result will be a syntax tree that represents that very same code.

For example, `reify(x < 10)` will generate an `Expr` object representing the same AST we created manually before.

More importantly, `reify` packages the result expression tree with the types and values

of all free references that occur in it. This means in effect that all free references in the result are already resolved, so that re-typechecking the tree is insensitive to its environment. All identifiers referred to from an expression passed to `reify` are bound at the definition site, and not re-bound at the call site. As a consequence, macros that generate trees only by means of passing expressions to `reify` are hygienic.

So, in a sense, Scala macros are self-cleaning. Their basic form is minimal and unhygienic, but that simple form is expressive enough to formulate a `reify` macro, which in turn can be used to make tree construction in macros concise and hygienic.

Another important compile-time metaprogramming operation is the *splicing*, which could be described as `reify`'s inverse operation. Using `Expr`'s splice method we can inject an existing AST inside a `reify`'s body.

Reification and splicing operations are crucial to our implementation, as we will see in the next chapter.

Chapter 3

Our Approach: FT-DECLOSURIFY

This chapter presents the core of our work. Section 3.1 will give us a thorough overview of FT-DECLOSURIFY and section 3.2 will describe the major FT-DECLOSURIFY implementation details.

3.1 FT-DECLOSURIFY Overview

For understanding at a high-level what FT-DECLOSURIFY does let's see how an example expression `List(1, 2, 3).map(_ + 1)` is translated from both the scalac and the FT-DECLOSURIFY points of view. `List(1, 2, 3).map(_ + 1)` is a shortcut for `List(1, 2, 3).map(x => x + 1)`, where `x => x + 1` is an anonymous function -closure- that returns its argument incremented by one. Applying that function to the `List(1, 2, 3)` will result in a new list `List(2, 3, 4)`. Since Scala code is translated into Java bytecode at the end and since Java doesn't support any notion of functions inherently, this anonymous function should be translated somehow in constructs that are supported by the Java bytecode. This trivial Scala expression `List(1, 2, 3).map(_ + 1)` is translated internally by scalac to Figure 3.1's code[1].

We can see that scalac converts the `_ + 1` function into a block of code (piece of code between two braces), where a class, called `$anonfun` here, is defined. That class extends

[1]The code listing below as well as most of the following listings are slightly abbreviated for readability reasons

```
1  immutable.this.List.apply[Int](Array[Int]{1, 2, 3}).map[Int, List[Int]]({
2        final class $anonfun extends scala.runtime.AbstractFunction1[Int,Int] with
             Serializable {
3          def <init>(): anonymous class $anonfun = {
4            $anonfun.super.<init>();
5            ()
6          };
7          final def apply(x$1: Int): Int = x$1.+(1)
8        };
9        (new $anonfun(): Int => Int)
10     }, immutable.this.List.canBuildFrom[Int]())
```

Figure 3.1: scalac's internal representation of `List(1, 2, 3).map(_ + 1)`

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That =
{
    def builder = {
      val b = bf(repr)
      b.sizeHint(this)
      b
    }
    val b = builder
    for (x <- this) b += f(x)
    b.result
}
```

Figure 3.2: Scala's default `map` implementation

the `AbstractFunction1[Int,Int]` class, which is an abstract class that represents the functions that accept one integer argument and return another integer. Inside the class, an `apply` method is defined which is called whenever we apply a class object to one integer argument. The `apply` body returns its argument incremented by one, e.g., `val a1 = new $anonfun(); a1(5);` returns 6. Just after the class definition, scalac creates a new object of this class and this is what is actually returned from that block of code. Eventually, the `_ + 1` is substituted by an object of a subclass of a class representing the functions internally, which leads us to the conclusion that the scala source-level closures are translated to regular class objects.

In Figure 3.2 we can see the Scala standard library's `map` implementation. Here `f` is the function object scalac passed during the `map` call, i.e., the object resulted from `new $anonfun()` call from Figure 3.1. `f(x)` invocation will expand to `f.apply(x)` and, therefore, it is a normal method call on object `f`. We can easily see how similar this `map` definition is with the one provided in Figure 1.2. They both suffer from The Problem. Here, `f`'s runtime type will, usually, be different on each `map` call since the functions we pass are generally different. As we explained in Chapter 1, such calls are called mega-morphic virtual calls and, currently, cannot be inlined efficiently by the JVM. So on each `map` we generally have the added overhead of a dynamic call to the passed function object. Even worse, the lack of knowledge of what goes on inside the virtual call prevents all kinds of crucial optimizations.

Let's see how FT-DECLOSURIFY translates the respective piece of code `List(1, 2, 3).macroMap(_ + 1)`. The only difference here is the use of `macroMap` instead of the plain `map`. It's transformed into Figure 3.3's code.

Here we use a `scala.collection.mutable.Builder` object to construct the target col-

```
{
  private def local1(x\$1: Int): Int = x\$1.+(1);
  private def builder1: scala.collection.mutable.Builder[Int,List[Int]] = {
    val b: scala.collection.mutable.Builder[Int,List[Int]] =
immutable.this.List.canBuildFrom[Int].apply();
    b.sizeHint(immutable.this.List.apply[Int](1, 2, 3));
    b
  };
  val buf = builder1();
  var these = immutable.this.List.apply[Int](1, 2, 3);
  while(!these.isEmpty){
    buf.+=(local1(these.head));
    these = these.tail
  };
  buf.result
}
```

Figure 3.3: Expanded `List(1, 2, 3).macroMap(_ + 1)`

lection. The builder itself is created from the `canBuildFrom` object the compiler passed in implicitly, as we will explain more thoroughly in section 3.2. The builder creation and initialization takes place in the `builder1` method. While we traverse the prefix, we apply the `local1` function to all elements and we append (through method +=) them to the builder. When the traversal is over, we call the builder's result method which returns the full target collection we want (a `List` in this example).

In `macroMap`'s transformation, we see that there is no explicit call to any `map` method. Instead, the list's traveral happens directly within a while loop where the `local1` local free method is applied to all of the list's elements. The `local1` method is called a free method since it's not directly attached to any class or object. During scalac's "flatten" phase, it will be lifted and become a method of the enclosing class with a new mangled name. That way `local1`'s receiver runtime type will, likely, remain unchanged during a program's execution. Therefore, it's much easier for JVM to inline `local1` and reason about further optimizations.

We can easily see that FT-DECLOSURIFY works well for cases where the closure is statically available at the call-site, like `List(1,2,3).macroMap(x => x + 1)`, where it will be expanded to something like

```
{
  def local1(x\$1: Int): Int = x\$1.+(1);
  ...
}
```

according to the transformations above. This case doesn't solve The Problem at its core but, instead, it "sidesteps" it at a "metalinguistic" level, by eager compile-time inlining and by "breaking" and transforming the passed function object to a local method.

A more interesting case is when we have something like `List(1, 2, 3).macroMap(myf)`, where we don't know much about the myf function except, let's say, its type is `Int => Int`. Then, `macroMap` will be expanded to something like:

```
1 {
2  def local1(x$1: Int): Int = myf.apply(x$1);
3   ...
4 }
```

Initally, it seems that we fall back to The Problem again, without having any advantage, since `myf.apply` seems to be a megamorphic virtual call again. But, in a program, all `macroMap` instances will be expanded at the call-site and their myf's runtime type will, usually, remain the same during each call and, theoretically, the type profiler can inline each `myf.apply`, which again alleviates The Problem. So, in both cases, we do achieve some wins against The Problem.

In summary, `macroMap` transformation takes advantage of:

- Knowing the static type of `macroMap`'s receiver, because it can apply different transformations depending on the type.
- Knowing the static type of the `macroMap`'s function, because it can transform it in a local method and, making it inlining friendly.
- Having a fixed built-in implementation, applying eager compile-time inlining.

3.2 FT-DECLOSURIFY Implementation Specifics

In this section, we will see the FT-DECLOSURIFY implementation in more detail.

3.2.1 `macroMap/macroForeach` definitions

The `macroForeach` method, just like the standard `foreach`, is meant to traverse all elements of the collection, and apply the given operation, `f`, to each element. The invocation of `f` is done for its side effect only; in fact any function result of `f` is discarded. Similarly,

`macroMap`, just like the standard `map`, traverses all elements of the collection, and applies the given operation, `f`, to each element but, also, it always returns a collection whose type depends on `f`. We could see `macroMap`'s functionality as a superset of `macroForeach`'s functionality and that's true on the implementation side too. The same code handles both cases but, in the case of `macroForeach`, we ignore the result. For that reason and without any loss of generality we will be referring only to `macroMap` in the rest of the text.

`macroMap` is defined in standard library's `scala.collection.TraversableLike` trait, where the default `map` method is defined too. Its definition is an unusual one:

```
1 def macroMap[B, That](f0: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That
      = ???'
```

Instead of a usual implementation on the right hand side, we see a `???`. The `???` is an actual special Scala method which is used to throw `NotImplementedError` exceptions. Our `macroMap` implementation will be generated at compile-time, as we will explain in the next subsections, so no exception is raised.

The above signature is exactly the same as the `map`'s one. Regarding the type parameters, `A` is the collection's element type and, therefore, `f0`'s argument type, `B` is `f0`'s return type, `Repr` is the underlying collection's type and `That` is the generated collection's type. Also, as we said, `macroMap` is a curried method. The first parameter list accepts the operation's function. The second list gets an implicit `CanBuildFrom` object. `CanBuildFrom` objects are builder factories which generate the appropriate `scala.collection.generic.Builder` objects depeding on their type parameters. Generally, builders makes the creation of a new collection out of existing ones easier and more maintable. For example, a `CanBuildFrom[ List[String], Int, Array[Int]]` object will create a `Builder[Int, Array[Int]]` object which can create an `Array` of integers out of a `List` of strings. Since `bf` is implicit, it will usually be generated and passed by the compiler automatically.

3.2.2 Linking `macroMap`/`macroForeach` definitions with implementations

The upstream DECLOSURIFY project can be used as a library adding the -extension- methods `macroMap` and `macroForeach` on all Scala collections that implement the `scala.collection.TraversableOnce` trait and `Array`s, through Scala's implicit conversions. Since, ultimately, we would like to substitute the default `map` and `foreach` methods with

FiXme Fatal: cite implicit conversions

the `macroMap` and `macroForeach` implementations respectively, we had to move the definitions of `macroMap`, `macroForeach` inside the Scala standard library. The Scala standard library doesn't have access to the new compile-time reflection capabilities since it doesn't depend on the `scala-reflect` and `scala-compiler` packages, so we cannot have macro definitions in it. As a solution, we used the scala compiler's *FastTrack* mechanism defined in the `scala.tools.reflect.FastTrack` trait.

`FastTrack` is a low level mechanism of scalac to invoke macro expansions for builtin macros and it is, also, the key machinery used by our implementation in order to register and invoke our transformations. One builtin macro that uses this mechanism is the `reify` function itself that we saw in the previous chapter. The `FastTrack` module uses a registry where one links methods' compiler `Symbol`s with special handler methods. For example, whenever the compiler sees a `Symbol` representing an application of `reify`, it calls its respective `FastTrack` special handler which has access to the reify appication's call-site context and all of its arguments' Abstract Syntax Trees (ASTs), through pattern matching. The call-site context object, generally, holds call-site information like the macro application's enclosing method, its enclosing class, its line in the file. The ASTs contain the internal representation of the macro application like `List(1, 2, 3).macroMap(_ + 1)` i.e., the receiver object, the `macroMap` method call and its function argument. Pattern matching happens upon these ASTs so, theoretically, we can choose to match successfully against `List(1,2,3).macroMap(_ + 1)` but not against `val r = List(1,2,3); r.macroMap(_ + 1)` depending on our needs. If the pattern matching is successful, its respective compiler handler will eventually generate the AST which replaces the macro application's AST at the call-site, completing that way the low-level macro expansion. In our case, whenever scalac finds a `Symbol` of an application of the `scala.collection.TraversableLike`'s `macroMap` method, it triggers the expansion we will describe in the next subsection.

FiXme Fatal: abbr

Regular Scala macros implementations can be generic in the sense that they can be customized with type parameters as any regular Scala method. On top of that, the Scala compiler allows us to "tag" each of these type parameters with special compiler-generated objects, of type (`Weak`)`TypeTag`, that store the type parameters' full types on each call and make them available at runtime. In short, it's Scala's solution against JVM's type erasure. This machinery is especially useful for macros, since we can inspect these `TypeTag` objects and generate the most suitable code for each occasion. Unfortunately, using the low level compiler's `FastTrack` mechanism doesn't permit us using this facility. Instead, we are forced to work directly with the compiler's internal type representations lowering the level of

FiXme Fatal: cite

abstraction we can work with.

`FastTrack`'s special handler for the `macroMap` method, after doing some preprocessing on the pattern matched ASTs, will call the method that does the actual transformation and code generation.

3.2.3 Transformation Method Interface

Firstly, let's see the transformation method's signature and what arguments it obtains from the handler. This method is defined in `scala.tools.reflect.declosurify.Declosurify` object

```
1 def mapInfix[A, B](c0: Ctx)(f0: c0.Expr[A => B], inElemTpe: c0.Type, outElemTpe:
    c0.Type, inCollTpe: c0.Type, outCollTpe: c0.Type, bfTree: c0.Tree): c0.Tree
```

Despite its name, `mapInfix` can generate ASTs for both `macroMap` and `macroForeach`. It's a curried method parameterized on the received function's input (`A`) and return (`B`) types. In the first parameter list, `c0` is a context object as we described it in subsection 3.2.2, giving us access to the macro application's call-site information. The most important field of `c0` is the *prefix* which represents the receiver collection object of the `macroMap` application. We will call it simply prefix from now on. In the second parameter list, `f0` represents the function's AST which is going to be applied to the prefix. Its type, `Expr`, wraps an AST and an internal type tag (`TypeTag`) to provide access to the type of the tree. As we mentioned before, using `Expr` doesn't really help us working at that level of the compiler, since we cannot really exploit the (`Weak`)`TypeTag`'s facilities. The next four arguments are:

- `inElemTpe`: the internal compiler type of the prefix's elements
- `outElemTpe`: the internal compiler type of the elements of the collection our `macroMap` is going to return
- `inCollTpe`: the internal compiler type of the prefix
- `outCollTpe`: the internal compiler type of the collection our `macroMap` is going to return
- `bfTree`: the AST of the implicit `scala.collection.generic.CanBuildFrom` object that got inserted by the compiler at the `macroMap` call-site. This provides us with an

FiXme Fatal: cite adrian moors phd

easy and accurate way to create the appropriate builder for the generated collection taking full advantage of all the existing implicit `CanBuildFrom` objects that are declared in the standard library.

It's easy to observe that all of second parameter list types are prepended with the context object `c0`. This qualified notation realizes the notion of Scala's path-depedent types. For example, here we choose the `Expr` that is defined inside the `c0` object. Generally, if we have two different objects `c0` and `c1` of the same type which include an inner type `MyType`, e.g., through type member or inner class, then `c0.MyType` is a different type from `c1.MyType` in Scala. The same holds for the `c0.Type`s that follow.

3.2.4 Transformation Requirements

Now that we know what our transformation method can operate upon, we can examine the main points of the actual transformation. One of the first things FT-DECLOSURIFY checks is if the `outCollTpe` is Unit. If it's Unit then it means the macro is applied only for its side-effects so it's a `foreach` call. Instead of relying on this heuristic, newer version of the Scala macros provide us with the exact name of the calling method.

After that, we check three conditions to ensure that a typical transformation can take place:

- the `f0` AST is actually a function AST or a block whose last expression is a function, since, in Scala, the value of a block is the block's last expression.
- the `f0` AST doesn't contain any return expressions.
- macro application is enclosed in a method. Currently, that limitation makes the transformation easier.

If any of the above points is not satisfied, then the mapInfix falls back to the default `map`/`foreach` implementation by generating an AST which calls the `map`/`foreach` method on the same prefix object:

```
1 def mkFallbackImpl: c.Tree = {
2   val name: TermName = if (isForeach) "foreach" else "map"
3   val pre = c.prefix.tree
4   val fallbacktree = Apply(Select(pre, name), f0.tree :: Nil)
5   fallbacktree
6 }
```

The next important step is the transformation of the passed closure AST into a local free method AST:

```
def functionToLocalMethod(fnTree: Function): DefDef = {
  val Function(fparams, fbody) = fnTree
  val frestpe = fbody.tpe
  val fsyms   = fparams map (_.symbol)
  val vparams = for (vd @ ValDef(mods, name, tpt, _) <- fparams) yield ValDef(
     mods, name, TypeTree(vd.symbol.typeSignature.normalize), EmptyTree)
  val method  = newLocalMethod(freshName("local"), vparams, frestpe)
  val tree    = DefDef(NoMods, freshName("local"), Nil, List(vparams), TypeTree(
     frestpe), c.resetAllAttrs(fbody.duplicate))

  tree setSymbol method
  c.resetAllAttrs(tree)
  c.typeCheck(tree).asInstanceOf[DefDef]
}
```

For example, the passed closure `x => x + 1` would be transformed to:

```
def local1(x$1: Int): Int = x$1.+(1);
```

A closure like `x => x + y` where `y` is defined in the local scope would be transformed to something like:

```
def local1(x: Int): Int = x.+(TestMacroMapObject.this.y)
```

Also, a closure like `{println("hi"); x => x + 1}` would be transformed to:

```
println("hi");
def local1(x: Int): Int = x.+(1);
```

3.2.5 Transformation Choice and Idiosyncrasies

The transformation choice depends, primarily, upon the prefix's type. Right now, there are three different transformation paths for different kinds of prefixes. We will check them in the following subsections, but all of them generate similar code with the original Scala `map` implementation, although they use more low-level constructs.

The reasons we need differrent implementations for different kinds of prefixes and target collections are:

- Each kind has different API, e.g., different supported methods.

- Different implementation logic is needed for each kind in order to produce faster code, by exploiting each kinds' specific characteristics since the same methods might take different time on different collections.

For example, trait `Seq` has two subtraits `LinearSeq` and `IndexedSeq`. These do not add any new operations, but each offers different performance characteristics: A linear sequence has efficient `head` and `tail` operations, whereas an indexed sequence has efficient `apply`, `length`, and (if mutable) `update` operations.

Also, we will see how important reifying and splicing operations are. The generated ASTs we splice inside the reify are constructed with one of these three ways depending on the occasion:

- we get them directly from the `FastTrack`'s pattern matching.
- we get them by using the `Symbol` and `Type` APIs.
- we construct them manually.

Whatever AST is returned from the `reify` will eventually replace the macro application's AST in the first place.

Finally, we should also mention that in contrast to most of the other collections operations, a `map` operation can generate a collection that has the same type constructor as the prefix collection but possibly with a different element type, or even a entirely different collection type. As an example of the former case, if `f` is a function from `String` to `Int`, and `xs` is a `List[String]`, then `xs map f` gives a `List[Int]`. Likewise, if `ys` is an `Array[String]`, then `ys map f` gives a `Array[Int]`. As an example of the later case,

```
1 Map("a" -> 1, "b" -> 2) map { case (x, y) => y}
```

returns `List(1, 2)` of type `scala.collection.immutable.Iterable[Int]`, which is different from the prefix's `Map` type. The upstream declosurify project could handle partially only the first case. FT-DECLOSURIFY can handle both of the cases successfully due to the introduction of the implicit `CanBuildFrom` object, making it a more general purpose transformation tool.

3.2.6 Array, scala.collection.mutable.ArraysOps and scala.collection.mutable.IndexedSeq Transformation

`scala.Array` and all collections that implement the `scala.collection.mutable.ArrayOps`

```
def mkMutIndexed[Prefix](prefixTree: Tree): c.Tree = {
  val prefix   = c.Expr[Prefix](prefixTree)
  val len      = c.Expr[Int]('xs dot 'length) // might be array or indexedseq
  val call     = c.Expr[Unit](('buf dot 'update)('i, closure('xs('i))))
  def mkResult = c.Expr[Nothing](if (isForeach) mkUnit else 'buf)

  val arrCons  =   Apply(Select(New(TypeTree(outCollTpe)), nme.CONSTRUCTOR), List
     (('xs dot 'length).lhs))
  val builderVal = c.Expr[Prefix](arrCons)

  reify {
    closureDef.splice
    val xs = prefix.splice
    var buf = builderVal.splice
    var i  = 0
    while (i < len.splice) {
      call.splice
      i += 1
    }
    mkResult.splice
  }.tree
}
```

Figure 3.4: Mutable indexed sequences tranformation code

and `scala.collection.mutable.IndexedSeq` traits, like `scala.collection.mutable.ArraySeq`, `scala.collection.mutable.StringBuilder`, `scala.collection.mutable.ArrayBuffer`, share the same transformation, since all of them are *mutable indexed sequences*. The transformation code is in Figure 3.4.

FiXme Fatal: defined on lines XX-YY

As an example, any `macroMap` applications on `Array`s like:

```
Array(1, 2, 3).macroMap(\_ + 1)
```

will expand to Figure 3.5's code.

Exactly the same transformation applies for all the other collections that fall into this category. The reasons we chose this transformation path for this category are:

- only this category's collections accept a length in the constructor
- the length method here takes constant time
- the element selection through xs.apply(i) takes constant time
- the update method is supported, since all collections of this category are mutable, and takes constant time

```
{
  def local1(x\$1: Int): Int = x\$1.+(1);
  val xs: Array[Int] = scala.this.Predef.intArrayOps(scala.Array.apply(1,
scala.this.Predef.wrapIntArray(Array[Int]{2, 3}))).repr();
  var buf: Array[Int] = new Array[Int](xs.length());
  var i: Int = 0;
  while(i.<(xs.length())){
    buf.update(i, local1(xs.apply(i))); // buf(i) = (local1(r(i)));
    i = i.+(1)
  };
  buf
}
```

Figure 3.5: Expanded `Array(1, 2, 3).macroMap(_ + 1)`

Also, all of this category's collections can only be mutated through the update method (the assignment operator) on a specific index, with one exception, the `scala.collection.mutable.ArrayBuffer`. `ArrayBuffer` supports mutating methods for appending/removing elements which could affect the semantics of `macroMap` in case we mutate the underlying collection in the function we pass in `macroMap`. Our transformation seems to behave in the same way as the default `map` method. For example both

```
buf map {x => buf += x; x+1 }
```

and

```
buf macroMap { x => buf += x; x+1 }
```

return `ArrayBuffer(2, 3, 4)`. So, we assume the `macroMap` keeps the same semantics on the `ArrayBuffer` as the original `map` method.

In the next chapter will see how much faster this version is compared to the default `map` on `Array`s and `ArraySeq`s.

3.2.7 `scala.collection.LinearSeq` Transformation

This category includes `scala.collection.{immutable.List, immutable.Stream, immutable.Queue, immutable.Stack, mutable.MutableList, mutable.MutableList, mutable.LinkedList, mutable.DoubleLinkedList}`. All of these collections are *linear*. For example, in this category's transformation we couldn't use the `apply` (for indexing) or the `length` method because both of them take linear time, instead of constant. The transformation is in Figure 3.6 .

FiXme Fatal: defined on lines XX-YY

```
1  def mkLinear(prefixTree: Tree): c.Tree = {
2   val prefix = c.Expr[Lin[A]](prefixTree)
3   val call   = mkCall('these dot 'head)
4
5   reify {
6     closureDef.splice
7     builderDef.splice
8     builderVal.splice
9     var these = prefix.splice
10    while (!these.isEmpty) {
11      call.splice
12      these = these.tail
13    }
14    mkResult.splice
15  }.tree
16 }
```

Figure 3.6: Linear sequences tranformation code

As an example, any `macroMap` applications on `scala.collection.immutable.List` like:

```
1 List(1, 2, 3).macroMap(\_ + 1)
```

will be replaced from Figure's 3.7's code.

This category uses this transformation because:

- `isEmpty` takes constant time
- `head` takes constant time
- `tail` takes constant time

In the next chapter will see how much faster this version is compared to the default `map` on `List`s.

3.2.8 `scala.collection.Traversable` Transformation

This category includes all colletions that don't fit in any of the previous categories. Roughly, it includes all kinds of `Set`s, `Map`s, `Buffer`s and `immutable.IndexedSeq`s. Obviously, since it includes such a broad range of collections we are allowed to use only methods that are supported from all the collections, meaning that we cannot use the `length`, `apply`, `head`, `tail` methods of the previous transformations. The transformation is in Figure 3.8

FiXme Fatal: defined on lines XX-YY

```
{
  private def local1(x\$1: Int): Int = x\$1.+(1);
  private def builder1: scala.collection.mutable.Builder[Int,List[Int]] = {
    val b: scala.collection.mutable.Builder[Int,List[Int]] =
immutable.this.List.canBuildFrom[Int].apply();
    b.sizeHint(immutable.this.List.apply[Int](1, 2, 3));
    b
  };
  val buf = builder1();
  var these = immutable.this.List.apply[Int](1, 2, 3);
  while(!these.isEmpty){
    buf.+=(local1(these.head));
    these = these.tail
  };
  buf.result
}
```

Figure 3.7: Expanded `List(1, 2, 3).macroMap(_ + 1)`

```
 def mkTraversable(prefixTree: Tree): c.Tree = {
  val prefix = c.Expr[Traversable[A]](prefixTree)
  val call   = mkCall('it dot 'next)

  reify {
    closureDef.splice
    builderDef.splice
    builderVal.splice
    val it = prefix.splice.toIterator
    while (it.hasNext)
      call.splice

    mkResult.splice
  }.tree
}
```

Figure 3.8: Traversable sequences tranformation code

```
{
  private def local1(x\$1: Int): Int = x\$1.+(1);
  private def builder1:
scala.collection.mutable.Builder[Int,scala.collection.immutable.Set[Int]] = {
    val b:
scala.collection.mutable.Builder[Int,scala.collection.immutable.Set[Int]] =
immutable.this.Set.canBuildFrom[Int].apply();
    b.sizeHint(r);
    b
  };
  val buf = builder1();
  val it = r.toIterator;
  while(it.hasNext){
    buf.+=(local1(it.next));
  };
  buf.result
}
```

Figure 3.9: Expanded `Set(1, 2, 3).macroMap(_ + 1)`

As an example, any `macroMap` applications on `scala.collection.immutable.Set` like:

```
Set(1, 2, 3).macroMap(\_ + 1)
```

will expand to Figure 3.9's code.

Here, we use the `iterator` method to generate a prefix's iterator object. All collections have an `toIterator` method since all of them implement the `scala.collection.Iterable` trait. Each collection implements it using its `scala.collection.Traversable`'s `foreach` method, which is implemented according to each collection's special characteristics in order to be more efficient.

For the collections that implement the `immutable.IndexedSeq` trait, like `Vector`, we could have used a modified version of the mutable indexed sequences transformation where we would use a builder object to build the target collection, since we cannot mutate it in place. Interestingly, our experiments showed us that solution was slower than the current one.

Chapter 4

Experimental Results

In this chapter we will benchmark all the categories we explored in the previous chapter against their default Scala counterparts. We choose a few representative collections for each category in order to see how the different transformation strategies and the different collections characteristics affect the speedups.

4.1 Setup

We used the ScalaMeter microbenchmarking and performance regression testing framework for the JVM platform. We use the same benchmark template for all the different categories. The code for the Array benchmarking is in Appendix X . In short, each benchmark compares the `macroMap`'s speed against `map`'s speed on different sizes of the same collection applying the same closure, the successor function.

FiXme Fatal: cite FiXme Fatal: appendix

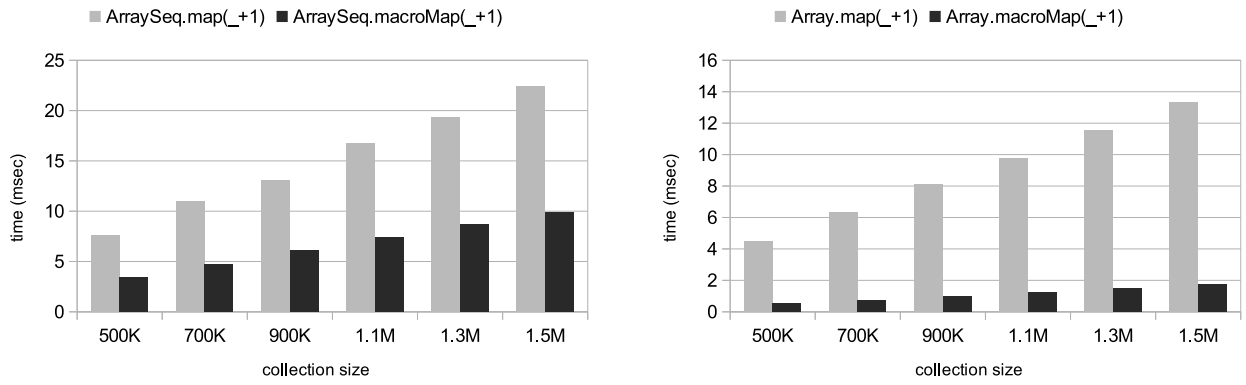
The benchmarks run on a Linux 64-bit machine with an Intel Core i7-2720QM 8-core CPU and 8GB RAM.

In order to make the benchmarking process more stable, reproducible and reliable we have tweaked the benchmark template with the following parameters:

- each measurement of `map`/`macroMap` on a specific collection size is run 25 times successively
- the 25 measurents are divided on 5 separate JVM instances
- every 2 measurements the current collection is reinstantiated
- every 2 measurements a full GC cycle is forced

4.2 Evaluation

In Figure 4.1 we see the benchmark results for `scala.collection.mutable.ArraySeq` and `Array` respectively, as representatives of the mutable indexed sequences category, i.e., subtypes of `scala.collection.mutable.IndexedSeq` and `Array`.



(a) `mutable.ArraySeq` benchmark

(b) `Array` benchmark

Figure 4.1: Benchmarks of mutable indexed sequences representatives

The speedup on `Array` is impressive and the highest one among all the other benchmarked collections. In this series of benchmarks, all collections hold integers and the supplied closure returns an integer too (the successor integer). The Scala language has only one integer type, `scala.Int`, which is similar to Java's boxed `java.lang.Integer`. scalac tries to use Java's respective primitive types under the hood, whenever possible, for optimization reasons. More specifically, after scalac's erasure phase, a `scala.Int` will become either a `int` or a `java.lang.Integer`. For example, in our benchmarks, the closure function will become a local method like:

```
1 def local1$1(x$1: int): int = x$1.+(1);
```

This method deals solely with Java's primitive integers under the hood. On the other hand, operations on `ArraySeq`, `List` and most of the collections except `Array`, deal with `java.lang.Integer`. So, for example, in an `ArraySeq` transformation the target collection is constructed like this:

```
1 buf.update(i, scala.Int.box(local1$1(scala.Int.unbox(xs.apply(i))))
```

while on `Array`s we have:

```
1 buf.update(i, local1$1(xs.apply(i)))
```

This is due the way `Array`s are constructed. In particular, when we ask for an `Array[scala.Int]` in Scala, the compiler inspects it and creates an `Array[int]` automatically and, as a consequence, all of its methods operate upon primitive integers. We can

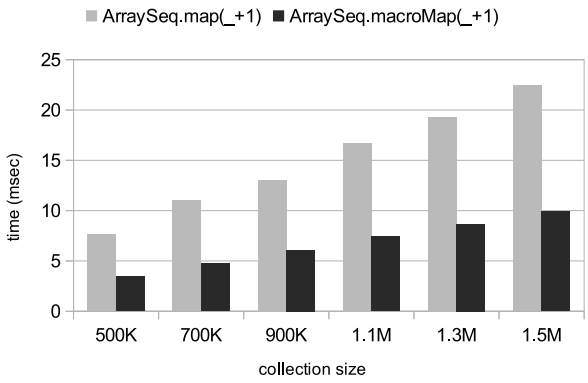


Figure 4.2: `List` benchmark

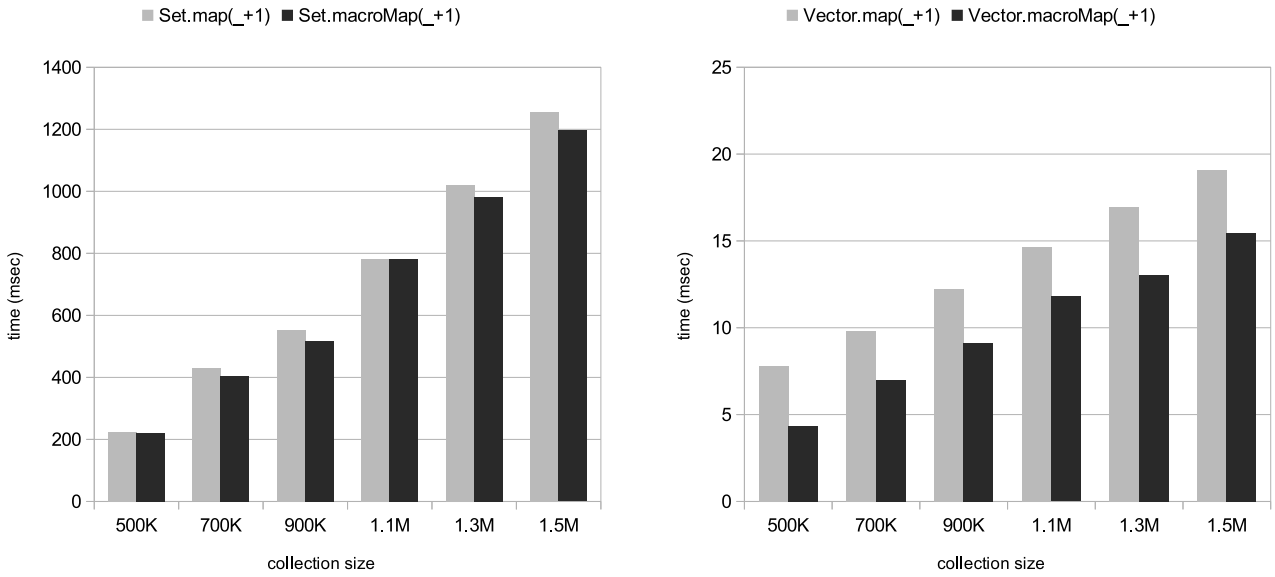
see that avoiding the boxing/unboxing operations can give us a huge performance boost. For that reason, Scala already provides the `@specialized` annotation for creating specialized classes/collections and, more generally, specialization is a hot research topic in the Scala ecosystem .

FiXme Fatal: cite FiXme Fatal: cite mini-boxing and I.Dragos' PhD

In Figure 4.2 we see the benchmark results for `scala.collection.immutable.List`, representative of the linear sequences category, i.e., subtypes of `scala.collection.LinearSeq`.

In Figure 4.3, we see the benchmark results for `scala.collection.immutable.Set` and `scala.collection.immutable.Vector`, representatives of the general category of traversables, i.e., subtypes of `Traversable`, which includes all collections.

Finally, Table 4.1 includes all benchmarks and their associated speedups.



(a) `immutable.Set` benchmark

(b) `immutable.Vector` benchmark

Figure 4.3: Benchmarks of `Traversable` representatives

			Collection Sizes					
Transformation Category	Collection	Method	500K	700K	900K	1.1M	1.3M	1.5M
Mutable Indexed Seq.	ArraySeq	map	07.63	11.02	13.06	16.71	19.28	22.42
		macroMap	03.46	04.75	06.06	07.42	08.67	09.92
		speedup	54.65%	56.89%	53.59%	55.59%	55.03%	55.75%
	Array	map	04.46	06.30	08.08	09.79	11.55	13.34
		macroMap	00.56	00.71	00.96	01.21	01.47	01.74
		speedup	87.44%	88.73%	88.11%	87.64%	87.27%	86.95%
Linear Seq.	List	map	08.49	12.17	14.76	17.83	20.58	23.08
		macroMap	06.26	08.49	10.80	12.77	15.41	16.79
		speedup	26.27%	30.24%	26.83%	28.37%	25.12%	27.25%
Traversable	Set	map	225.10	428.75	552.36	783.19	1021.15	1256.04
		macroMap	221.50	404.77	517.20	780.34	980.57	1196.13
		speedup	01.60%	05.60%	06.37%	00.36%	03.40%	04.77%
	Vector	map	07.80	09.82	12.20	14.64	16.93	19.06
		macroMap	04.35	06.97	09.13	11.80	13.05	15.43
		speedup	44.23%	29.02%	25.16%	19.39%	22.92%	19.05%

Table 4.1: Collections Benchmarks and Speedups

Chapter 5

Related Work

FiXme
Fatal:
Replace
me

Chapter 6

Conclusions

By using the Scala 2.10 compile-time reflection APIs we were able to add the `macroMap`/`macroForeach` methods to the Scala standard library and make them about 40% faster than the default `map`/`foreach` methods, at average. That performance improvement is, in part, due to our choice to trade off the implicitly and programmatically duplicated code at the call site for the reduction of the megamorphic virtual calls, alleviating that way one part of The Problem.

Even if the `macroMap`/`macroForeach` iterators aren't particularly big or complex, this trade off's code duplication can blow out CPU's instruction cache in programs which use these methods heavily, affecting the performance negatively. Also, applying these transformations statically and eagerly at compile-time, lead us to a few more problems:

- We cannot override a collection's `macroMap`/`macroForeach` methods, violating that way Liskov's Substitution Principle. .
- In code like `Traversable[Int] tr = List(1,2,3); tr.macroMap(_+1)`, `macroMap` will expand according to `tr`'s static type, so it won't expand to the most performant alternative.

FiXme Fatal: cite LSP

We intend to continue improving our implementation and looking for new ways to improve Scala's performance in general.

Acronyms and Abbreviations

Abbreviation	Full Name

Appendix A

`mapInfix` Code???

FiXme
Fatal:
Add me