NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM

MASTER THESIS

Faster Scala Collections with Macros

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

Γρηγορότερες Δομές Δεδομένων στη Scala με χρήση Macros

Γεώργιος Κόλλιας

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

ΑΘΗΝΑ

ΜΑΙΟΣ 2013

MASTER THESIS

Faster Scala Collections with Macros

Georgios Kollias

RN: M1049

SUPERVISOR:

Yannis Smaragdakis, Associate Professor NKUA

THESIS COMMITTEE:

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

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Γρηγορότερες Δομές Δεδομένων στη Scala με χρήση Macros

Γεώργιος Κόλλιας

ΑΜ: Μ1049

ΕΠΙΒΛΕΠΩΝ :

Γιάννης Σμαραγδάκης, Αναπληρωτής Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ:

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

Περίληψη

Φιξμε

Φαταλ:

Ρεπλαξε

με

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: …

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: …

Abstract

…

SUBJECT AREA: …

KEYWORDS: …

Acknowledgements

FiXme
Fatal:
Add me

Contents

List of Figures

List of Tables

List of Corrections

Todo list

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. This work tries to mitigate many of its implications. In the next chapter, we will have an overview of the Scala language, its standard library collections and its new compile-time reflection subsystem in order to prepare us for Chapter X where we will explain our project's implementation and core functionality. Chapter X provides us with several benchmarks, showing promising performance speedups. Chapter X presents some similar work in the area. Finally, Chapter X summarizes this work.

FiXme Fatal: add abbr

Section: The Problem and Our Approach

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 mega-morphic virtual call in the loop and not much else. Here is 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 }
```

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`:

```
1 // A sample iterator function
2 void map( CallableOneArg fcn1arg, long[] dst, long[] src) {
3   for(int i=0; i < dst.len; i++)
4     dst[i] = fcn1arg.call(src[i]);
5 }
```

We need only 1 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 complaints 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 X, and the declosurify suffix suggests that the implementation is based on Paul Phillips's original DECLOSURIFY project fxfatalcite 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. The next chapter will give us an overview of both the Scala collections and the new compile-time reflection capabilities.

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 subpackages `mutable`, `immutable`, and `generic`. Most collection classes needed by client code exist in three variants, which are located in packages `scala.collection`, `scala.collection.imm` 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 in package `scala.collecti` define the same interface as the immutable collections, and the mutable collections in pack-

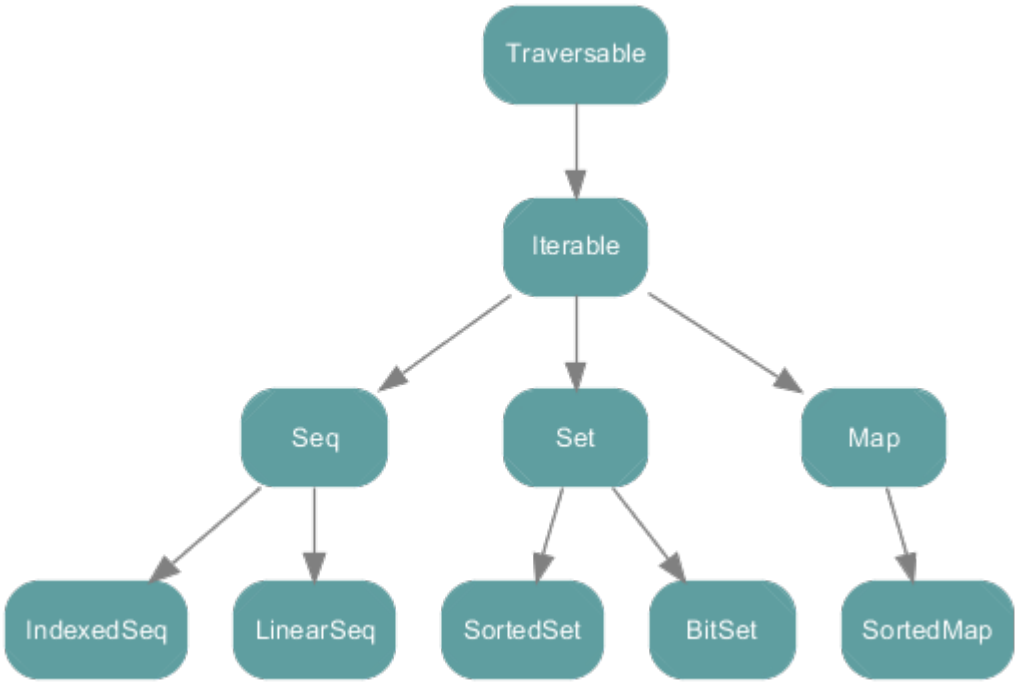

Figure 2.1: Basic collections in `scala.collection`

age `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]

And the following figure shows all collections in package `scala.collection.mutable`. (figure from ...)

[figure]

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 cite Scala Macros, a Technical Report.

FiXme Fatal: date

FiXme Fatal: cite Template Haskell

FiXme Fatal: cite Nemerle

FiXme Fatal: cite each language

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, ISLISP and Racket . These languages are especially suited for this style of macro due to their uniform, parenthesized syntax (known as S-expressions).

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 reflection APIs, we can create, inspect or change the compiler's `scala.reflect.api.Symbols` and `scala.reflect.api.Types` 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 FiXme Fatal: cite [19, 8]. Scala's compile-time reflection solves the non-hygiene problem providing a built-in macro, called `reify`, that produces its tree one stage later.]

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, which we will see in the next chapter.

Chapter 3

Over-Approximating Escaped Objects

FiXme
Fatal:
Replace
me

Chapter 4

Safe Publication

FiXme
Fatal:
Replace
me

Chapter 5

Experimental Results

FiXme
Fatal:
Replace
me

Chapter 6

Related Work

FiXme Fatal: Replace me

Chapter 7

Conclusions

FiXme
Fatal:
Replace
me

Acronyms and Abbreviations

Abbreviation	Full Name

Appendix A

Escape Analysis Code

FiXme
Fatal:
Add me