Elements of Multiparadigm Programming in Scala

Doctoral Dissertation
2020

Gergely Attila Nagy
gergely.a.nagy@gmail.com

Thesis advisor: Dr Zoltán Porkoláb, docent
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School
Doctoral program: Foundations and Methodologies of Informatics
Head of the doctoral school: Prof. Dr Erzsébet Csuhasj-Varjú
Head of the doctoral program: Prof. Dr Zoltán Horváth
# Contents

1 Introduction 4  
1.1 Scala as a Multiparadigm Language .......... 4  
1.2 The Motivation for our Research ............ 5  
1.3 The Structure of this Dissertation .......... 5  
  1.3.1 Improvements to The Type System of Scala ◆ 7  
  1.3.2 Effects of Error Handling on Software Complexity ◆ 8  
  1.3.3 Scale-free Properties of Software Systems ◆ 8  
  1.3.4 Read-Copy-Update in Scala ◆ 9  

2 Theoretical Foundations ◆ 10  
2.1 Programming Paradigms ................. 10  
  2.1.1 Imperative Programming ............... 10  
  2.1.2 Declarative Programming .......... 14  
  2.1.3 Other Programming Paradigms .......... 18  
2.2 Multiparadigm Programming ............... 20  
2.3 Software Metrics ...................... 21  
  2.3.1 Complexity Metrics .......... 21  
  2.3.2 Coupling Metrics .......... 23  

3 Improvements to The Type System of Scala ◆ 25  
3.1 Motivation .................................. 25  
  3.1.1 Shortcomings of The Type Inference Algorithm ........ 25  
  3.1.2 Performance and Maintainability Concerns of Implicit Usage . 28  
3.2 Related Works ................................ 31  
  3.2.1 Typing in Scala .................. 31  
  3.2.2 Success Typing .................. 32  
  3.2.3 Type Inference in Modern C++ ...... 33  
  3.2.4 Implicit Resolution ........ ....... 35  
3.3 Theoretical Resolution .................... 36  
  3.3.1 The $\lambda_s$ Language .............. 36  
  3.3.2 Derivation Rules .............. 37
## Contents

3.4 Practical Approach ................................................. 40  
3.4.1 Typing Recursive Functions ................................. 40  
3.4.2 Performance Analysis of Implicit Resolution ............... 42  
3.5 Evaluation .......................................................... 43  
3.5.1 Typing Simple Recursive Functions ............................ 43  
3.5.2 Typing Recursive Functions with Multiple Branches ......... 46  
3.5.3 Performance Aspects of Implicit Resolution ................. 46  
3.5.4 Approaches to Improve Implicit Resolution Performance ... 49  
3.6 Thesis I. ............................................................... 53  

4 Effects of Error Handling on Software Complexity ............... 54  
4.1 Motivation .............................................................. 54  
4.2 Related Works ......................................................... 56  
4.2.1 The A-V Complexity Metric .............................. 56  
4.2.2 Monads ....................................................... 58  
4.2.3 Error Handling with Monads ............................... 59  
4.2.4 Error Handling in Pure Functional Languages .......... 60  
4.3 Theoretical Resolution ............................................. 60  
4.3.1 Extending Metrics for Exception Handling ................. 60  
4.3.2 Weyuker’s Properties ...................................... 62  
4.4 Evaluation ............................................................ 64  
4.4.1 Comparing Exceptions and Other Structured Error Handling ... 64  
4.4.2 Comparing Exceptions and Monadic Error Handling ........ 66  
4.5 Thesis II. ............................................................... 73  

5 Scale-free Properties of Software Systems ....................... 74  
5.1 Motivation .............................................................. 74  
5.2 Related Works ......................................................... 75  
5.2.1 Scale-free Networks ...................................... 75  
5.3 Theoretical Resolution ............................................. 77  
5.3.1 Definitions ....................................................... 77  
5.3.2 Hypotheses ....................................................... 80  
5.4 Practical Approach .................................................. 80  
5.5 Evaluation .............................................................. 81  
5.6 Thesis III. ............................................................... 83  

6 Read-Copy-Update in Scala ........................................ 84  
6.1 Motivation .............................................................. 84  
6.2 Related Works ......................................................... 84  
6.2.1 Amdahl’s Law ....................................................... 84
CONTENTS

6.2.2 Scalability ............................................. 85
6.2.3 Read-Copy-Update .................................. 85
6.3 Theoretical resolution ................................. 87
   6.3.1 The happens-before relationship ................. 87
   6.3.2 Guarantees of a read-write lock ................. 91
   6.3.3 Guarantees of RCU .................................. 92
6.4 Practical Approach .................................... 92
   6.4.1 The RCU API ...................................... 93
   6.4.2 The Naïve Implementation ....................... 93
   6.4.3 Using Read-write Locks ......................... 93
   6.4.4 Correctness of the Read-Write Lock RCU .......... 95
6.5 Evaluation ............................................. 95
   6.5.1 Concurrent HashMap in Scala ................... 96
   6.5.2 RCU-based Concurrent HashMap in Scala ........ 96
   6.5.3 Benchmarks ....................................... 96
   6.5.4 Results ........................................... 98
6.6 Thesis IV. ............................................ 99

7 Summary ................................................. 100
   7.1 Improvements to The Type System of Scala ........ 100
      7.1.1 Thesis I. ..................................... 101
   7.2 Effects of Error Handling on Software Complexity .... 101
      7.2.1 Thesis II. ................................... 102
   7.3 Scale-free Properties of Software Systems .......... 102
      7.3.1 Thesis III. .................................. 102
   7.4 Read-Copy-Update in Scala ....................... 103
      7.4.1 Thesis IV. .................................. 103

My Publications ........................................ 104

My Lectures ............................................... 106

Bibliography ............................................. 107

Appendices ................................................ 117
   A.1 Multiparadigm Programming Language Elements in Scala .... 118
      A.1.1 A Unified Type System ....................... 118
      A.1.2 Classes and Traits ........................... 119
      A.1.3 Case Classes ................................. 119
      A.1.4 Functions as First-class Citizens ............ 120
      A.1.5 Generics, Variances and Structural Types .... 120
A.1.6 Type Inference ........................................ 121
A.1.7 Type Fields and Self Types .......................... 121
A.1.8 Implicit ............................................... 122
A.1.9 Pattern Matching .................................... 122
A.1.10 Extending the Compiler ............................. 124

A.2 Dissertation Summaries ................................. 125
A.2.1 Dissertation Summary ................................. 125
A.2.2 Disszertáció Összefoglaló ......................... 126
## List of Figures

1. The $\lambda_s$ language. .................................................. 36
2. The syntax of simple recursive functions. .............................. 37
3. Derivation rules for $\lambda_S$. ............................................. 38
4. Average distribution of time spent on each compile phase. ........ 42
5. Compilation times with a constant number of available implicits and increasing implicit usage. ................................. 47
6. Compilation times with constant implicit usage and an increasing number of available implicits. ................................. 48
7. One explicit function call and one implicit resolutions with a varying number of implicits in scope. ................................. 49
8. 1000 explicit functions call and 1000 implicit resolutions with a varying number of implicits in scope. ................................. 50
9. $WACG$ distributions for the Scala compiler. .......................... 81
10. $ACG$ distributions for the Scala compiler. ............................ 82
11. $WACG$ distributions for different versions of the Scala compiler. . . 82
12. $ACG$ distributions for different versions of the Scala compiler. . . 82
13. Amdahl’s law formulated. .................................................. 84
14. RCU operations example. .................................................. 86
15. Benchmark results I. ....................................................... 98
16. Benchmark results II. ....................................................... 98
17. The unified type system of Scala. ........................................ 118
List of Tables

1 Compile times (µs) with and without using our extension. .............. 41
2 Results of analyzing Tomcat. ............................................. 66
3 Summary tables of the computed complexity changes between exceptions and monads. ............................................. 72
4 Descriptive statistics of the benchmark results. ......................... 99

List of Listings

1 A recursive function with no defined return type. ...................... 26
2 A recursive function with return types of a hierarchy. ................. 26
3 A recursive map implementation with explicit type annotation. ....... 27
4 A multi-level class hierarchy. ............................................. 27
5 A recursive function using type hierarchy on Listing 4. with explicit type annotations. ............................................. 28
6 Implicit class example. ..................................................... 29
7 Implicit argument list example. ......................................... 30
8 Implicit method example. .................................................. 30
9 Definition of an STL Vector iterator in C++03. ......................... 33
10 Definition of the same iterator from Listing 9 in C++11. ............. 33
11 Using the auto keyword for function return type. .................... 34
12 Using the auto keyword to infer function return types in C++14. ... 34
13 Using the auto keyword to infer return types of recursive functions in C++14. .................................................. 35
14 A non-compiling recursive function due to wrong branch order. ... 35
15 A recursive function with no defined return type. ...................... 44
16 A recursive map implementation with type inference. ................. 44
17  A recursive function using type hierarchy on Listing 4. with type inference. ................................. 45
18  A function with multiple recursive branches. ................................................................. 46
19  Scala monad. ............................................................. 58
20  Scala functions without exceptions. ................................................................. 65
21  Scala functions with exceptions. ................................................................. 65
22  Error Handling with try-catch and Try ................................................................. 67
23  Client code for Listing 22. ................................................................. 67
24  Error handling with higher-order functions. ................................................................. 68
25  Testing for multiple errors. ................................................................. 68
26  Client code for Listing 25. ................................................................. 69
27  Transformation steps. ................................................................. 69
28  Client code for Listing 27. ................................................................. 69
29  Client code for Listing 27 using a for-comprehension. ................................................................. 70
30  Client code for Listing 27 providing default values. ................................................................. 70
31  Client code for Listing 27 providing default values using Trys I. ................................................................. 70
32  Client code for Listing 27 providing default values using Trys II. ................................................................. 71
33  Asynchronous code and exceptions. ................................................................. 71
34  Asynchronous code using monadic error handling. ................................................................. 71
35  Single-threaded happens-before relation. ................................................................. 88
36  Synchronized-block happens-before relation. ................................................................. 88
37  Volatile happens-before relation. ................................................................. 89
38  Volatile happens-before relation. ................................................................. 89
39  Volatile happens-before relation. ................................................................. 90
40  Deadlock caused by throwing an exception. ................................................................. 90
41  Possible deadlock resolved with using a finally-block. ................................................................. 91
42  RCU Scala API. ................................................................. 93
43  RCU using basic locking logic. ................................................................. 94
44  RCU using read-write locks. ................................................................. 94
45  RCUHashMap += implementation. ................................................................. 97
46  RCUHashMap get() implementation. ................................................................. 97
47  A class constructor. ................................................................. 119
48  A sum type using case objects. ................................................................. 119
49  A product type using case classes. ................................................................. 120
50  Usage patterns for functions. ................................................................. 120
51  Structural type in a generic class definition. ................................................................. 121
52  Structural type in a type field definition. ................................................................. 122
53  A product type using case classes. ................................................................. 123
Chapter 1

Introduction

Software programming paradigms define the frameworks in which software developers solve problems. As the number and the degree of variance of problems solved by software increases, we need more rigorous but freer paradigms. Furthermore, we can also see an uptick in mixing programming styles within the same language to help programmers tackle problems more easily. This has also paved the way for true multiparadigm programming.

1.1 Scala as a Multiparadigm Language

Scala is a general-purpose, multiparadigm programming language. It had been designed from the ground up to be scalable in the sense that it should be able to support solving a wide variety of problems and it should be able to grow with the demands of its users. Compared to other languages inheriting features from outside of their core paradigm (such as Java, C# or C++), Scala has always been a truly multiparadigm language, supporting object-oriented and functional programming equally. In its current incarnation, it is compiled to JVM Bytecode and supports full interoperability with other JVM-based languages, making a virtually infinite number of libraries available. While supporting all OOP principles, Scala advocates using functional programming constructs (such as immutability, pattern matching or algebraic data types) where they make sense. On the other hand, the designers of the language also admit that not all problems can be efficiently solved in pure languages, so they added support for structured programming as well. This realization is what makes Scala a really powerful tool; it lets the programmer decide what construct is best fitted for the problem at hand and then they have the freedom to mix different paradigms using the same environment and toolchain. It has a strong, static type system with support for many advanced features missing from other OOP languages (e.g. type fields, type variance or structural typing). It has built-in support for lazy
evaluation, pattern matching and currying, among many others. Its object-oriented
capabilities are extended by traits and the multiple inheritance model they support,
algebraic data types and generic programming. It is widely used in both academia
and the industry, creating a continuously evolving infrastructure and adding more
and more features to the language. Its implementation is open-source – including
the compiler and the standard library of the language – and has a very open archi-
tecture. This makes it a prime candidate for various types of analyses, prototyping
and experimentation.

1.2 The Motivation for our Research

Scala has gained a respectable level of popularity in the software development in-
dustry, making it appear in the top 40 of all programming languages in the TIOBE
index [121]. Its popularity stems from the fact that it is very flexible, provides high
levels of abstraction and the possibility to easily write sound programs in it due to
the rigorous type system. Since it runs on the JVM and has always kept an easy
way to integrate with Java code, it offers a seamless upgrade path from one of the
most popular programming languages and its numerous libraries. Its origins root in
academia, extending the range of applied theories of type systems, compiler theory
and computation theory. Due to its background it is also popular among researchers
and has many academic applications as well. It is natural that these two applications
of the language have different areas of interest. The motivation for our investigations
come from this discrepancy. We have been using the language daily and had the op-
pportunity to experience the need for the improvements outlined in this dissertation.
Our goal was to apply scientific rigour to the problems while keeping real-world,
direct industry applications in focus.

1.3 The Structure of this Dissertation

This dissertation is structured as follows.

In Chapter 2 we will lay down the theoretical foundations that are required for
the later parts of the dissertation. In this chapter we give a brief introduction to
programming paradigms in Section 2.1 then to multiparadigm languages in Section
2.2. In Section 2.3 we describe software metrics in general, then focus on complexity
(Subsection 2.3.1) and coupling metrics (Subsection 2.3.2).

In Chapter 3 we investigate how certain type system features affect long-term
maintainability of code bases and how these affect the performance of the compiler.
There are two aspects of our analysis: first we investigate a novel approach to typing
simple recursive functions in Scala then focus on the performance characteristics of
implicit resolution. We start the Chapter by providing the motivation behind our work. In Subsection 3.1.1 we detail at a high level how type inference in Scala works then we discuss one of its biggest shortcomings: it does not support inferring types of recursive definitions. We give two examples where this can be especially problematic. Following this we turn to the performance and maintainability concerns of using implicits in Subsection 3.1.2. We discuss the types of implicits and give details about a real-world project that was used for our analyses. In Section 3.2 we list works related to the topic, including typing in Scala, success typing and type inference in C++ as well as describe implicit resolution in more detail. In Section 3.3 we create the theoretical background for our typer extension first by defining a formal language (Subsection 3.3.1) and giving its derivation rules (Subsection 3.3.2). In Section 3.4 we discuss our practical approach, including how typing recursive functions works in practice (Subsection 3.4.1) and how we analyzed the performance implications of implicit resolution (Subsection 3.4.2). We evaluate our results in Section 3.5, where in Subsection 3.5.1 we revisit the recursive examples from our motivation then in Subsection 3.5.4 we propose a guideline that helps mitigate the aforementioned concerns of implicit resolution. The Chapter concludes in Section 3.5.4.

Chapter 4 is dedicated to analyzing how software complexity is affected by various error handling methods. In Section 4.1 we illustrate why we believe this is an important topic, then in Section 4.2 we investigate related analyses: we describe the A-V complexity metric, monads, error handling with monads and other error handling approaches in pure functional languages in Subsections 4.2.1, 4.2.2, 4.2.3 and 4.2.4, respectively. Theoretical resolution of the problem is presented in Section 4.3, firstly by extending complexity metrics (Subsection 4.3.1) then evaluating Weyuker’s properties (Subsection 4.3.2). We evaluate our theories in Section 4.4, first by comparing traditional error handling to exceptions in Subsection 4.4.1 then doing the same for monads and exceptions in Subsection 4.4.2. We then summarize our results in Section 4.4.2.

Our focus in Chapter 5 is the coupling of components in complex software systems. The interest in such system is depicted in Section 5.1. In the next Section (namely Section 5.2) we introduce scale-free networks. We then shift our focus to the foundations of the problem in Section 4.3: we define new metrics in Subsection 5.3.1 and construct hypotheses in Subsection 5.3.2. After using the aforementioned definitions in Section 5.4 and evaluating them in Section 5.5 we conclude this topic in Section 5.5.

In Chapter 6 we describe the read-copy-update locking strategy and how it can be used in Scala. The reason for our interest in the topic is described in Section 5.1. In Section 6.2 we present related works: Amdahl’s law, scalability in general and read-copy-update (Subsection 6.2.1, 6.2.2 and 6.2.3, respectively). The approach we
have taken is displayed in Section 6.4: first we describe our RCU API in Subsection 6.4.1, then we give two implementations – first a naive one in Subsection 6.4.2 and another using read-write locks (Subsection 6.4.3). Evaluating these happens in Section 6.5, first by introducing the concurrent hash map in Scala (Subsection 6.5.1), giving a new implementation of it in Subsection 6.5.2, how we measured its performance (Subsection 6.5.3) and describing the results in Subsection 6.5.4. The Chapter is summarized in Section 6.5.4.

The dissertation is closed by Chapter 7 where we summarize the results.

1.3.1 Improvements to The Type System of Scala

Motivation

A key goal of Scala has always been providing ways to write terse code. The less code the programmer needs to write, the less errors they can make. This has had an effect on many features of the language, including the type system of Scala – most notably on its type inference features and the its support for implicits. Scala supports type inference for most language constructs, including local variables, values and function return types. Unfortunately, the current version of the Scala compiler does not support type inference of recursive definitions at all. This exposes issues not only by exposing nonconforming requirements, but in many cases it advocates possibly erroneous definitions where programmers might choose unnecessarily wide types for their recursive functions. Similarly to other modern languages, Scala supports implicit programming elements. This helps with reducing clutteredness of the code, improving readability and most importantly, embedding more fluid domain-specific languages in core Scala. This comes with a cost though: the compiler needs apply the implicits itself based on a non-trivial rule set. Finding all the applicable implicits then reducing them to only one exposes a significant increase in algorithmic complexity. Furthermore, the misuse of implicits can lead to software code that is hard to maintain and contains bugs that are hard to discover and fix.

Thesis I.

I have analyzed how the type system of Scala supports providing succinct and elegant solutions to problems and I have found two areas of possible improvement.

Firstly, I have created a novel approach to type simple recursive functions. I have proved its correctness by defining a new formal language based on typed lambda-calculus and used typing rules to prove the correctness of the theorem. I have implemented this type inference algorithm as an extension to the Scala compiler and tested the implementation on various cases.
Furthermore, I have investigated how implicit resolution affects the performance of the compiler. I have created a specialized compiler plugin to make meaningful measurements and created a test suite to generate synthetic code to test my hypotheses. I have worked out easy-to-follow guidelines for programmers that help reduce the potential errors and the time compilation takes when using implicit.

1.3.2 Effects of Error Handling on Software Complexity

Motivation

Handling exceptional cases is an essential part of any software. It is absolutely required to create robust or self-healing programs. There have been many ways developed to handle these exceptional cases, ranging from global error values and using function return types in structured programs through exception handling to using monadic constructs. An interesting question is how these different approaches affect the most important software metric, complexity. Investigating this has the potential to answer when and how to use different error handling methods.

Thesis II.

I have investigated how different error handling methods affect software complexity. I have extended two standard complexity metrics – McCabe’s cyclomatic complexity and A-V – to the case of exceptions and evaluated how these metrics measure complexity for exceptions. I have compared traditional error handling methods (using ERRNO and return values) to modern exception handling constructs. I have concluded that exception handling helps reduce software complexity.

I have also analyzed monadic error handling comparing it to using exceptions and through examples I have shown that in general, monadic error handling does not increase complexity while it helps reduce it for non-structured programming constructs.

1.3.3 Scale-free Properties of Software Systems

Motivation

It is a generally accepted good practice to design software components to have as low coupling as possible. It is said to help with code reuse, testing and maintainability of software. Several fascinating questions arise around this practice, including if the lowering of coupling has a practical minimum, how evenly distributed it is among components, if it is conserved throughout the lifetime of software and if there are properties shared with other non-artificial networks.
Thesis III.

I have investigated ideal coupling levels in large software systems. I have defined two software metrics and proved they are coupling metrics. I have shown that these metrics would display scale-free properties on the Scala compiler and I have also shown that the scale-free property is kept throughout major versions of codebase of the Scala compiler.

1.3.4 Read-Copy-Update in Scala

Motivation

Computational complexity is an ever increasing property. To answer this, multicore CPUs have gained wide adoption, but software is also required to leverage the available resources. This is the reason why we have seen an rising interest in concurrent and parallel programming. This has led to investigate new approaches in the field, especially ones with higher abstraction levels so programmers can focus on solving the domain problems. One of these higher-level constructs is read-copy-update, but so far we have only seen it implemented in low-level languages, such as C or C++. Could it be used in a multiparadigm language effectively? How would it compare to simpler and more rudimentary locking strategies?

Thesis IV.

I have implemented the read-copy-update locking scheme in Scala providing a high-level API that fits into the standard library of the language. As a validation of the RCU method, I have used the happens-before relation to prove the soundness of the implementation. To analyze the performance characteristics of the approach I have created a concurrent hash map implementation using my RCU API and investigated how it compares to the one in the Java’s standard library. It has comparable performance, so I have concluded that RCU is a viable locking strategy even in high-level language constructs.
Chapter 2

Theoretical Foundations

2.1 Programming Paradigms

Programming paradigms are sets of features that work together as frameworks that help humans think about how problems can be solved. They also serve as a fundamental classification of programming languages. There are two major categories [39]:

- The imperative paradigm where programmers mainly focus on how the underlying machine will execute statements. Programs define control flow and usually change program state – that is, the memory space of the running program.

- The declarative paradigm where instead of describing how the code should execute, the program describes what to execute via the program logic. It mainly omits execution details.

The imperative classification can be broken down into multiple paradigms such as structured, procedural and object-oriented, while the declarative category includes query languages, functional and logic programming.

Other paradigms, orthogonal to the imperative or declarative classification can include, for example concurrent and generative programming. These paradigms can be implemented in languages that can be classified as either imperative or declarative.

2.1.1 Imperative Programming

In the imperative programming style the end goal of the program is reached through changes in its state. These changes are defined as commands the computer directly or
indirectly executes (these are called program statements). In its purest form, imperative programming uses direct assignments to data fields in common data structures that are held by global variables. Since almost every hardware implementation is based on the imperative paradigm, it was the natural first step to develop software. One can think of purely imperative programming languages as simple abstractions over the machine code. The biggest criticism of imperative languages is having direct control of the program flow (with goto statements) and shared mutable state makes it hard to argue about program correctness, especially in enterprise-scale or concurrent software.

**Structured Programming**

Structured programming is a style of imperative programming that tries to solve certain problems of the purely imperative methodology. The direct aim of this style is improving clarity, quality and maintenance time of programs. The tools that enable this include structured control flow (hence the name) in the form of if/then/else and repetition using while and for loops, as well as block structures and subroutines – and restricting or completely abandoning the usage of goto. These concepts are merely abstractions over the program flow but they manage to greatly reduce software errors [131]. Structured programming does not try to answer the issues of data handling.

**Procedural Programming**

Procedural programming is considered a subtype of structured programming. As the name suggests, the paradigm is based upon procedure calls. Procedures (also called subroutines or functions) provide structure for statements that logically belong together; they are the basis for modularity in procedural languages. Modularity is a desired property of software, especially in large, complex ones. Procedures can be called at any point in time by other procedures or the current one itself, creating recursion. Most CPU architectures provide hardware support for procedures through the stack register which is used to keep track of the call stack. This register can either be a dedicated one (for example in x86) or a specific one can be used by convention [133]. Procedures can have inputs (usually called arguments) and outputs (one or more output arguments or return values). This provides a natural scoping for these variables and thus helps reduce the usage of global state. Procedures can also be used to define interfaces between modules (application programming interfaces, APIs). This level of abstraction greatly helps with software development as APIs can be used as the interaction contract between modules, while their implementations can change without modifying the client code [54]. Procedures can also be
organized into libraries that help reusability as they can collect generalized solutions to common problems.

Object-oriented Programming

Object-oriented programming (OOP) is a programming paradigm that is based on objects which can both contain data and have behavior. The first language supporting all features of OOP that we today consider essential was Simula that was published in 1966 [33]. Code reuse and extensibility are supported by an inheritance model; today we can distinguish two major approaches to this, class-based on prototypal. In class-based inheritance models, classes are used to define how an object of that class behaves - they provide a recipe that we can instantiate and use. Inheritance is also present at the class-level as classes create a hierarchy. On the other hand, prototypes dismiss the idea of classes and objects can directly be used to define inheritance relationships; in effect, objects inherit data and behavior from other objects. OOP also inherited other features from its predecessor paradigms such as the small set of built-in primitive types and the support for some form of procedures. Here we will discuss the class-based approach to OOP for two reasons:

- Scala supports the class-based inheritance model.
- Even the most popular language supporting prototype-based inheritance, JavaScript, added support for classes in 2015 [132].

Classes. Classes provide definitions of the data (and its layout) as well as describe the available procedures of an object instance of a class. They also participate as the building block of inheritance. Objects are instantiated for a particular class. In the class-based OOP model, while objects and classes are in a close relationship, they have a very clear distinction. Classes are the blueprint of how an object works, but themselves do not participate at the runtime of the software.

Encapsulation. This concept describes that the data and the procedures manipulating that data should be enclosed together and the internal representation should be separated from the external API to reduce erroneous usage. Hiding the internal representations also helps with changing implementations as the contract of the class can remain unchanged. Encapsulation can be enforced by using access modifiers, such as private, protected or public. Applying these properties to instance variables or methods in a class will change the scope of accessibility, so that only instances of the given class, instances of derived classes or any other object can access them, respectively. There are other language-specific modifiers, such as friend classes in C++ or package private in Java. Instances of friend classes can access each other’s protected and private fields or methods while package private internals can be access by objects of classes belonging to the same package.
Inheritance. Inheriting from existing classes is one of the key concepts of object-oriented programming and it is the foremost approach to code reuse. Programmers can create hierarchies that describe an is-a-type-of relationship: all of the instance fields and methods of the base class will become available on the derived class. Derived classes can override public or protected members to provide a specialized behavior. In most languages, overrides cannot change the types of fields or the signatures of methods as this would lead to potential misuse of features of the base class. Derived classes can add their own fields and methods that can be inherited further. While most languages only support single inheritance (in effect, a class can only extend one other class) there are languages with multiple inheritance (for example C++) and others providing a middle-ground (Scala supporting inheritance from multiple traits). Abstract classes are special classes that cannot be instantiated – they can only participate in type hierarchies. Most languages also support avoiding further inheritance (e.g. by marking classes with final in Java or Scala).

Composition. Classes can contain variables to objects of other classes. This enables them to compose, delegate and extend behavior without creating a closer relationship via inheritance. This way the container class can only depend on the public interface of the contained class so encapsulation is not broken. Many languages and methodologies prefer composition over inheritance since it can provide a more flexible way to architect software. This can be attributed to less advanced type system of the given language [20] and can lead to losing important compile-time checks.

Polymorphism. Object-oriented languages support subtype polymorphism. This describes the fact that a subtype can be safely used as any of its supertypes. This substitutability principle has been formalized by Barbara Liskov [64]. With the help of subtype polymorphism, the caller of a method can remain indifferent to the exact type hierarchy and still use it effectively – this is especially useful while using container objects such as lists. A list can be declared to contain objects of a base class and since we can substitute that with any derived class, those can be contained in there as well. If using strong and static typing, the compiler can check for errors in these cases as well.

Dynamic dispatch. Dynamic dispatch is the process of selecting which implementation of a given method to be called. It is the object’s responsibility to decide and does not depend on the client code. As with the list example above, if we were to call a method on each object in the list, the proper method would be selected at run-time depending on the fact that if a derived class overrode it or not. Methods that can be overridden in derived classes are called virtual methods. In most languages all methods are virtual and the programmer has no way of changing it. Since this type of method lookup potentially requires an additional dereference operation
– hence a performance penalty –, e.g. C++ lets the programmer decide if they want to make a given method virtual.

Open recursion [66]. Methods on objects can call other methods on the same object. There is an implicit variable in scope usually called this or self that represents the current object. Method calls on the this reference use the regular dynamic dispatch approach, meaning that they have a late binding. This process is called open recursion which is in contrast to closed recursion in which the programmer selects the method to call statically.

2.1.2 Declarative Programming

The declarative style of programming – in contrast to imperative – focuses on expressing the logic behind a computation and not the control flow of the program. It is usually described as the difference between the what and the how: programs in declarative languages define what the program must achieve compared languages using imperative paradigms which describe how to achieve it as a sequence of executable statements. With declarative languages, the exact steps to be carried out to complete the program are up to the language or the runtime to define. These languages or runtimes try to eliminate side effects thus resulting in referential transparency [110]. We call a computational step referentially transparent when it can be symbolically replaced with its end value without changing program behavior. More specifically, a function is referentially transparent when given a set of parameter values it will always return the same value without modifying any global state. These simple constraints allow a wide variety of approaches to declarative programming, including query languages, logic and functional programming.

Relational Model

SQL is a domain-specific language designed to manage data in relational databases [25]. It was designed to use with structured data in mind, where different entities in the data set have relationships to other entities. Its theoretical backgrounds are rooted in relational algebra but is extended to be able to define, control and manipulate data on top of querying it. The language consists of the following elements:

- **Queries** are used to retrieve data based on the specified criteria.
- **Expressions** can produce scalar values or tables.
- **Predicates** specify conditions to limit the effects of the data manipulation or retrieval queries.
- **Statements** that have permanent effects on the stored data.
• *Clauses* are constituent components of queries.

SQL has become an ANSI standard in 1986. Since then, many variations and vendor-specific extensions have been implemented in relational database systems. It also has a procedural extension that lets programmers define *stored procedures* that can list statements that are executed by the RDBMS. SQL as a query language only describes the intents of the programmer and the exact steps to be executed are defined by the database engine. In most cases they produce a high-level *query plan* [61] that can be transformed and optimized. The query plan can be investigated by programmers to improve their data definitions (e.g. by introducing new indexes). The plan is then executed by the RDBMS and the results are returned. Although SQL has elements that modify the data set, we can still consider it referentially transparent as the queries and the data are separated. The transformation statements themselves are referentially transparent as well if we consider the underlying data as an input to them.

**Logic Programming**

Logic programming is based on formal logic [65]. Programs in logic languages are defined as a set of logic sentences that express known facts and deduction rules of the problem domain. These are in the form of Horn-clauses [122]. As a declarative paradigm, it does not allow the definition of the theorem-solving strategy since that is up to the language or the runtime to decide. In the simple cases, an and-or-tree is built based on the deduction goal and the sentences in the logic program. Solving the problem can be expressed as a search in this tree. This algorithm can be improved by parallel search, intelligent backtracking or best-first approaches. The runtimes can decide which strategy to use based on the types of the logic expressions. Since Horn-clauses can be represented as procedural programs [116], one could argue that logic programming is in fact, imperative in nature, and it is simply an abstraction over some procedures.

The most widely used logic programming language is Prolog [121]. Prolog was introduced in 1972 as the first of its kind. Its resolution strategy is *Selective Linear Definite* (SLD) clause resolution. It is based on resolution in first-order logic and is proven to be sound and refutation complete for Horn clauses [111]. After defining the domain-specific rules and facts, running a Prolog program is fairly simple: the user needs to define a single goal – called the query – that the runtime will try to prove through SLD. If the negated query can be refuted, the resolution is said to have succeeded. Prolog is still widely used in some areas of computer science, such as artificial intelligence, linguistics or voice control systems.
THEORETICAL FOUNDATIONS

Functional Programming

Functional programming uses mathematical functions as the building blocks of computations. Mathematical functions are relations that associate every element in a set to exactly one in another set. Since these functions cannot have shared, mutable state, purely functional languages also completely avoid it. Naturally, functions and thus full programs written in such languages are referentially transparent. Without complex shared mutable state it can be an easier task to understand a program or argue about its correctness [53]. This is especially the case when we consider highly concurrent, large-scale software systems. Originally based on lambda calculus, functional programming languages can be considered improvements to this formal system. The functional programming paradigm has its own core features that we will discuss in the following Section.

Pure functions do not have side effects, in the sense that evaluating them will not change the memory outside of local to the function or produce any IO operations. These constraints can be leveraged to help with understanding their usage and also with optimizing software:

- If the result is not used, the function can be removed without affecting other functions.
- No two pure functions can have effects on each other thus their evaluation order can be changed or they can even be evaluated concurrently.
- Evaluation strategies of pure functions can combine or re-order them.
- Calling a pure function with a given set of input values will always generate the same output. This means we can substitute its evaluation with the end result. This strategy is called memoization.

Higher-order functions are functions that either take other functions as their arguments or return one. They require methods or functions to be first class citizens in the programming language, so they can be treated like any other value: they can be referenced and passed around. Many non-purely functional languages support this via function references or pointers.

Currying a function means applying it to one argument and then returning a new function with the remaining argument list. This abstraction proves to be useful in functional interfaces and managing how their arguments are passed to them. Currying sometimes referred to as partial application.

Recursion is one of the key concepts in functional programming. During recursion, a function will call itself until it reaches the base case. Recursion can be used to the same effect as iteration in imperative languages. Most computer architectures
provide a function call stack with a limited size, that recursive functions can fill up quickly. To eliminate such issues, *tail recursion* can be used. When the last call in a recursive function is the recursive call, it is called tail recursion. It can be implemented without adding a new stack frame to the call stack, eliminating the potential issue of filling it.

*Type systems* of functional languages are almost exclusively based on the typed lambda calculus and the Hindley-Milner type inference system [49, 74]. Using such type systems can find errors in the program at compile time, reducing the risk of runtime errors. The ultimate goal is to reject all invalid programs but this is not always possible as not all types can be deducted prior to runtime. User-defined types in functional languages are *algebraic data types* of either product types or sum types. A product of types is a new compounded type in some structure. The simplest representative example is the two-tuple of types $A$ and $B$ where the new type is the (ordered) pair of the underlying types. Sum types, on the other hand can hold a value of several, but fixed set of types. A basic example of a sum type is a linked list. A linked list can either be an empty list (usually noted as *Nil*) or a combination of an element and another list (referred to by $x\, xs$). Advancements of this type system usually involve generalized algebraic data types [113] and dependent types [87].

*Lazy evaluation* lets programmers define functions whose arguments are only evaluated when they are used during the evaluation of the function itself. This method helps with separation of concerns [53] and can be used to create and transform data streams with large number of elements efficiently.

Functional programming has its drawbacks as well. There are several problems that are natural to be solved by maintaining state, for example using IO. Since this is not directly supported in pure functional languages, there had to be introduced various abstractions that let the programmer emulate such state. One of the possible implementations uses *monads* [127] that represent a chain of computations that can simulate state transformations in a purely functional way. Another weak point of functional programming languages comes from the underlying hardware. There are several algorithms and data structures that are naturally simple to represent with state using the current hardware platforms. Using these directly in high-level programming paradigms can be problematic or not even possible but omitting them completely can reduce performance of programs.
2.1.3 Other Programming Paradigms

There exist computer programming paradigms that are agnostic to the fact that they are used in declarative or imperative languages. These can still be considered independent paradigms as they expose a complete methodology as to how we think and solve problems.

**Concurrent Programming**

A concurrent programming model means that computations are alive in overlapping time periods. It is related to, but separate from parallel computing. Running computations parallelly means they are running physically at the same time, while concurrency is related only to the lifetime of these computations. Concurrent programs can be run on a single CPU core while true parallelism is not possible in such a setup. Natural consequences of parallel programming include improved throughput and better utilization of computer resources. There are many problems that a simple *divide and conquer* [60] approach can speed up, including image processing or solving numerical and financial problems.

Concurrent programming can be achieved using many abstractions and computational models. The most basic and widely used one is based on concurrently running threads which represent different computations of a program. These threads can communicate with each other through shared memory. When they do so, careful considerations need to be implemented so that operations modifying the memory are executed *atomically*: one needs to guarantee that they complete from the operation’s beginning to the end without other threads reading or writing the same memory. This leaves the memory in a correct state at all times. Atomicity is achieved through *mutual exclusion* which uses mutexes, locks and other synchronization constructs between threads. These high-level concepts have hardware support on most modern CPU architectures to aid with performance.

Another popular approach is the actor model [47]. In this methodology, computations are represented by independent actors that share no state with other actors in the system. They communicate with each other through immutable messages. The connections between the actors can either be static or dynamic, depending on the implementation framework and needs. Since actors lack a shared state, it becomes easier to argue about the correctness of the system. They can be implemented in the threading model with shared message queues or they can be separate processes potentially running on different physical computers.

It is easy to draw parallels between imperative and functional programming and the threaded and actor-based models. They also share some of the same advantages – such as easier comprehension in case of functional programming and the actor
model – and deficiencies – including potentially reduced performance compared to shared memory and imperative paradigms.

Generative Programming

Generative programming is a paradigm where algorithms are implemented without fixed types that are specified later. When a new type needs to be substituted in the algorithm, a new version is instantiated. This approach greatly helps reduce code duplication since it allows writing the same algorithm once, but can be used with many specific types. Most languages support generative programming through language-level generics (or sometimes called parametric polymorphism). These languages include C++, Java, Haskell and Scala. There are many possible implementations available including

- Template instantiation in C++ where the compiler will generate new code for each template use it needs. This can be leveraged to run programs at compile time – called template metaprogramming – and the supported computational model is Turing complete [124].

- For JVM-based languages (e.g. Java or Scala), generics are only usable with reference types, but only one version is generated for all of them. Since all references can be treated the same from the perspective of the generic code, there is no need to have special versions. This method is called type erasure and it exposes several problems at runtime especially with reflection. The Scala compiler can work around this problem by generating additional type information for generics that can handle all features of its type system.

- On the .NET CLR, generics are implemented via reification: when the runtime meets a new usage of some generic code, it compiles a new version specific to the new type. This allows all reflection features to remain available for generics and it enables generic usage with value types as well.

Most programming languages allow generic code to have constraints on the type parameters. This can be used to only allow types to be used with the generic code that meet its core assumptions and requirements. Usually this means deriving from or extending a specific type. There are also type systems (including the one in Scala) that let generics define their variance: they can be invariant, covariant or contravariant. These control how generic types relate to the type hierarchy of its parameters: they can have no relations; if two type parameters are subtypes of each other, their generic version will also be; or the subtype relation will be flipped respectively. This feature is important to keep subtype polymorphism in line with generics.
2.2 Multiparadigm Programming

As we have discussed in the previous Section, there are programming paradigms that have a close relationship with each other (structured, procedural and object-oriented) and there are ones that can be implemented alongside other paradigms (for example generative or concurrent programming). Henceforth we can safely say that almost all modern programming languages support more than one paradigm. It is easy to see that, for example, all object-oriented languages support (and require to use, to some extent) procedural programming. A similar statement can be made about declarative languages as their execution model is not defined by the program code, hence they can very well be ran concurrently. Coplien also argues that most large-scale software products need to have multiple paradigms involved in both the design and implementation phases to successfully serve their purpose [31].

The need for languages supporting more than one programming style comes from the variety of the problems software developers need to solve. One could argue that mixing multiple languages in a software product can be the answer, but the complexity of inter-language communication and the differences between various levels of tooling available for languages makes this approach less feasible. Programmers need to be able to select the right construct to solve the problem at hand without doing a complete context switch. This is enabled by programming languages that support multiple paradigms at least to some extent.

We argue though that the aforementioned mixtures of paradigms do not truly result in multiparadigm languages. The reason for this is that the level of support for these differing paradigms is not the same. While one can pass around function pointers in C++ and create lambdas or const expressions, it still lacks major properties of functional languages (for example pattern matching or compile-time checked referential transparency). Furthermore, to use functional constructs, the programmer needs to go the extra mile and apply additional caution to not break other abstractions.

True multiparadigm languages like Scala, on the other hand, have been designed from the ground up with blended styles in mind. The mixture of programming methodologies and abstractions do not break down because of each other, rather they build on top of the strengths of each other.
2.3 Software Metrics

Software metrics are standard definitions of a methodology to measure the degree of some property a software system holds. It is a thoroughly investigated research area with a long history: the first dedicated book on the topic has been published in 1976 [42]. Although they tend to be well-defined, software metrics do not necessarily share properties of the mathematical definition of the identical name. As the broad definition suggests, there are many software metrics defined for measuring various properties of software. These measurements can be divided into numerous categories, such as size, complexity, comprehensibility, cohesion, coupling, maintainability, testability and many more [78]. We will focus on the categories here that are related to our later investigations: complexity and coupling.

2.3.1 Complexity Metrics

Software complexity metrics have originated in industrial experiences [70]. Later Weyuker [129] created a framework so further research can build on solid theoretical foundations. This framework consisted of nine properties to which – based on her recommendation – every sensible complexity metric should adhere. Let us use the following notations:

- \( S \): the set of all programs, elements are \( s_1, s_2, s, s' \), etc.
- \( m : S \rightarrow A \): the metrics function. \( A \) can be any set, most metrics use \( \mathbb{N}, \mathbb{Z} \) or \( \mathbb{R} \).
- \( \oplus : S \times S \rightarrow S \): the extension function (it extends a program with another one).
- \( s_1 + s_2 \): simple textual concatenation of \( s_1 \) and \( s_2 \).

With these, the nine properties can be formulated as follows:

1. \( \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \implies m(s_1) \neq m(s_2) \)
   
   This is a very simple property describing the need that the metric should be able to distinguish different programs.

2. 
   
   Let \( c \in A, c \geq 0, S_1 \subset S \forall s \in S_1 : m(s) = c \implies |S_1| \leq \infty \)
   
   Here we require that for any given complexity only a finite number of programs should exist.
3. \[ \exists s_1 \in S \land s_2 \in S : m(s_1) = m(s_2) \]

There need to be two non-equivalent programs with the same complexity.

4. \[ \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \implies m(s_1) = m(s_2) \]

Even if two programs are equivalent based on their functionality, their complexity does not necessarily need to be.

5. Let \( s \in S, s_1 \in Ss' = \oplus(s, s_1) \implies m(s) \leq m(s') \)

Here we say that combining two parts of the system should not decrease complexity.

6. \[ \exists s_1 \in S \land s_2 \in S \land s' \in S : m(s_1) = m(s_2) \land \oplus(s_1, s') = \oplus(s_2, s') \implies m(s_1) \neq m(s_2) \]

In this property we state that the relationship of two components with a third one can be different even if they have the same complexity.

7. \[ \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \text{ (only in the order of statements)} \implies m(s_1) \neq m(s_2) \]

In this property we describe that a complexity measure is sensitive to the permutations of the statements.

8. \[ \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \text{ (only in the names of variables)} \implies m(s_1) = m(s_2) \]

With this we describe that naming in programs should not affect complexity.

9. Let \( s \in S, s_1 \in S \land s_2 \in S : s = s_1 + s_2 \implies m(s_1) \leq m(s) \land m(s_2) \leq m(s) \)

Combining two programs should not decrease the complexity of the parts.

As one can see, these rules describe simple properties that would allow the adhering metrics to properly measure and compare programs. Furthermore, they allow
us to measure how changes in the software code would result in changes in its complexity. This is an important factor as increasing complexity necessarily increases development and maintenance costs.

Even though Weyuker’s properties help eliminate trivially wrong metrics, adherence to them does not guarantee the opposite: it has been shown that a metric can satisfy all nine rules without providing any useful information [26]. Weyuker’s answer to this was that the probable cause for such issues is that we do not know what to expect from a software metric and how exactly it would describe the code [96]. This relates to the biggest criticisms of software metrics: providing useful definitions still relies heavily on heuristics and we still need to tweak metrics on a case-by-case basis [57].

2.3.2 Coupling Metrics

Coupling metrics of software describe the independence of different entities. Entities can vary based on the language and the paradigm; they can be modules, packages, classes or functions, but in most cases the entities in question usually are of the same kind for a given metric (e.g. the metric measures symbolic references between modules). The connections between the investigated entities can also be numerous (e.g. calls between functions). Putting the two concepts together we arrive at a graph describing the coupling. As such, we can either have a directed or non-directed graph and we can apply weights to the connections. Coupling metrics focus on the connections and not the nodes.

There are different kinds of coupling metrics in the literature – including, but not limited to conceptual coupling [98], coupling between objects [75], and SAS or NAS [46]. Similarly to complexity metrics, there are a set of properties that are required to be met to call a metric a coupling metric:

1. Null value. The coupling of an entity is zero, if there are no connections to other entities. (If the entity is denoted by $e$, then let $\text{Outer}_o(e)$ denote the number of connections towards other entities, and $\text{Outer}_i(e)$ the number of connections towards $e$. The distinction between incoming and outgoing connections is not necessary.)

3. Monotonicity. Let $\text{Outer}(e) := \text{Outer}_i(e) + \text{Outer}_o(e)$ and $\text{Coupling}(e)$ be the function that assigns the result of the coupling metric to entity $e$. In this case, if we add a new inter-entity relationship to $e$ (let us denote this modified module
by \( e' \), its coupling will not decrease:

\[
Outer(e) \leq Outer(e') \Rightarrow Coupling(e) \leq Coupling(e')
\]

4. **Merging of entities.** If two entities are merged together (e.g. all functions from one module are moved to the other) then the coupling of the merged entity is not greater than the sum of the individual entities.

5. **Disjoint entity additivity.** If two entities, \( e \) and \( e' \) are not connected (e.g. there exists no function call between the two modules) and there is no entity \( e'' \) such that it connects to both \( e \) and \( e' \), then the sum of the coupling of the individual entities equals to the coupling of the merged entity.

Coupling metrics can be used to discover entities in software systems breaking good development practices, e.g. modules with multiple responsibilities or show hard-to-test and hard-to-refactor modules. Furthermore, they can pin-point issues with module reusability or increased compile times due to overblown inter-dependencies of modules.
Chapter 3

Improvements to The Type System of Scala

3.1 Motivation

As we discuss in Section A.1, Scala is a multiparadigm language providing both object-oriented and functional programming language elements. As a consequence, the language syntax needs to reflect both paradigms that results in a high level of expressiveness. The design of Scala is targeting the extensibility, safety and flexibility of the language [125]. One of the key aspects of this is improving the readability of the source code; including the ability to avoid unnecessary boilerplate, repetitive code elements. Terseness is important not only to make software code cleaner, but also to accentuate key parts of the solution expressed by the existing elements. Furthermore, the more automatically computed information the compiler can provide, the less possibly erroneous code snippets the developer writes. Scala’s terseness can be attributed to several language features, such as its advanced type inference capabilities and the ability to omit certain code elements that are only defined implicitly.

3.1.1 Shortcomings of The Type Inference Algorithm

Type inference features a wide range of elements from deducing variable types based on their initializations to inferring type parameters of generic functions based on call-side information. Function return types are inferred for most trivial cases as well: the type of the lastly evaluated expression provides the return type of the function. In cases when multiple return statements are present the least upper bound (LUB) type of these will be used.
Since the subtyping relation \((\prec)\) is reflexive \((A \prec A)\), transitive \((A \prec B \land B \prec C \Rightarrow A \prec C)\) and antisymmetric \((A \prec B \land B \prec A \Rightarrow A = B)\), it defines a \textit{partial ordering} on the set of the types and thus this set has the \textit{least upper bound} property. Scala supports all three kinds of variance, so it is up to the programmer to define how the type arguments affect the subtyping in case of generics.

However, if any of the before-mentioned statements include a reference to the containing function, this algorithm cannot provide a meaningful result. This causes recursive functions without explicitly provided return types (such as on Listing 1) to fail the compilation process.

```
1 def factorial(n: Int) = n match {
2   case 0 => 1
3   case _ => n * factorial(n-1)
4 }
```

Listing 1: A recursive function with no defined return type.

However, for any developer it is obvious that the \texttt{factorial} function will return an \texttt{Int}. Such similar simple recursive functions (see formal definition on Def. 3.3.1) occur frequently in most functional codebases. Not inferring their return type is both against the original design philosophies of Scala and puts an unnecessary burden on the programmer even in these simple cases.

```
abstract class Base
1 class Derived1 extends Base
2 class Derived2 extends Base
3 class Derived3 extends Base
4
5 def lousyType(d: Base, m: Double) = d match {
6   case _. Derived1 if m > 3 => new Derived3
7   case x: Derived2 if m < 2 => lousyType(x, m)
8 }
```

Listing 2: A recursive function with return types of a hierarchy.

Consider the example on Listing 2 that does not compile under current Scala type inference rules. In such situations the programmer may choose an unnecessarily wide type, such as the top type, \texttt{Any}, corrupting the highly praised type system of Scala.

The type inference algorithm of Scala is far from complete. Not only it does not support function recursion – be it a simple recursive function or a recursive chain – it omits inferring recursive type declarations as well. This can be surprising and frustrating to anyone writing Scala code, furthermore it can lead to non-trivial issues in one’s code and later, when it becomes a software product.
In the following we show a few examples where the lack of type inference on recursive functions may lead to possible runtime application errors.

```scala
1 def map[C, A <: C, B <: C](y: Seq[A], f: A => C) /*: Seq[C]*/ = {
2   y match {
3     case Nil => Seq[B]()
4     case x :: xs => map(xs, f) :+ f(x)
5   }
6 }
```

Listing 3: A recursive map implementation with explicit type annotation.

In our first example we start off with a higher-order function, `map` with our own interpretation. In the version that can be seen on Listing 3, we leverage generic types as well as functions as first-class entities in Scala. We would always like to get the minimum type of the collection from this function, hence we are providing information on the relationships of the types. Unfortunately the Scala compiler is not much of a help here: it will throw an error when we try to call `map` with the remainder list. This becomes even more annoying when we provide an incorrect return type: the compiler will be able to recognize the error at the place of concatenating the computed element to the list. One can spend minutes on trying to find the type that makes the constraints in the type system satisfied by recompiling several times, but it would be much more convenient if the compiler were able to find it for us at the very beginning.

```scala
1 class Level1
2   case object Class1Level1 extends Level1
3   case object Class2Level1 extends Level1
4 class Level2 extends Level1
5   case object Class1Level2 extends Level1
6   case object Class2Level2 extends Level1
```

Listing 4: A multi-level class hierarchy.

Let us consider another recursive method that computes its result that has one of the types of a multi-level class hierarchy, as shown on Listing 4. An example method using these types can be found on Listing 5. The Scala compiler will fail to infer the correct return type as `Level1`, and will require the developer to define it for the function explicitly. Determining `Level1` as the return type is trivial in this case as we have listed all the types involved near the function definition in one place, but recognizing it when, for example, class definitions are scattered across a fairly large and complicated framework can be challenging and time consuming even for seasoned developers.
def deepRec(n: Int): Level1 = {
  if (n == 0) {
    Class1Level2
  }
  else if (n == 1 || n == 2 || n == 3) {
    n match {
      case 3 => Class2Level2
      case _ => {
        if (n != 1) {
          deepRec(n - 1)
        } else {
          Class2Level1
        }
      }
    }
  } else if (n == 4) {
    Class1Level2
  } else {
    Class1Level1
  }
}

Listing 5: A recursive function using type hierarchy on Listing 4. with explicit type annotations.

Unfortunately, there is a pretty easy shortcut to make compile errors disappear in this case: define the return type as Any, making the type system and the compiler temporarily happy. As we all know, marking objects by the widest type is simply neglecting the type system, thus we are not using one of the main services offered by the Scala compiler. In this Chapter we will present a novel approach to type simple recursive functions that does not require calculating a fixed point and we will formally prove its correctness.

3.1.2 Performance and Maintainability Concerns of Implicit Usage

Besides type inference, Scala also supports omitting explicit usage of certain code elements that can be deducted from the context. In programming terminology, the term ‘implicits’ covers a wide area, and they have an important role in making the code more succinct, easier to understand and maintain. Implicits can take several forms, including implicit types, implicit converters or implicit arguments or argument lists. All of these are supported in Scala through the implicit keyword. As Scala is a block-structured programming language with strict scoping rules, implicits naturally need to adhere to these. To let the compiler automatically apply any implicits, they need to be in scope; they need to be either defined in the current scope
or be imported into it. They are resolved when the compiler encounters a typing error. If such an error happens the compiler will not simply abort the compilation after reporting to the output, but tries to find an entity marked with the implicit keyword that would resolve the error. We discuss implicits at a high-level in Section A.1.8, but here we will describe them in more detail.

Implicit Classes

*Implicit classes* are classes that have their primary constructor available for implicit conversions when the class is in scope [105]. They are a form of implicit converters, as the types of the parameters to their primary constructor are converted to the implicit type in question. Usually they are used to enrich APIs. One of the most typically representative examples is `RichString` (Listing 6).

```
1 implicit class RichString(val self: String) {
2   def apply(n: Int): Char
3   def capitalize(): String
4   ...
5 }
```

Listing 6: Implicit class example.

There are some restrictions that apply to implicit classes:

- They must be defined inside another `trait`, `class` or `object`.
- They may only take one non-implicit argument in their constructor.
- There cannot exist any other entity with the same name in scope.

These restrictions are in place to make resolution unambiguous.

Implicit Argument Lists

In many functional languages, methods can take multiple argument lists. In Scala, an argument list can be marked as implicit [85], meaning that the compiler will try to find values for the parameters implicitly, without explicitly expanding the given argument list; if the developer wishes so, they can still have the values explicitly defined. Implicit argument lists are often used in DSLs and high-level APIs to let developers omit passing a common value in a code block repeatedly (be this block a class or a method). A good example from the standard library can be seen on Listing 7 where we show a snippet of Scala’s `Future` implementation, that needs an `ExecutionContext` to be passed in.

The usual usage pattern for `Futures` would potentially require passing in the same `ExecutionContext` each time a `Future` is used. Instead, the developer can
Implicit Methods

As Listing 8 shows, implicit methods take a simple form of definition [85].

```scala
implicit def int2Range(i: Int): Range = {...
```

Listing 8: Implicit method example.

The rules for their application is a little more involved. The compiler will try to apply a method of type \( S \Rightarrow T \) marked as implicit when

- An expression \( e \) is of type \( S \) and \( S \) does not comply to the expression’s expected type \( T \).
- A selection \( e.m \) with \( e \) of type \( S \) if the selector \( m \) does not denote a member of \( S \).

In simpler terms, if an expression does not have the correct type at its place of use, or if a method or field is accessed on an instance of a type without the given name, the compiler will look for a properly typed implicit method that would resolve the type error. As one can see, implicit methods can be used as type converters without bloating the code with explicit method applications.

Based on the features, rules and restrictions we have discussed in this Section, one can easily see that resolving implicits imposes a non-trivial algorithmic complexity. The Scala compiler handles many other features, and on top of those, resolving implicits adds many additional steps. This has a huge effect on compilation performance [24] that can turn working with Scala a frustrating experience for developers. In this Chapter we investigate the root causes of this performance degradation and suggest methods to improve the compilation performance.

**Behind the motivation: a real-world project**

The original motivation of our investigations into the compiler performance had come from a large industrial partner. This firm has the largest production Scala
codebase at 3.6 million lines. The project plays a significant role in the firm’s strategy and has the focus of higher management. The global team consists of around 400 developers who work daily on the codetree - a monorepo containing all the related code. This software product has two main parts: a general computation platform – that consists of a compute graph, a bitemporal object store, a distribution engine and a UI framework – and implementations of many business functions that use this platform.

It is easy to see that the sheer size of the codebase makes it discover unique problems that other projects simply never meet and can have a significant effect on the effectiveness of large developer group. These include code organizational challenges, change management, configuring development environments and naturally, the performance of the compiler. Even a small improvement in any of these areas can lead to notable increases in developer productivity and reduce the cost of the project.

3.2 Related Works

3.2.1 Typing in Scala

As Pierce writes in his book, “a type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute” [93]. That is, the type system of a language determines how easy it is to use it to write bug-free software. The more the type system covers, the easier it is to write correct programs using it. Functional and object-oriented programming put a lot of emphasis on their type systems and Scala is no exception. It has a strong, static type system [85]. The following advanced features are supported on top of more common object-oriented features:

- **Parametric polymorphism.** Generic programming: when algorithms are implemented without concrete types.

- **Type inference.** Certain type annotations do not need to be present explicitly, the compiler can deduct them based on the context. In Scala, this only works for local types.

- **Existential quantification.** When the concrete type symbol can be omitted, only its existence matters.

- **Variance.** In parametric types, subtyping rules can be applied based on the sub or super types of the type parameter.
• **Structural types.** Structural types can be used to describe what form a type needs to support instead of referring to it by a specific symbol. This is also known as “duck typing”.

• **Type views.** Type view definitions can be used to define how a user of a given type can substitute that type, requiring an implicit converter to be available.

As one can expect, these increase the complexity of the language as well as its implementation. The current Scala compiler consists of many phases (based on the version, between 13 and 30) [101]. One of these phases is the typer, which handles checking type annotations as well as runs the type inference algorithms. This includes *implicit resolution* that further increases both the complexity and the required resources needed to compile Scala code.

### 3.2.2 Success Typing

Most statically typed languages such as C# or Java require explicit type declarations. Other statically typed languages like Haskell or ML use static type inference to calculate types for functions. Some unification-based type inference [56] can be used to calculate types for functions in these languages. Another unification-based type inference is the Hindley-Milner method, supposing that the return type of a function has a well defined type. Scala on the other hand is less restrictive on return types, branching expressions like the `match` construct allow that the return values on different branches have different types [32]; e.g. a function can return either an integer or a string. In this case the return type of the function will be the LUB-type of integer and string which is the top type, Any.

Dynamically typed languages like Erlang [126] also allow to return values of different types. These kinds of *polymorphic* return types are extensively used in Erlang. Since types are not first-class citizens, external tools were developed to check for discrepancies in software [63], incorporating a type system called *success typing* [62]. The major difference between success typing and other type inference algorithms is that the aim of success typing is not to prove the type correctness of the program, but rather to discover cases where there would most certainly be a type error at runtime. It uses least upper bound types to enable the constraint solving algorithm to reach a fixed point.

Success typing uses union types to express coupling between types that are not in subtype relation. It is very useful for languages like Erlang where types are not an integral part of the language. It has a major drawback though when both the input parameter and the return type of a function are union types. The connection between the input and output is not expressed by the inferred type, and that decreases the
number of discoverable errors. A possible improvement can be the use of conditional
types similarly to the work of Aiken et al. [15]. This soft-type system (also using
union types) includes conditional types where the constraints between type variables
are built into the type. This type is more accurate in the above sense of finding
discrepancies, but the size and complexity of inferred types make it incomprehensible
for humans. If the human-readable criterion is ignored then the precision of inferred
types can be increased without the need to calculate a fixed point [88]. These types
are also very complex but can be used for specific tasks, e.g. automatic test data
generation.

Success typing inspired us to type recursive functions. Since Scala is statically
typed and types are inserted into the AST, using union types is not suitable for our
needs. Unions would introduce new types to our program, that we do not intend to
do since it would be hidden to the programmer and might cause unforeseen errors.
Instead, we use the type hierarchy already present in the language. Scala already has
a solid type system for nested classes, abstract types, path dependent types, etc. [32,
87]. Since type inferring is solidly working in Scala, we do not want to replace or
improve these theories.

3.2.3 Type Inference in Modern C++

C++ is a strongly typed programming language in the sense, that the type of every
(sub)expression is determined at compilation time. However, templates use duck-
typing, meaning that in effect, no constrained generics exist in current C++. There
are plans to improve the template mechanism with constraints.

Earlier C++ codebases were known for notoriously long type notations. To un-
burden software developers and make source code more readable, the C++11 stan-
dard introduced the auto keyword as a placeholder for types [120]. Its primary
usage is to avoid needlessly verbose type declarations, for example ones that are
used along STL algorithms (see Listing 9).

```cpp
1 typedef std::vector<T>::iterator i = v.begin();
```

Listing 9: Definition of an STL Vector iterator in C++03.

This can be replaced by usage of keyword auto. The type of the i variable will
be inferred from the initialization expression v.begin() (Listing 10).

```cpp
1 auto i = v.begin()
```

Listing 10: Definition of the same iterator from Listing 9 in C++11.
The keyword `auto` can be used to replace the return type for functions but only with a new trailing type syntax that was introduced in C++11, as shown on Listing 11.

```
template <typename T, typename S>
auto max(T a, S b) -> decltype(a > b ? a : b) // C++11
{
    if (a > b)
        return a;
    else
        return b;
}
```

Listing 11: Using the `auto` keyword for function return type.

Notice that this usage of `auto` syntax does not imply type inference, the return type is explicitly expressed in the trailing syntax. The role of `auto` here is only a placeholder: since the language elements used in the trailing syntax (a and b parameters in the `decltype` expression are not in scope before the function name).

Since C++14, however, developers can use automatic type inference for function return types in the most simple cases [73] (Listing 12).

```
auto f(); // return type is unknown
auto f() // return type is int
{
    return 42;
}
auto f(); // redeclaration
int f(); // error, declares a different function
```

Listing 12: Using the `auto` keyword to infer function return types in C++14.

A function with `auto` return type can have multiple return statements. However, there is a strict restriction here, that each return statement should return the same single type, otherwise the compiler reports an error. That is different to Scala where in case of multiple return statements the return type is inferred as the least upper bound of the return types.

Furthermore, recursion is allowed by the C++14 inference rules in a very restricted way. The recursive return branch should be preceded by at least one non-recursive return, from which the return type of the function is inferred. Subsequent return statements are checked against this type. Therefore the code on Listing 13 will be accepted by the compiler while the variant where we change the order of the recursive and non-recursive branches will be rejected (Listing 14).

We believe that these rules are unnecessarily restrictive and can be relaxed without compromising compile-time efficiency.
auto fib(int n)
{
    if ( 0 == n )
        return 1;
    else
        return n*fib(n-1);
}

Listing 13: Using the auto keyword to infer return types of recursive functions in C++14.

auto fib(int n)
{
    if ( n > 0 )
        return n*fib(n-1);
    else
        return 1;
}

Listing 14: A non-compiling recursive function due to wrong branch order.

3.2.4 Implicit Resolution

Implicit resolution happens during the typer phase. There is a well-defined set of rules the compiler uses to look for applicable candidates \[85\], then selecting one of these candidates. These rules can be summarized as follows.

- **Marking rule.** Only entities marked as implicit can be considered.

- **Scope rule.** An inserted implicit conversion must be in scope as a single identifier, or be associated with the source or target type of the conversion.

- **One-at-a-time rule.** Only one implicit is applied by the compiler.

- **Explicits first.** This seem trivial, but if a value is provided explicitly, it must be used – the compiler will not try to apply implicits if typing succeeds.

- **Occasion rule.** Implicits are only used in three places: conversions to an expected type, conversions of the receiver of a selection and with implicit parameters.

- **Naming rule.** The names of implicits do not matter.

The actual implementation of implicit resolution \[102\] fairly closely follows the rules listed above. When the typer fails to type an AST, it will start the implicit resolution process. In this, there is a set of possible candidates that are marked as implicit. This set is then reduced in a loop until only one implicit remains – the one
that fixes the type error – or the resolution fails, which means that the typing error is valid and the compiler needs to abort.

3.3 Theoretical Resolution

3.3.1 The $\lambda_s$ Language

We propose a small language and the corresponding calculus to demonstrate the theoretical soundness of our approach to type simple recursive functions in Scala. Let us call the language $\lambda_s$ and be defined on Figure 1. This small language is not intended to be either generic-purpose or a full representation of Scala, rather to be the minimal language that can help us describe our proposed method formally.

\[
\begin{align*}
e & ::= x \mid c(e_1, \ldots, e_n) \mid e_1 e_2 \mid f \\
& \quad \text{let } x = e_1 \text{ in } e_2 \\
& \quad \text{letrec } x = f \text{ in } e \\
& \quad \text{case } e \text{ of } \\
& \quad \quad (p_1 \text{ if } g_1 \Rightarrow b_1); \\
& \quad \quad \ldots; \\
& \quad \quad (p_n \text{ if } g_n \Rightarrow b_n) \\
& \quad \text{end} \\
f & ::= \lambda(x) \Rightarrow e \\
p & ::= x \mid c(p_1, \ldots, p_n) \\
g & ::= g_1 \text{ and } g_2 \mid g_1 \text{ or } g_2 \mid x_1 = x_2 \mid \text{true} \mid e \ x
\end{align*}
\]

Figure 1: The $\lambda_s$ language.

The language contains variables ($x$) that are immutable. Data constructors ($c$) can be used to construct any kind of data, including constants, objects, etc. $\lambda_s$ contains only single-argument function application. We assume that all Scala functions with at least one parameter can be curried, that is, they can be transformed to a function with multiple parameter lists containing only one parameter. It contains the standard polymorphic let expression. The recursive let expression has only one function component ($x = f$) since here we would like to focus on only self-recursive functions. We define a branching expression (case). In the head of the case expression, $e$ is matched against the patterns ($p$) sequentially. The first matched pattern will invoke the evaluation of the body ($b$) for the pattern. If no patterns match, then an exception is raised. Each branch has a pattern and a guard. A pattern can be a variable or a construct of patterns. Guards, that are always present in the syntax,
can be type checks or other value checks. Using \texttt{true} as a guard, we can express the case when we actually do not need any guards.

Our main focus will be on combining recursive \texttt{let} and \texttt{case} expressions. Recursive functions can be typed if they have a branch that is not recursive. Having this branch fulfills the termination criteria. If a function does not contain any terminating branches, then the function is divergent, and it cannot be typed.

\[
E ::= \texttt{letrec } x = \lambda(a) \Rightarrow \
\texttt{case } a \texttt{ of} \\
(p_1 \texttt{ if } g_1 \Rightarrow b_1); \\
\ldots; \\
(p_n \texttt{ if } g_n \Rightarrow b_n) \
\texttt{end in } e
\]

Figure 2: The syntax of simple recursive functions.

\textbf{Definition 3.3.1} (Simple recursive function). The expression \(E\) on Figure 2 is considered a \textit{simple recursive function}, iff there exists \(i \in [1..n]\) that \(b_i\) symbolically contains \(x\) and there exists \(j \in [1..n]\) that \(b_j\) does not contain \(x\).

Simple recursive functions do exist and provide an abstract pattern over recursive functions that are not a in recursive call chain.

\section*{3.3.2 Derivation Rules}

We provide type derivation rules for the syntactic constructs of \(\lambda_S\). We assume that the type of objects, member functions and other language constructs not covered here can be computed.

We present type derivation rules (Figure 3) in the following form of statements: \(\Gamma \vdash e : \tau\), read as “supposing \(\Gamma\) the type of the expression \(e\) is \(\tau\).” \(\Gamma\) is the context of mappings from variables to types. The \(\cup\) operator is used to denote that a particular mapping is present in the context.

The derivation rules describe a standard way to type our language. A variable can be typed (RULE (VAR)), if its type is present in the variable context. The type of a data constructor (RULE (CONS)) is composed of the types of the components. A type is considered a subtype of another type (RULE(SUB)) if an expression of the subtype can also be typed to the wider type. A function application can be typed if the argument expression \(e_2\) is a subtype of the parameter type \(\tau_1\) of arrow type. In our case the arrow type is calculated via the existing type inferring algorithm of Scala. The type of a function (RULE (FUN)) and the let (RULE (LET)) expression
\[ \Gamma \cup x : \tau \vdash x : \tau \quad \text{(VAR)} \]

\[ \Gamma \vdash e_i : \tau_i \quad \quad (\forall i \in [1..n]) \quad \quad \text{(CONS)} \]

\[ \Gamma \vdash c(e_1, \ldots, e_n) : c(\tau_1, \ldots, \tau_n) \]

\[ \Gamma \vdash e : \tau \quad \tau < : \tau' \quad \quad \text{(SUB)} \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau' \quad \tau' < : \tau_1 \quad \quad \text{(APPL)} \]

\[ \Gamma \vdash \lambda(x) \Rightarrow e : \tau_1 \rightarrow \tau_2 \quad \quad \text{(FUN)} \]

\[ \tau_1 < : \tau'_1 \quad \tau_2 < : \tau'_2 \quad \quad \tau'_1 \rightarrow \tau_2 < : \tau_1 \rightarrow \tau'_2 \quad \quad \text{(S-FUN)} \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup x : \tau_1 \vdash e_2 : \tau_2 \quad \quad \text{(LET)} \]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \]

\[ \Gamma \vdash e : \tau_e \quad \Gamma \cup \{ x : \tau_x | x \in \text{Var}(p_i) \} \vdash \\
\quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau_{\text{bool}} \quad (\forall i \in [1..n]) \quad \\
\quad \tau_e < : \bigsqcup_{i=1}^n \tau_{p_i} \quad \quad \text{(CASE)} \]

\[ \Gamma \vdash e : \tau_e \quad \Gamma \cup \{ x : \tau_x | x \in \text{Var}(p_i) \} \vdash \\
\quad b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau_{\text{bool}} \quad (\forall i \in [1..n]) \quad \\
\quad \tau_e < : \bigsqcup_{i=1}^n \tau_{p_i} \quad \quad \text{(LETREC)} \]

\[ \Gamma \vdash \text{case} : \bigsqcup_{i=1}^n \tau_{b_i} \]

\[ \Gamma \vdash \text{case} : \bigsqcup_{i=1}^n \text{FIX}\tau_{b_i} \]

Figure 3: Derivation rules for \(\lambda_S\).
follow standard definitions. With the derivation rule of subtyping of functions (Rule (S-Fun)) we would like to express that it is safe to allow a function of one type \( \tau_1' \rightarrow \tau_2 \) to be used in a context where another type \( \tau_1 \rightarrow \tau_2' \) is expected as long as none of the arguments that may be passed to the function in this context will surprise it \( (\tau_1 <: \tau_1') \) and none of the results that it returns will surprise the context \( (\tau_2 <: \tau_2') \).

To type a \texttt{case} expression, first we type the head expression \( (e, \text{which has the form defined in the case branch on Figure 1}) \). For each branch we extend the context with types for free variables of patterns \( (\text{Var}) \) and calculate types for the patterns and the body of the function. The guards must evaluate to the boolean type. We denote the least upper bound type by the operator \( \tau_1 \sqcup \tau_2 \). The head of the expression has to be the subtype of the LUB of the types of the patterns \( (\tau_{p_i}) \). The return type is the LUB of the types of the bodies of the branches.

\texttt{Letrec}(Rule (\text{Letrec})) is similar to \texttt{case}, but it uses the fixed point of the return types of the branches \( (\text{FIX}f = f(\text{FIX}f)) \).

The recursive let expression can be typed in our scope only if it consists of a simple recursion function. We provide the constructive algorithm in the following theorem:

**Theorem 3.3.2** (Constructive derivation rule of \texttt{letrec}). Suppose the notation of Figure 2. Let us denote \( J := \{i \mid b_i \text{ does not contain } x\} \). Then the following constructive derivation rule holds:

\[
\begin{align*}
\Gamma & \vdash e : \tau_e \\
\Gamma \cup \{y : \tau_y \mid y \in \text{Var}(p_i)\} & \vdash b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau_{\text{bool}} \ (\forall i \in J) \\
\Gamma \cup \{y : \tau_y \mid y \in \text{Var}(p_i)\} & \cup \{x : \bigsqcup_{i=1}^{n} \tau_{p_i} \rightarrow \bigsqcup_{i=1}^{n} \tau_{b_i}\} \vdash \\
b_k : \tau_{b_k}, p_k : \tau_{p_k}, g_k : \tau_{\text{bool}} \ (k \in [1..n] \setminus J) \\
\tau_e & <: \bigsqcup_{i=1}^{n} \tau_{b_i} \\
\Gamma & \vdash E : \bigsqcup_{i=1}^{n} \tau_{b_i}
\end{align*}
\]

(LETREC-C)

where \( E \) is the \texttt{letrec} expression defined on Figure 2.

**Proof.** Let us first divide the case expression into two parts: the ones that do not contain recursive calls and the others that do. For the non-recursive branches we use the regular \texttt{case} typing derivation rule, hence \( \Gamma \cup \{y : \tau_y \mid y \in \text{Var}(p_i)\} \vdash b_i : \tau_{b_i}, p_i : \tau_{p_i}, g_i : \tau_{\text{bool}} \ (\forall i \in J) \) holds. This expression has the type of \( \bigsqcup_{\forall i \in J} \tau_{b_i} \) as per the case derivation rule. Let us later refer to this as the non-recursive type.
For the recursive branches, we have two cases:

1. Tail-recursion, as in $b_k \equiv x b'_k$. The expression in this case holds the type of the previously mentioned non-recursive type. We extend the type context with $x : \tau_{p_k} \rightarrow \tau_{b_k}$ where $\tau_{p_k} \leftarrow \bigcup_{i=1}^{n} \tau_{p_i}$ and $\tau_{b_k} \equiv \bigcup_{i \in J} \tau_{b_i}$, i.e. not changing the non-recursive type.

2. Non-tail recursion. We type the body by applying the intermediate type that has been calculated so far to the recursive expression, then calculate $x : \tau_{p_k} \rightarrow \tau_{b_k}$. This will be then added to the type context thus the intermediate type of the expression will be extended by $\tau_{b_k}$ to $\bigcup_{i \in J} \tau_{b_i} \sqcup \tau_{b_k}$.

With the above considerations we can type all branches of letrec, turning it into a regular case expression that has the type of $\bigcup_{i=1}^{n} \tau_{b_i}$, resulting in the following type: letrec : $\bigcup_{i=1}^{n} \tau_{b_i}$.

3.4 Practical Approach

3.4.1 Typing Recursive Functions

The fundamental design and structure of the Scala compiler make it an excellent candidate to be extended. The features of the compiler that make this possible are high separation of compiler phases, the support for macros and fully independent compiler plugins and the compiler being an open source project [118]. The phases of the compiler start by parsing the source files and generating an AST; later phases transform this AST. The current version of compiler is written in Scala, using Scala objects to describe the nodes of the AST. The compiler also acts as a library to analyze and compile Scala source code, providing a programmatic API that can be accessed from applications.

As the first step, we had to find a way to circumvent the type error generated for recursive functions. In the default version of the compiler, when the typer starts calculating the type of a (recursive) function and it meets an entity that has been defined previously, but the type of it is yet to be determined, it will throw a CyclicReference exception that is collected and handled by the generic error handling infrastructure of the compiler. This does not stop the typing phase from continuing with typing other entities. We leverage this property, as it collects all the erroneous recursive calls, but types all other to-us trivial cases.

Our main approach of extending the compiler has focused on creating a separate codebase, as modifying the main branch directly was found to be too time consuming
and difficult. Fortunately, we were helped in this effort by the various API calls provided by the package `scala.tools.nsc._` (nsc stands for New Scala Compiler).

The extended `scala.tools.nsc.typechecker.Analyzer` contains methods overridden that handle control flow ASTs – namely `if`s and `case`s – and method definitions, so we can annotate these methods with our calculated type. We of course use the original typer to first type these entities and only interrupt cases that are of interest to us. Finding the least upper bound of types is another key point in our algorithm, but we were very fortunate in this regard: we can use the function `lub` on `scala.tools.nsc.TypeChecker.Typer`.

The last remaining piece to have a working compiler was inserting the newly created typer into the chain of compile phases and invoking it from the first step, the parser. We have achieved this with our own entry point to an application that simply passes a path to the compiler and invokes it. We only use regular console reporting by `scala.tools.nsc.reporters.ConsoleReporter` and global settings by `scala.tools.nsc.{Global, Settings}`.

We have measured how our extension affects compile speeds by calculating the total time spent in the `main` method of our application. The results are shown on Table 1. For simplicity, we have listed measured microseconds with the default version of the compiler – i.e. invoking it without setting the extended typer – and with the extension in place. As it can be seen in change percentage, the extension has no significant impact on performance. Example1 and Example2 refer to the examples seen in Section 3.1, while Multiple methods contains several other test cases.

<table>
<thead>
<tr>
<th>Code snippets</th>
<th>Default</th>
<th>Extended</th>
<th>% change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example1</td>
<td>2440166</td>
<td>2479896</td>
<td>101.62</td>
</tr>
<tr>
<td>Example2</td>
<td>3390745</td>
<td>3422435</td>
<td>100.93</td>
</tr>
<tr>
<td>Multiple methods</td>
<td>5691124</td>
<td>5712335</td>
<td>100.37</td>
</tr>
</tbody>
</table>

Table 1: Compile times (µs) with and without using our extension.

Restrictions

Scala supports defining recursive types using explicit type annotations. Soundly calculating all recursive types would require extending our inference method with a fixed point calculation. Henceforth, our proposed method in its current form does not support recursive types.

We implemented and tested our compiler extension only on Scala compiler version 2.12.4. It is not guaranteed to work with any other version than 2.12.4.
3.4.2 Performance Analysis of Implicit Resolution

Finding performance bottlenecks in software is not always easy as the complexity of the software increases. The Scala compiler is possibly one of the most complicated Scala products. The code heavily relies on the standard library to implement ASTs. For this reason, regular JVM-based profiling tools do not necessarily identify high-level hotspots, but rather “low-level” APIs, such as list append (`List.::`) and hash value computation. This led us to develop our own methodology to understand which parts of the compiler are slow to run.

![Figure 4: Average distribution of time spent on each compile phase.](image)

**Compiler Plugin**

We have developed a compiler plugin (please see Section A.1.10 for reference) that measures the time it took finish each compile phase. There are phase groups that cannot be interrupted by custom plugins because the intermediate AST transformation would break. Due to the nature of the compiler, we then needed to appropriately aggregate the results for larger codebases. These measurements let us notice that the compiler, depending on some characteristics of the source code, spends between 30% and 50% of its time in the typer phase. The compilation time distribution of a large, proprietary codebase can be seen on Figure 4. Besides this software we have also analyzed the Scala language distribution itself – the compiler and the standard library of the language. Both of these provided the same performance characteristics, and made it easy to see that the biggest performance improvement can be
achieved by enhancing the typer. Since it also includes resolving implicits, we have taken a closer look at this subphase, and recognized it as a single point for possible improvements.

**Synthetic test suite**

To test our hypotheses, we needed a method to measure orthogonal aspects of the compiler and how they affect the performance. We had to realize that it is not feasible to find one live codebase that meets our criteria and comparing different codebases can lead to false conclusions since the language allows developers to use significantly different coding styles.

These realizations led us towards creating a test suite that can generate Scala code that meets our requirements while maintains enough similarity so that comparisons remain meaningful. Since parsing is one of the least significant phases, we have decided to generate Scala code directly and run the compiler with our benchmark plugin to gain a better understanding on how different aspects influence the speed.

The implicit resolution algorithm has two main factors, the number of in-scope implicits and their usages. The different types of implicits used – parameters, classes or converters – all use the same algorithm so the usage patterns do not drive relevant changes. A small example of generated code can be seen on Listing 15.

### 3.5 Evaluation

In this Section we will evaluate our approaches to improve both the type inference and the use of implicits in Scala.

We show how the previously described type inference algorithm works in practice: we refer back to our original issue statement (Section 3.1) and demonstrate how the two examples there can be fixed by our solution.

We also introduce the results of our extensive analysis of the performance characteristics of the compiler. Furthermore, we will outline possible techniques to improve both the speed of the compiler as well as the long-term maintainability of software code using implicit resolution.

#### 3.5.1 Typing Simple Recursive Functions

Our first example is an unusual version of the map function on Listing 16. Its purpose is to return the mapped results in a list of the smallest type possible. As one can see in the Listing, the main body of the function is a `match` with two possible `case` branches.
package gen.implicits
import scala.language.implicitConversions

class T
class I1

object Impls {
  implicit def imp1(x: I1) = {
    println("implicit conversion from I1 to T")
    new T
  }
}

import gen.implicits._
import gen.implicits.Impls._

object ClientImplicitly {
  def doCall(t: T) = println("implicit conversion may apply")
  def main(args: Array[String]) = {
    doCall(new I1)
    doCall(new I1)
  }
}

object ClientExplicitly {
  def doCall(t: T) = println("implicit conversion may apply")
  def main(args: Array[String]) = {
    doCall(imp1(new I1))
    doCall(imp1(new I1))
  }
}

Listing 15: A recursive function with no defined return type.

def map[C, A <: C, B <: C](y: Seq[A], f: A => C) = {
  y match {
    case Nil => Seq[B]()
    case x :: xs => map(xs, f) ++ f(x)
  }
}

Listing 16: A recursive map implementation with type inference.

This instruction flow is very similar to the extended $\lambda_S$ language. Each branch has a return type, namely the first one returns a Seq[B] and the second one a Seq[C]. The type of the second case is defined by the result of the concatenation(:+) method. This part can be rewritten in a form of map(xs, f).:+(f(x)). If we consider this function call, the appended element is of type C, as we have declared f to be a function of A => C. Since A is a subtype of C, the result of the map function is
a subtype of C, making the result of the append call be a type of Seq[C]. This leads us to two calculated types for the branches: Seq[B] and Seq[C]. The typing mechanism we propose would now calculate the least upper bound for these types that would be Seq[C], and typing our special map function with Seq[C].

```scala
def deepRec(n: Int) = {
  if (n == 0) {
    Class1Level2
  } else if (n == 1 || n == 2 || n == 3) {
    n match {
      case 3 => Class2Level12
      case _ => {
        if (n != 1) {
          deepRec(n - 1)
        } else {
          Class2Level11
        }
      }
    }
  } else if (n == 4) {
    Class1Level2
  } else {
    Class1Level1
  }
}
```

Listing 17: A recursive function using type hierarchy on Listing 4. with type inference.

The next example is more involved. First we start off with declaring a multi-level type hierarchy as seen on Listing 4. This hierarchy declares 3 levels with leaf nodes being case classes. The recursive function using these classes is defined on Listing 17. The control flow graph created by the predicates in the function has several branches, unlike the straight tree of the match and cases in the first example. This will cause no problems to our typing algorithm, as it can be applied to all leaf branches, and then going upwards in the tree using the previously calculated types. This calculation starts with the if-else on line 9. The first branch contains a recursive call, so we cannot type this branch. We need to start with the else branch first, making the calculated type Class2Level1. The case branch on line 8 then would be typed the same. We have a trivially typed case on line 7, with Class2Level12. The least upper bound for these types is Level12, so the else branch will be typed Level12. We have arrived at the top-most level of predicates, that has types of Class1Level12, Level12, Class1Level12 and Class1Level11, in respective order. The least upper bound defined by the hierarchy is therefor Level11. This will be the final type of the deepRec function.
The \( \lambda \)s language was only defined for single-parameter functions. The reason we can still use it to calculate types for these functions is that by currying multi-parameter functions, we can always transform them to a chain of function applications having only one parameter. Since Scala supports currying by default – by denoting a function call with the \( _ \) (underscore symbol) –, we assume that all functions in question can be transformed to this kind.

### 3.5.2 Typing Recursive Functions with Multiple Branches

In the previous examples we discussed functions with a single recursive branch. In the following we show that a similar solution exists for simple functions with multiple recursive branches.

```scala
1 class A {
2   def toC: C = ...
3   def toD: D = ...
4 }
5
6 class B extends A
7 class C extends A
8 class D extends A
9
10 def multi(n: Int) = n match {
11   case 0 => new B
12   case n > 0 => foo(n - 1).toC
13   case n < 0 => foo(n + 1).toD
14 }
```

Listing 18: A function with multiple recursive branches.

In the example shown on Listing 18 there are two recursive branches resulting in two different but related types. Our algorithm finds the non-recursive branch on line 11 and calculates type \( B \). Typing the second branch will use this information as the return type of the \( foo \) call. Using \( B \) as a placeholder type, the default type inference algorithm of Scala will calculate type \( C \) as the return type of the branch on line 12. Similarly, type \( D \) will be calculated for the branch on line 13. Finally, the \( LUB \) of types \( B, C \) and \( D \) will be determined as \( A \). Therefore, the return type of function \( multi \) will be \( A \). This result complies with the expected result type of our algorithm and meets the intention of the developer.

### 3.5.3 Performance Aspects of Implicit Resolution

There are two main factors affecting implicit resolution performance, namely how many implicits are in a given scope and the amount of implicit usage. We will
TYPE SYSTEM IMPROVEMENTS

present data that steered us towards trying to simplify the search criteria and then
providing some coding guidelines and practices with regards to implicit usage (in
Section 3.5.4).

Implicit Usage

![Compilation times with a constant number of available implicits and increasing implicit usage.](image)

The most basic variable to change is the number of implicit calls the compiler
has to resolve. For the following tests we have fixed the number of available implicits
in the given scope, then increased the number of required implicit applications. On
Figure 5, the blue bars represent the number of implicit uses, the dotted lines display
how many implicits are in scope, between 1 and 1000. As it can be seen, there is no
high correlation between the two variables: increasing the implicit usage does not
affect compilation times by a great amount – the change could be attributed to the
growing source code alone.

Available Implicits in Scope

Based on the rules listed in Subsection 3.2.4, another dimension to investigate is
the number of implicits available when the compiler needs to resolve them. We have
generated code with an increasing number of implicits, while the actual usage of
implicits remained constant. The aggregated analysis can be found on Figure 6. As
previously, the blue bars show the main factor we are testing: the number of implicit
definitions in scope. The dotted lines are now used to express how many implicits
are actually used in the code. It is easy to see the direct correlation between the two
factors for each test case. As we increase the set of definitions in which the compiler
needs to search for implicit candidates, the typer phase takes proportionally longer.

Figure 6: Compilation times with constant implicit usage and an increasing number of available implicits.

Conclusions

Based on the performance analysis we have described in this Section, we can draw the conclusion that the main factor that affects the performance of implicit resolution, and more widely, the typer phase in the Scala compiler, is going through all the available definitions in the scope to search for implicit candidates. There are two possible ways to optimize this:

- Make analyzing each item faster, thus applying the operation on the whole set faster.
- Decrease the size of the input set – in effect, reduce the number of implicit candidates in scope.

In the following Section, we will consider both approaches and describe what we have observed.
3.5.4 Approaches to Improve Implicit Resolution Performance

Relaxed Search Criteria

The inferring algorithm that is used by the implicit resolution code path in the Scala compiler needs to decide if two types are compatible with each other in the meaning of the Liskov substitution principle [64]. This problem exposes a non-trivial algorithmic complexity, and based on the profiling information we have described previously, takes considerable amount of time for every typing error. The implementation [102] has a way to relax (or shortcut) the search algorithm so it does not perform a full type check, but rather uses the much simpler `isPlausibleSubtype` method. Interestingly enough, the incoming parameter has been denoted as `fast: Boolean`, making it a trivial choice for experiments. We have assumed that this shortcut could be used for some usages of the compiler, for example for incremental compilations used in IDEs. Although it would not solve the performance issue once and for all, it could provide big enough improvements for software developers.

![Figure 7: One explicit function call and one implicit resolutions with a varying number of implicits in scope.](image)

We have compiled our own version of Scala and used this modified version in our performance tests. On Figures 7 and 8, we can see that relaxing the search criteria does improve implicit resolution times significantly. As the line charts display, there is not much of a difference between explicitly applying the implicit function and letting the compiler find it as a candidate.

We have managed to build the modified Scala version and run our performance tests with it, but as a semantic correctness test, we have also tried to compile the same version of compiler code and run the enclosed test suite with it. Unfortunately
we have run into several issues. Resolving implicits involving existential types has proven to be problematic:

```scala
[error] [...]/scala/trytobuild/scala/src/library/scala/collection \n/parallel/ParIterableLike.scala:527: type mismatch;
[error] found : scala.collection.parallel.IterableSplitter[T]
[error] Note that implicit conversions are not applicable because \nthey are ambiguous
```

Figure 8: 1000 explicit functions call and 1000 implicit resolutions with a varying number of implicits in scope.

We have tried to work around these issues by not running the relaxed algorithm for these existential types, but this has proven no use. As we have been working through these issues, we had to increase the complexity to decide if the fast code path resulted in sufficient type checks. By the time we have managed to cover all the known cases, we lost all the performance gains with more complex and less maintainable code.

**Coding Patterns**

As discussed previously, reducing the size of the candidate set for implicit resolution can be another useful approach. For this reason, we would eliminate all useless implicit definitions before running the typer phase. We have considered creating a programmatic way for removing these definitions right before the typer meets them,
but concluded that we would probably create an algorithm that is at least as slow as the one in the compiler. Doing this would probably not help the performance.

Instead, we have realized that the most trivial way to reduce the number of available implicits is by not having them in the software code. We have examined several large software projects written in Scala, and identified usage patterns for implicits that worsen the performance, but do not really help the developer or improve the code. With a little more care, these can be changed, resulting in a cleaner and faster-compiling code. We have created some coding patterns and guidelines that are easy to follow. They not only help with the performance, but adhering to them can possibly reduce implicit-related errors that usually turn out to be hard to find and debug.

- **No file-level imports for implicits.** A lot of times, imports are moved to the top of a source file either by IDEs or developers themselves. Normally, this improves neatness of the code, but with implicits, this necessarily bogs down the compiler. Every implicit needs to be added to the candidate set and for larger files that contain multiple classes, the size of the list of imported implicits can be significant. If there are no implicits imported at the file-level, the mixture of these classes will have no effect on each other.

- **Import implicits only in the block they are being used.** Minimizing the scope for implicits should be the ultimate goal. Since Scala is a block-structured language, it supports imports in all blocks, and due to the visibility rules, these imports cannot escape the block. If implicits are only imported in methods or bodies of structures where they are used, they cannot be applied—accidentally or not accidentally—in other places, and the compiler does not need to check them for every type error outside of the block.

- **Do not wildcard-import implicits.** Importing all contained entities from a namespace is often useful, but it can accidentally import many implicits. The most trivial solution to the problem is only importing the necessary definitions by name.

- **Create implicits in smaller namespaces.** Providers of implicits should try to reduce the number of them in their namespaces. This encourages the previous point and lets users of implicits selectively add them to their code. This also better documents code as it’s easier to track down the intent for a given implicit.

- **Do not create silver-bullet type collections of implicits.** This reinforces the previous pattern, but it’s important to highlight this as many patterns we have
observed dumped all seemingly related implicits in on huge namespace that got imported in many places. This should be avoided by both the providers and the clients of implicits.

Due to the nature of the project we did not have the opportunity to further analyze the effects of applying these guidelines. The reasons are two-fold. Firstly, because of the size of the project, any refactors or non-functional but significant changes to the whole codebase cannot be completed at once and usually take multiple months. Secondly, we could not have granted access to key project properties that are required to understand the cost-effectiveness of such refactors. For these reasons we have searched for publicly available issue descriptions, posts and recommendations that would have either been prevented by our guidelines or are in line with them. The first one [104] describes a case where having two distinct implicits having the same name imported in the same scope gives a compilation error, while removing one of them resolves this. The issue originates in the rules of static Java imports and could be avoided by applying our rule 4. Two very similar cases can be found at [130] and [103]. The problem there is that the automatically imported scala.predef package contains implicits that convert a String to Int and Double and users also defined these. The issue could have been avoided by adhering to rules 2 and 4. Lastly we mention a Scala best practice described at [67]. Importing the same name more than once into any given scope will shadow the first imports. In most cases this is discovered by IDEs or gives a compilation error, but it can become a hard-to-discover bug when happens with implicits. While giving unique names is a workaround, applying our third guideline can a better choice as changing or obfuscating a name of a method decreases maintainability and should be avoided if there are other solutions.
3.6 Thesis I.

I have analyzed how the type system of Scala supports providing succinct and elegant solutions to problems and I have found two areas of possible improvement.

Firstly, I have created a novel approach to type simple recursive functions. I have proved its correctness by defining a new formal language based on typed lambda-calculus and used typing rules to prove the correctness of the theorem. I have implemented this type inference algorithm as an extension to the Scala compiler and tested the implementation on various cases.

Furthermore, I have investigated how implicit resolution affects the performance of the compiler. I have created a specialized compiler plugin to make meaningful measurements and created a test suite to generate synthetic code to test my hypotheses. I have worked out easy-to-follow guidelines for programmers that help reduce the potential errors and the time compilation takes when using implicits.

<table>
<thead>
<tr>
<th>Thesis Title</th>
<th>Relevant Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Improvements to The Type System of Scala</td>
<td><img src="bullet" alt="bullet" /></td>
</tr>
<tr>
<td>(2) Effects of Error Handling on Software Complexity</td>
<td><img src="circle" alt="circle" /></td>
</tr>
<tr>
<td>(3) Scale-free Properties of Software Systems</td>
<td><img src="circle" alt="circle" /></td>
</tr>
<tr>
<td>(4) Read-Copy-Update in Scala</td>
<td><img src="circle" alt="circle" /></td>
</tr>
</tbody>
</table>
Chapter 4

Effects of Error Handling on Software Complexity

4.1 Motivation

Handling errors in computer programs is a mandatory and essential feature. The origins of errors can be numerous, including bad user input, unexpected changes in the runtime environment or a misuse of APIs and their implementations. It is easy to see that errors can be caused by all parties that are involved in the lifetime of the software: the developers, the operators and the clients. Depending on the use case, these errors can be very costly or even catastrophic.

Error handling methodologies have evolved with software development practices. One of the trivial solutions is encoding errors as part of the data passed to and from subroutines. The developer can choose to dedicate values of the output to represent erroneous execution. This method has many drawbacks, including mixing the domain-specific code with error handling, the need to encode all possible errors in the domain-specific types and representing error types with unrelated types. The approach is still present in many places, notably in process return values: in most operating systems, if a process did not encounter any issues, its return value will be 0, any other value represents an error. The values and their meanings vary between implementations. Using this practice has severe limitations and is only usable in small-scale software.

An improvement to the previous method is introducing a global error state. A notable implementation of this is the ERRNO variable in the C standard library [38]. ERRNO is a global integer value which stores all error states of system calls, but its value is also tied to the return value of the system call: it only signifies an occurrence of an error if the return value is either -1 or NULL. The header file also defines many preprocessor macros that bind symbols to integer values, e.g. EDESTADDRREQ:
Destination address required (POSIX.1-2001). While using a global state does help with the separation of concerns, it still has many disadvantages. The caller must remember to check the state of the global error value (and in the case of ERRNO, the return value of the call as well) to determine if errors occurred. It makes the handling part of error handling a concern of the client which is easy to miss, leading to unhandled exceptional cases. The code that checks for errors is still mixed with the production code, reducing readability and maintainability. Having a global state for errors has the same drawbacks as having any other global state: any part of the program can access it. In concurrent software, this is a serious issue as we can never be sure that the occurred error has been properly handled. While ERRNO is per-thread (thus trivially thread-safe) this is not necessarily a requirement. Furthermore, the errors are still encoded as a global type that is usually a simple type. A mapping must be defined and managed to give meaning to the values.

To help with the separation of error handling code from normal code, the C standard library defines `setjmp` and `longjmp` calls which can be used as non-local gotos: they define a control flow that differs from regular subroutine call sequences. They are usually used to define an implementation of an exception mechanism. `setjmp` saves the state of the current running program while `longjmp` can return this state. There are several limitations of this. Even though this allows for non-standard jumps in the stack, no automatic stack unwinding happens, including any necessary cleanup. This can lead to resource leaks if not used carefully. The pair of these instructions can only be used safely once in every thread, as the stack becomes corrupted. Furthermore, using these instructions directly can be very cumbersome and require close attention to details.

The notion of exception handling tries to answer the shortcomings of the previously described error handling methods. The first implementation of error handling that we would consider exception handling by today’s standards was done in LISP [41]. This implementation already had the notion of raising and catching exceptions, while the generic cleanup phase – most commonly referred to as `finally` today – was later introduced in NIL (New LISP Implementation).

Error types in languages supporting exception handling are dedicated to represent the error only. Depending on the language, either values of any type (e.g. C++) or instances of subtypes of a dedicated built-in type (Java, Scala or C#) can be used as exceptions. Usually these are complex objects (not considering the case when an simple type is thrown in C++) containing valuable information on the error itself. They can further have a user-friendly message or contain the callstack to the point where the exception happened. Java also distinguishes between checked or unchecked exceptions – that is, if the caller must handle the exception or not. The usefulness of checked exceptions has been of a great debate [34]. There are languages
that allow defining the possible exceptions thrown by a method. In Java, this is done by using the \texttt{throws} keyword, while in Scala one can use the \texttt{@throws} annotation.

Stopping the execution and raising an exception usually happens with the \texttt{throw} keyword. In most languages, this requires creating a new object on the heap. At this point we have the ability to include all information we deem useful for later investigations of the problem. Catching a thrown exception happens by defining a \texttt{catch} block or statement for a \texttt{block}. The execution points of throwing and catching an exception can be separated by many calls, helping differentiate normal program code from exceptional code paths. Finding the point of catch for any thrown exception is done by unwinding the stack: the runtime will keep stepping backwards on the callstack until it finds the corresponding catch. Most languages have a top-level handler for exceptions that were not caught by user code. While stack unwinding happens, all intermediate objects will be destructed, making sure the program execution cleans up resources properly. Modern runtimes (e.g. the JVM or the CLR) support \texttt{finally} blocks, that are guaranteed to run after either a normal or exceptional completion of a try block. Finally blocks can be used to free resources (e.g. release locks) even in complex situations or ones that are hard to reason about, such as in a multithreaded environment.

Despite its wide adoption, there are some critiques for exception handling as well. Since it allows for non-structured code paths (behaving like a convoluted goto), some programmers and institutions consider it bad practice for safety-critical applications \[82, 50\]. Another point of criticism is that exception handling code is not handled correctly in large systems, resulting in code paths that are hard to reason about \[128\]. As with many other programming language features, exceptions can be used poorly or advantageously.

Even though exception handling is considered an essential feature of all modern programming languages supporting the structured paradigm, several questions arise around the different error handling methods, such as how traditional error handling compares to using exceptions with regards to complexity or how having non-structured constructs changes this and if monadic error handling can help with the increasing complexity.

4.2 Related Works

4.2.1 The A-V Complexity Metric

The basis of our analysis is the A-V software complexity measure, that has been defined in 2002 \[96\]. The need for this new metric originates in the appearance and wide adoption of object-oriented and multiparadigm methodologies. While most
complexity metrics can be used to analyze structured programs, in the last decade they have been proven to be unsatisfactory. Cyclomatic complexity, for example, was defined in 1976 with Fortran programs in focus [70]. Classical software metrics were designed with languages at lower levels of abstraction in mind. A good example with OOP languages is that some branching logic is implicit via method overloading and virtual function resolution, therefore the cyclomatic complexity does not discover them, even though they still increase complexity. A-V metrics designed to measure these features and proved to be effective for multiparadigm programming languages. In this Section we will give a short introduction to A-V and will define terms and methods required for our extension.

The A-V builds on McCabe’s basic constructs. The basis for its calculation is the same control flow graph [40], but instead of using cyclomatic numbers to describe this graph, it defines nested deepness for predicates. Later research, however showed the importance of nested deepness of control statements [51, 95].

**Definition 1** (Nested deepness, ND). *This number represents how many predicates are in scope for a given statement in the program. The formal definition of nested deepness can be found in [109].*

Using nested deepness directly describes how many predicates and branches need to be understood for a statement in the code: the more branches and loops you need to apprehend, the more predicates and variables you need to keep track of.

Using data is the central matter for almost all software. This trend has been rising with the spread of the object-oriented paradigm, where the core ideas revolve around objects, how they are represented and what operations one can apply to them. These factors led to the necessity of considering how data is handled when one argues about software complexity. The A-V adheres to this trend with extending the control flow graph with data access nodes. Edges to these nodes are present from control flow nodes, when a variable is either written or read. It also allows counting these multiple times between the same nodes. The weight of these data access nodes is always given by the nested deepness of the statement they are connected to.

These consideration lead us to the following definitions:

**Definition 2** (A-V measure of a method $f$).

$$\sum_{s \in \{\text{Statements of } f\}} \text{ND}(s) + \sum_{v \in \{\text{Variables of } f\}} \text{ND}(v)$$

**Definition 3** (A-V measure of a class $C$).

$$\sum_{f \in \{\text{Method of } C\}} AV(f) + |\{v : V \text{ is a member in } C\}|$$
The above definitions make it clear that the A-V metric is a natural response to the issues of cyclomatic complexity. Measuring complexity using these methods leads to a better description of modern software and also provides a solid basis for our own analysis.

4.2.2 Monads

Monad is a design pattern in the functional programming world [127] that is usually used to express a chain of computations. Its origins come from the category theory definition as it behaves similarly to mathematical monads [19]. Any functioning monad needs three essential parts:

- **A type constructor.** For any type \( T \) it returns \( \text{Monad}[T] \).

- **A unit function.** For any object \( x \) of type \( T \) it returns \( \text{Monad}(x) \) that is of type \( \text{Monad}[T] \).

- **A bind function.** For any types \( A \) and \( B \) and the monad \( \text{Monad}[A] \) it transforms \( \text{Monad}[A] \) into \( \text{Monad}[B] \).

The following code could be used as the basis for implementation of monads in Scala [27]:

```scala
trait Monad[A] {
  def flatMap[B](f: A => Monad[B]): Monad[B]
}
object Monad {
  def apply[A](x: A): Monad[A]
}
```

Listing 19: Scala monad.

Here, the trait itself is the type constructor, `flatMap` serves as the bind, and `apply` as the unit function. More intuitively, monads can express chains of computations, wrapping modifications to the computation’s state. One major usage pattern in pure functional languages is handling I/O [44]. The I/O monad’s state is passed between the instances that apply modifications to the underlying resource, be it a file or the console output of the program. The aforementioned operations are essential to create and use a monad, but on top of them, they need to also satisfy the monadic laws, as described in [77]:

1. **Left identity.** \( \text{Monad}[A](x).flatMap(f) \) equals to \( f(x) \).

2. **Right identity.** \( \text{Monad}[A](x).flatMap(\text{Monad}[A](\_)) \) equals to \( \text{Monad}[A](x) \).
3. **Associativity.** $\text{Monad}[A](x).flatMap(f).flatMap(g)$ equals to $\text{Monad}[A](x).flatMap(f(_).flatMap(g))$.

These three laws guarantee that using a type providing these functions naturally behaves as expected from a monad.

### 4.2.3 Error Handling with Monads

While error handling has been made straightforward for structured programs by the concept of exceptions [43], their usage inherently depends on having one execution stack. This has been the case for the majority of software code in the past, but with the advent of multi-core systems and the need to leverage concurrent programming models, this has drastically changed. Furthermore, the headway of functional and multiparadigm languages has opened the way for easier asynchronous and reactive programming [59]. These improvements either require additional code to make exceptions work or we need to use a completely different model. This is where monads come into the picture. As we have discussed, monads naturally serve as building blocks to form a chain of computations. A link in these chains can be one that handles errors. Since monads have no dependence on how they are executed, they can reduce the reliance on the stack.

There are several monads in the Scala standard library that we could potentially use for error handling (e.g. $\text{Option}[+A]$, $\text{Either}[+A, +B]$ or $\text{Try}[+A]$), but we will focus on $\text{Try}[+A]$, as that is the designated monad to handle errors in computations.

The $\text{Try}$ type represents a computation that may either result in an exception, or return a successfully computed value. It is similar to, but semantically different from the $\text{scala.util.Either}$ type [107]. It has two final subtypes, $\text{Success}[T]$ and $\text{Failure}[T]$ and users can only create instances of the subtypes. Based on the API documentation cited above, one can see that $\text{Try}$ has the necessary methods with the right signatures to be used as a monad:

- **def apply[T](r: => T): Try[T]** on the companion object serves the purpose of the unit function.

- **def flatMap[U](f: (T) => Try[U]): Try[U]** is the bind function.

- Finally, as we have discussed earlier, we do not need anything special for the purpose of the type constructor in Scala.

It is easy to see that the monadic laws are satisfied by $\text{Try}$, we only mention two technicalities one needs to keep in mind while working through the proof:
• It is important to use the same exception instance while proving left identity and associativity hold, as Java does not define well-formulated equality between throwables (it relies on reference-based equality); even though we would intuitively expect value-based equality.

• One needs to use \texttt{Try} itself instead of \texttt{Success} when proving left identity as there is a difference between how these two types handle uncaught exceptions: while \texttt{Success} will bubble it up as a regular exception, \texttt{Try} will convert it to \texttt{Failure} – and this is the correct approach needed for the proof.

4.2.4 Error Handling in Pure Functional Languages

It is easy to see that the imperative way of handling exceptions by throwing and catching them is not pure. While throwing an exception does only depend on the parameter values and the definition of the function, catching an exception can be non-referentially transparent. Moggi had defined the exception monad \cite{76} based on category theory. His work also provides denotational semantics for evaluating monads. He introduces two operations for such monads, \texttt{raise} and \texttt{handle}, that behave very similarly to what we would expect. He also argues that the exception monad is the most “well-behaved” compared to other side-effect monads.

Schroder and Mossakowski had extended Moggi’s work by defining and implementing an equational definition for the Java monad \cite{108}. This work shows how abnormal termination can be transformed then handled by using monads. They also provide formal proof through Hoare-calculus which leads to the possibility to extend these calculi to missing computational features. Simon Marlow et al. had defined asynchronous exception handling for their extended version of Haskell \cite{68}. In their work they define scoped combinators for synchronous and asynchronous interrupts. The main concept is \texttt{throwTo} that can be used to throw an exception to a given thread. Its implementation is non-trivial as the receiving thread can be blocked. The solution to their problem is applying the blocked and unblocked combinators. Furthermore, they define operational semantics to formally prove the correctness of their approach.

4.3 Theoretical Resolution

4.3.1 Extending Metrics for Exception Handling

In this Section we propose extensions to the cyclomatic complexity and A-V metric for the case of exception handling. We start with cyclomatic complexity, because it is the basis of the A-V metric, we then extended the latter measure as well.
The basis for the metric defined by McCabe is the role of predicates in the code, because these predicates set the execution path of the given code block. We need to introduce the notion of catching and throwing exceptions to this model. A trivial consideration is to think of all catch blocks after a throw as predicates: if there has been an exception, these catch blocks are predicates on the type of the exception, hence in the extension each catch branch increases nested deepness by 1. Throwing exceptions is not this trivial. All the conditional statements that lead to a throw are of course accounted for, however we disregard them when calling these functions. The reason for this is that these function calls do not increase the number of predicates in client code, thus they should not be considered based on the original ideas of McCabe. If these functions are part of our code base, they will increase the complexity of our software.

The original A-V metric uses the idea of nested deepness to better characterize complexity, furthermore it registers data access points in the control flow graph, because data is becoming the central concept of our software. The extension we propose takes steps further along these lines; it tries to be faithful to the prime considerations of the A-V measure. We utilize the notion of nested deepness as well as extend the data graph with exceptions.

Most programming languages represent exceptions with a typed object. We can safely ignore the fact in our analysis if these types have to extend one root exception type (e.g. most JVM-based languages), thus we make no differences to the original A-V when it comes to creating, throwing or catching exception objects. It will be noted as a simple data access, considering the nested deepness of the statement. The call to the constructor might involve other variables 1..n, but these would be considered by the original A-V as well.

The most fundamental concept of A-V is to consider all decisions that need to be made leading to a given point in the program. This is handled by counting the number of predicates to which a statement belongs. Using this concept we can better describe throwing exceptions: we consider these statements with their nested deepness; apart from explicit throw statements, we also consider function calls as well, because they can potentially throw exceptions. If a function can throw more than one exception, we register all of them using the nested deepness of the call. The reason for this is that one needs to understand all of the conditions that can result in throwing exceptions and that increases the number of test cases and complexity in general. Of course, throw statements should be counted only once, since they can only create one type of exception – they are only responsible of describing one type of exceptional condition.

Catching exceptions in client code happens in the same method in which they were raised often times, although there can be significantly complex code between
these. It can easily happen that there are several branches that need to be evaluated, and this increases complexity. Because of these arguments, we consider the difference of nested deepness between catching and throwing an exception. Amidst a pair of a try-catch there can be multiple levels of checks that need to be understood. We naturally adhere to the semantics of programming languages by matching explicit and implicit throws with their catches and counting these occurrences. These pairs can always be made unambiguously – much like balanced parentheses.

**Definition 4** (A-V measure of the exception $e$).

$$
\Delta ND(e) = ND(\text{throw}(e)) - ND(\text{catch}(e))
$$

Contrary to client code, in libraries we cannot properly deal with an exception at the point of creation, making throwing and catching an exception far apart. To express this in our extension, we count all non-caught exceptions in the given method weighted by their nested deepness, and add it to the complexity of the method. The sole reason for this is that developers of the library need to understand the complexity that leads to raising an exception, but this increases complexity at the point of the use of the function.

With these considerations we arrive to the following definitions:

**Definition 5** (Extended A-V measure of the function $f$).

$$
AV(f) + \sum_{e \in \{\text{Thrown exceptions of } f\}} ND(e) + \sum_{e \in \{\text{Exceptions of } f\}} \Delta ND(e) + \sum_{e \in \{\text{Uncaught exceptions of } f\}} ND(e)
$$

Applying these definitions and methods we try to give an answer on how using exceptions should be considered in software metrics, so we can better account for software complexity.

### 4.3.2 Weyuker’s Properties

The original purpose of Weyuker’s properties [129] is to filter out all trivially wrong complexity metrics that will be proven to be useless in any real application. These statements are simple sanity checks against newly defined metrics. To show that we meet all of the properties, we build on the original proofs [96] and will use the same notations and formalism as in Section 2.3.1.
1. \[ \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \implies m(s_1) \neq m(s_2) \]

It is trivially easy to find two different programs with different complexities: one example can be a more defined `catch` block that has more branches than the other.

2. \[ \text{Let } c \in A, c \geq 0, S_1 \subset S | \forall s \in S_1 : m(s) = c \implies |S_1| \leq \infty \]

The original A-V only meets this requirement if the data connection edges are present multiple times when there is multiple use of the data [96]. If we keep this constraint while extending the metric, we will keep it intact, as we have defined data access for exceptions as regular variables.

3. \[ \exists s_1 \in S \land s_2 \in S : m(s_1) = m(s_2) \]

The simplest example here would be the snippet of code that calls two different functions that can throw the same exceptions (the number of exceptions can be 0).

4. \[ \exists s_1 \in S \land s_2 \in S : s_1 \neq s_2 \implies m(s_1) = m(s_2) \]

Two programs can meet this property if they, for example, differ in the way where they handle exceptions. If one function catches all possible exceptions in its body, while another one does not contain any `catches`, their complexity will be the same, while they hugely differ.

5. \[ \text{Let } s \in S, s_1 \in Ss' = \oplus(s, s_1) \implies m(s) \leq m(s') \]

Since all extensions in the original A-V increase the calculated complexity, this monotonic property is also true for the extended metric.

6. \[ \exists s_1 \in S \land s_2 \in S \land s' \in S : m(s_1) = m(s_2) \land \]
\[ \oplus(s_1, s') = \oplus(s_2, s') \implies m(s_1) \neq m(s_2) \]

For the original A-V definition this holds, because the metric considers data flow, thus it also depends on the environment of the program. Since we inserted the exception handling code into this data flow, it still holds for the extended definition.
Using the notion of nested deepness, it is easy to find programs for this constraint. Since catch blocks modify nested deepness, if we consider nested catch blocks and the same ones, but in a linearized way, we get two programs that meet this property.

Since we have not used these code items in our definitions, we trivially meet this requirement.

The consideration to prove this statement is the same as for the 6th properties: because we consider both the nested deepness and the data flow while calculating complexity, this becomes true.

4.4 Evaluation

4.4.1 Comparing Exceptions and Other Structured Error Handling

In this Section we will discuss how incorporating exceptions affect the complexity of the code measured by our extended metrics. We show through a simple example that exception handling tends to decrease software complexity compared to older error handling methods. We also analyse an industrial-sized software product, first accounting for exceptions, then without them, thus we show that our extension does affect complexity.

In the following example code snippets, we demonstrate two ways of error handling. First, we show on Listing 20 how one would use an old-style return value based error handling, then we achieve the same functionality with exceptions on Listing 21.

Based on the previous methods for calculating the extended A-V measure for the case with exceptions is 17, while the other error handling method results in 19. This
private val errors = new HashMap<Integer, String>()

def calcNE(x: Int): Int = {
  var tmpX: Int = x
  if (tmpX == 0) {
    0
  } else if (tmpX == 1) {
    -1
  } else {
    tmpX = tmpX - 1
    calcNE(tmpX)
    tmpX
  }
}

def clientNE(): Unit = {
  val number = 15
  val res = calcNE(number)
  if (res == 0) {
    println(s"IllegalArgumentException: \${errors.get(res)}")
  } else if (res == -1) {
    println(s"MyException: \${errors.get(res)}")
  }
}

Listing 20: Scala functions without exceptions.

def calcWE(x: Int): Int = {
  if (x == 0) {
    throw new IllegalArgumentException("Zero");
  } else if (x == 1) {
    throw new MyException("One");
  } else {
    tmpX = tmpX - 1
    calcNE(tmpX)
    tmpX
  }
}

def clientWE(): Unit = {
  val number = 15
  try {
    calcWE(number);
  } catch (Exception e) {
    println(e.getMessage())
  }
}

Listing 21: Scala functions with exceptions.
increase does not seem relevant, but considering the size of the examples, and the
fact that we should also examine the result of the calcNe call on the 12th line, we
arrive at a complexity of 23, which is a 18\% increase, being a considerable amount.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Not measuring exceptions</th>
<th>Measuring exceptions</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>279830</td>
<td>279830</td>
<td>0%</td>
</tr>
<tr>
<td>McCabe</td>
<td>14812</td>
<td>16801</td>
<td>13,428%</td>
</tr>
<tr>
<td>A-V</td>
<td>50438787</td>
<td>52723815</td>
<td>4,5303%</td>
</tr>
</tbody>
</table>

Table 2: Results of analyzing Tomcat.

Apart from synthesized examples, it is worth to exercise our newly created com-
plexity measure on a real-world example. For this purpose, we have created an
analyzer tool based on the Eclipse JDK that calculates McCabe and A-V metrics
(both in original and extended forms) for Java code. We have chosen Java instead
of Scala for simplicity and to have an easier choice of the example project. From
our perspective, Java and Scala behave identically when it comes to handling ex-
ceptions, so this does not compromise our analysis. The tested software was Tomcat
version 6 [17]. The purpose of this analysis is to show that the extensions do increase
calculated complexity. On Table 2, the end results are shown.

The increase in cyclomatic complexity is around 13\% which we would consider
a significant change, and this exemplifies how widely exception handling is used
nowadays. This amount is exceptionally interesting considering that Tomcat is more
of a library, while McCabe’s original constructs were intended for client code. This
possibly leads to the conclusion that Tomcat developers use exception handling
within the framework and also to notify users about errors.

The change in the results of the A-V measure is lower than cyclomatic complexity
at around 4\%. The probable reason for this smaller increase is that regular object-
oriented data usage suppressed what the extension of the metrics increased. This is
in line with our intuition, since data usage for normal control flow should be much
higher than that of error handling. This also describes how exception handling is
used: when there are no other options, we throw one with some information about
the circumstances, but that is negligible compared to normal data usage.

4.4.2 Comparing Exceptions and Monadic Error Handling

We have examined how regular exception-handling code compares to using the Try
monad for error handling by computing the A-V complexity metric. To circumvent
any reliance on an inherent property of A-V we will also calculate the difficulty
and volume metrics defined by Halstead [45]. These examples are meant to cover
idiomatic use cases of various real-world problems. Even though there exist valid
critiques of the Halstead metric, including small sample sizes in the original research,
the possibility of drawing high-level conclusions from simple calculations and the difficulty of implementing the measurement. In our analysis we only use the simpler calculations – complexity and volume – and we only use them as a control measure, we do not draw improbable conclusions based on them.

**Simple Throw-catch**

Let us consider some functions providing some functionality (Listing 22) and its client code (Listing 23).

```scala
1 def f1_e(a: T): T = {
2   if(test(a)) a
3   else throw new IllegalArgumentException
4 }
5
6 def f1_t[T](a: T): Try[T] = {
7   if(test(a)) Success(a)
8   else Failure(new IllegalArgumentException)
9 }
```

Listing 22: Error Handling with try-catch and Try.

```scala
1 val res: Int = try {
2   f1_e(1)
3 } catch {
4     case exception: Throwable => println(exception)
5     default
6 }
7 val res: Int = f1_t(1).recoverWith {
8     case exception => println(exception)
9     default
10 }
```

Listing 23: Client code for Listing 22.

If we compute the A-V metric for the code using exception handling, we get 3, as returning the value or throwing an exception are both under the scope of a condition (1+1) and the difference between the nested deepness of the `throw` and `catch` is again, 1. The Halstead difficulty metric is 8.125 while the volume is 102.19. By using Try, we end up with a slightly more complex code, as we need to wrap either the result or the exception in the monad, but the client code gets simpler, as there are no conditions there. The A-V value here is 4. For the Halstead values we get 8.125 and 81.75 for difficulty and volume, respectively.
Higher-order Functions

In the example on Listing 24, we test multiple inputs with the previous test methods. Since the test methods remained the same, we will focus only on the complexity difference of the client code. Using exception handling, we get complexity of 1 for the caller, as return false has 1, while by using Try, we can keep the complexity at 0, since we have not introduced any conditions. The difficulty metrics values are 9.5 and 5.5 and volume metrics values are 174.23 and 87.57.

```scala
val res: Seq[Boolean] = Seq(1, 2, 3).map { x =>
  try { f1_e(x)
    true
  } catch { case _: Exception => false }
}
val res: Seq[Boolean] = Seq(1, 2, 3).map {
  f1_t(_).isSuccess
}
```

Listing 24: Error handling with higher-order functions.

The example shown on Listing 25 mimics a function that can throw multiple errors (we re-use the exception instances) and the client code (Listing 26) will try to group some inputs to possible outcomes.

```scala
def test_t(a: Int): Try[Boolean] =
  if(a % 2 == 0) Failure(iae)
  else if(a % 3 == 0) Failure(ae)
  else if(a % 5 == 0) Failure(re)
  else if(a % 7 == 0) Failure(nse)
  else if(a % 8 == 0) Failure(npe)
  else Success(true)

def test_e(a: Int): Boolean =
  if(a % 2 == 0) throw iae
  else if(a % 3 == 0) throw ae
  else if(a % 5 == 0) throw re
  else if(a % 7 == 0) throw nse
  else if(a % 8 == 0) throw npe
  else true
```

Listing 25: Testing for multiple errors.

As previously, the tester method has the same complexity in both cases, but the client codes differ significantly: while the Try-case remains at 0, all possible cases of the exceptions need to be manually handled, resulting in a complexity of 5. Here we can calculate 8.125 and 3 as difficulty and 213.97 and 19.65 for volume as per
val range = 1 to 20
val g_t = range.groupBy(test_t)
val g_e = range.groupBy{ x => try {
test_e(x)
} catch {
    case e: ArithmeticException => e
    case e: IllegalArgumentException => e
    case e: RuntimeException => e
    case e: NoSuchElementException => e
    case e: NullPointerException => e
}
}

Listing 26: Client code for Listing 25.

Halstead. The purpose of this example is to show how multiple types of errors affect the code, increasing the difficulty of comprehension. Furthermore, it demonstrates that treating errors as data instead of instructions can be hugely advantageous.

Chaining Computations

Often we have data transformation defined as discrete steps that rely on the outputs of previous transformations. Let’s consider a few transformation steps as depicted on Listing 27.

def f0: T1
def f1(x: T1): T2
def f2(x: T2): T3
def f3(x: T3): T4

Listing 27: Transformation steps.

Each of these methods have their own way of handling invalid input or other erroneous state and they use exceptions to communicate these to the outside world. Using them can be done in a single try-block (as shown on Listing 28).

val res: Either[Throwable, T4] = try {
    val s1 = f0
    val s2 = f1(s1)
    val s3 = f2(s2)
    Right(f3(s3))
} catch {
    case e: Exception => Left(e)
}

Listing 28: Client code for Listing 27.
To let the caller know of the success or the failure of the whole transformation flow, we need some holder object – we have used Either here for the sake of simplicity, but we will not use any of its monadic features. This example can be written with Try in a very similar fashion (Listing 29).

```scala
val res: Try[T4] = for {
  s1 <- f0
  s2 <- f1(s1)
  s3 <- f2(s2)
} yield f3(s3)
```

Listing 29: Client code for Listing 27 using a for-comprehension.

This latter version has smaller complexity by one, as we do not need to manually wrap the failure case. If we were to implement the holder class in the first example as well, the difference would grow further. These code snippets yield 15.75 and 5 for difficulty and 199.7 and 78.14 for volume. The code gets more involved if we want to provide default values in each step. Using exceptions, we do not have much of a choice other than catching each error and manually returning the default value (Listing 30).

```scala
val v1 = try { f0 } catch { case e: Exception => T1Def }
val v2 = try { f1(v1) } catch { case e: Exception => T2Def }
val v3 = try { f2(v2) } catch { case e: Exception => T3Def }
val v4 = try { f3(v3) } catch { case e: Exception => T4Def }
```

Listing 30: Client code for Listing 27 providing default values.

```scala
f0_t.orElse(Try(T1Def))
.transform(f1_t, _ => Try(T2Def))
.transform(f2_t, _ => Try(T3Def))
.transform(f3_t, _ => Try(T4Def))
```

Listing 31: Client code for Listing 27 providing default values using Trys I.

We have several possibilities when using monads though (Listings 31 and 32). As previously, the difference in complexity grows by the number of the cases that we need to cover. In this example, Try has a complexity advantage of 4. In these examples the calculated difficulty is 21.54 compared to 7.5 while volume is 382.37 against 132. If we use chained computations, monads have significant advantages by providing simple and well-defined interfaces to easily wrap different outcomes of these computations. These advantages grow as the number of combinations we need to cover increases.
Asynchronous Code

As we have discussed earlier, the exception handling infrastructure hugely depends on the presence of the execution stack, but running asynchronous code results in not having one unified, contiguous stack. For this reason, we need to introduce code that resolves this issue if we use exception handling. Consider the code snippet on Listing 33, reusing methods from 4.4.2:

```scala
val res = try { ec.submit(new Callable[Int] {
  override def call(): Int = f1_e()}).get() } catch { case e: Throwable => handleError(e) } handleValue(res)
```

Listing 33: Asynchronous code and exceptions.

We submit a task to an `ExecutorService` which will return a `Future` holding the value. In case the method throws an exception, the `get()` call on the `Future` object will re-throw it, thus we wrap that call in a try-catch and handle this case.

```scala
Future { f1_e(1) }.andThen{
  case Success(value) => handleValue(value)
  case Failure(e) => handleError(e)
}
```

Listing 34: Asynchronous code using monadic error handling.

The same functionality using monads seems simpler even at a first glance (Listing 34). Considering only the error handling bits in both cases, the first example has 2 as its complexity, while the monadic version has 1. Calculating difficulty results in 20 and 9.167; volume is 175.93 and 87.6. Here we have seen that composing monads (`Future` with `Try` in this case) can be very powerful and has the potential of greatly reducing the effort needed for human comprehension.
<table>
<thead>
<tr>
<th>Measure</th>
<th>Code</th>
<th>Try-catch (Listings 22 and 23)</th>
<th>Higher-order functions I. (Listing 24)</th>
<th>Higher-order functions II. (Listings 25 and 26)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Handling</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exception Handling</td>
<td>A-V</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Halstead Complexity</td>
<td>8.125</td>
<td>9.5</td>
<td>8.125</td>
</tr>
<tr>
<td></td>
<td>Halstead Volume</td>
<td>102.19</td>
<td>174.23</td>
<td>213.97</td>
</tr>
<tr>
<td>Monad</td>
<td>A-V</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Halstead Complexity</td>
<td>8.125</td>
<td>5.5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Halstead Volume</td>
<td>87.6</td>
<td>87.6</td>
<td>19.7</td>
</tr>
<tr>
<td>Percent Change (Monad over exception)</td>
<td>A-V</td>
<td>133.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Halstead Complexity</td>
<td>100</td>
<td>172.7</td>
<td>270.8</td>
</tr>
<tr>
<td></td>
<td>Halstead Volume</td>
<td>125</td>
<td>198.9</td>
<td>1088.9</td>
</tr>
</tbody>
</table>

Table 3: Summary tables of the computed complexity changes between exceptions and monads.
4.5 Thesis II.

I have investigated how different error handling methods affect software complexity. I have extended two standard complexity metrics – McCabe’s cyclomatic complexity and A-V – to the case of exceptions and evaluated how these metrics measure complexity for exceptions. I have compared traditional error handling methods (using ERRNO and return values) to modern exception handling constructs. I have concluded that exception handling helps reduce software complexity.

I have also analyzed monadic error handling comparing it to using exceptions and through examples I have shown that in general, monadic error handling does not increase complexity while it helps reduce it for non-structured programming constructs.

<table>
<thead>
<tr>
<th>Thesis Title</th>
<th>Relevant Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Improvements to The Type System of Scala</td>
<td></td>
</tr>
<tr>
<td>(2) Effects of Error Handling on Software Complexity</td>
<td>![bullet]</td>
</tr>
<tr>
<td>(3) Scale-free Properties of Software Systems</td>
<td>![circles]</td>
</tr>
<tr>
<td>(4) Read-Copy-Update in Scala</td>
<td>![circles]</td>
</tr>
</tbody>
</table>
Chapter 5

Scale-free Properties of Software Systems

5.1 Motivation

The optimal modularization and using the right level of coupling between components are important for the overall quality of software systems. Although the generic suggestion is to minimize the coupling between modules [112], earlier research on object-oriented programming showed that there is a natural limit to eliminating dependencies between classes [14]. In our research we extend these findings to non-OOP systems and show that this limitation seems to be paradigm-independent. We define coupling in a generic way, stating that two separate entities are coupled if they are connected by some kind of dependency. This still leaves freedom in defining the entities and the dependency relation. From the point of conducting these investigations, any kind of directed connection can be considered a dependency (e.g. one function calls another or one class inherits from another, etc.), and any kind of grouping of pieces of code can be an entity (e.g. function, class, module or package). We focus on a common subset of the artifacts and dependencies of two programming paradigms, namely object-oriented (OOP) and functional paradigms (FP). We have chosen these as they are both separately and independently gaining popularity in scientific and industrial applications [97] and Scala supports the mixture of them.

It is important to note that our analysis considers static and not dynamic calls. Dynamic binding can only happen at run-time, and it can depend on previous program state or user input. Analyzing dynamic calls leads to a completely separate problem space that is out of scope for our investigations.

The evolution of large software systems usually involves significant refactors throughout their lifetimes. This is especially true in agile software development environments, where meeting time-to-market demands drive developing proof-of-concepts
then minimal valuable products. As more business needs are discovered during the
development of a software product, more generalization opportunities emerge. Re-
ducing code duplication and producing reusable components are a continuous need.
It is important to understand what levels of such refactors help the maintainability
of these large software systems. We instinctively expect that there is both a lower
and upper bound for the optimal coupling between modules; our goal is to gain an
understanding of these limits and provide a rigorous analysis and framework that
can be applied to real-world software.

5.2 Related Works

5.2.1 Scale-free Networks

Considerable amount of effort has recently been put into researching the structure
of networks, their transformations over time and properties describing them [83].
Networks that originate in real-world systems rather than follow a mathematically
designed structure have been found to usually follow a degree distribution described
by a power-law distribution [18]. This opposes previous findings of networks gener-
ated by algorithms that tend to have a Poisson degree distribution [36]. Network
properties of large computer programs were investigated for several publicly avail-
able computer programs written in various programming languages including Java,
C, C++, Erlang, etc. [79, 52, 21]. Moreover, the scale-free property was justified for
large scale test systems as well [115].

The Power-law Distribution

Definition 6. Scale-free Network [89]. A graph is called a scale-free network, if its
degree distribution follows a power-law distribution, that is for all nodes in the graph
having $k$ connections to other nodes, the following must be true for large enough $k$
values:

$$P(k) \sim Ck^{-\lambda}$$

(5.1)

where $\lambda$ is typically in the range of $2 < \lambda < 3$ and $C$ is the normalization constant
that needs to satisfy

$$\sum_{n=1}^{\infty} Cn^{-\lambda} = 1$$

(5.2)
Properties of Scale-free Networks

In practice, this describes two key characteristics of these networks:

1. The graph has a set of vertices possessing the number of degrees that is considerably above the mean of the whole network. These nodes are usually called “hubs” of the graph.

2. The clustering coefficient tends to correlate with the node degree: as the degree decreases, the local clustering coefficient decreases as well. This also declares that the coefficient needs to follow a power-law distribution.

The aforementioned properties are thought to be descriptive of how fault-tolerant the network is [30, 29, 23] (if this can be considered regarding the network in question), percolation theory had been used to analyze the fault tolerance of scale-free networks previously. Findings in these studies have identified hubs as both the weaknesses and the strengths of networks. If the graph has mostly low-degree vertices and failures randomly affect nodes, losing a hub is less likely. Although, the fall of a hub can mean severe loss of connectedness of the graph, this effect is still negligible compared to a random graph.

While matching the power-law distribution to a data-set can become resource-intensive, there is a much easier (although less precise) method to analyze empirical data. By transforming the definition of power-law distribution as described in [117], we can arrive to a practical solution to analyze data. After plotting the data-set in the fashion of degrees assigned to frequencies on a log-log graph (that is to have a logarithmic scale on both the $x$- and $y$-axes), this graph should show a linear regression. This method has been shown to be statistically imprecise [28], although it matches our practical requirements, since it is between the error limits.

There is also a need to ignore some points in the data-set, as these tend to corrupt the linearity of the plot. Our definition takes this into consideration, the condition to check *large enough* $k$ values is to avoid noise at the ends of the graph. This is studied and explained in detail in [28].

Lastly, we make an observation on the naming of these networks. Scale-freeness can be observed when one considers nodes at different levels of degree distribution. These nodes tend to “look familiar” at all levels, thus providing the self-similar nature of the network.
5.3 Theoretical Resolution

We have formally defined the metrics we use and proved that they comply with the previously defined properties of coupling metrics (Subsection 2.3.2). The metrics do not depend on the programming paradigm, we only suppose that there are functions, functions are grouped in sets (modules or classes) and function calls are possible among these sets.

Firstly, we define Aggregated Coupling Metric ($ACG$)\(^1\) as the sum of incoming and outgoing function calls to a module: $ACG(m) = ACG_i(m) + ACG_o(m)$. If we have modules $a$ and $b$, and one function call from $a$ to $b$, then $ACG_i(b) = 1$ and $ACG_o(a) = 1$. We also define a variant of $ACG$ that considers the cardinality of function calls: $WACG$; in the case of multiple function calls existing between $a$ and $b$, the number of these function calls will be the result of this metric.

Let us denote a module or class by $m$. Suppose that function $f$ is defined in module $m$. Denote a function call from $f$ to a function $g$ in module $m'$ by $m.f \rightarrow m'.g$. Denote $M$ as the set of modules of the software system. Let us denote $\chi$ a characteristic function that returns 1 if the parameter set is non-empty and 0 otherwise. Furthermore, $\# \#$ will be used as the cardinality of a set.

5.3.1 Definitions

The $ACG$ metric is the sum of the number of incoming calls from other modules and the number of outgoing calls of the module.

**Definition 7.** (Incoming function calls)

$$S_i(m) = \{(f, g) | \exists f \in m, \exists g \in m' : m'.f \rightarrow m.g\} \quad (5.3)$$

**Definition 8.** (Outgoing function calls)

$$S_o(m) = \{(f, g) | \exists f \in m, \exists g \in m' : m.f \rightarrow m'.g\} \quad (5.4)$$

**Definition 9.** ($ACG$ metric)

$$ACG(m) := ACG_i(m) + ACG_o(m) \quad (5.5)$$

where

$$ACG_i(m) = \chi(S_i(m)) \quad (5.6)$$

$$ACG_o(m) = \chi(S_o(m)) \quad (5.7)$$

\(^1\)ACG stands for Aggregated Coupling to avoid the acronym ACM.
Definition 10. (WACG metric)

\[ WACG(m) := WACG_i(m) + WACG_o(m) \]  

where

\[ WACG_i(m) = \#(S_i(m)) \] \hspace{1cm} (5.9)

\[ WACG_o(m) = \#(S_o(m)) \] \hspace{1cm} (5.10)

The following theorem shows the coupling nature of the incoming and outgoing components of metrics based on the coupling properties described in Section 2.3.2.

**Theorem 5.3.1.** \( WACG_i \) and \( WACG_o \) as well as \( ACG_i \) and \( ACG_o \) are coupling metrics.

**Proof.** It is enough to prove for \( ACG_i \) and \( WACG_i \). The proof for \( ACG_o \) and \( WACG_o \) is analogous. We check the five properties of coupling metrics.

1. Trivially true, since both \( \chi \) and \# return non-negative values.

2. \( \text{Outer}(m) = 0 \) implies that there exists no \( g \in n \) so that \( n.g \rightarrow m.f \), hence the resulting set is always empty. It makes both \( \chi \) and \# equal to 0.

3. To prove monotonicity, we need to consider two cases:

   (a) Let us suppose that we add a function call to module \( m \) and \( S_i \) in (7) was not empty. In this case, the number of elements in \( S_i \) increases, but the characteristic functions \( \chi \) and \# return the same value for \( S_i \). This means that \( ACG_i \) and \( WACG_i \) satisfies the non-decreasing requirement of monotonicity.

   (b) In the case of adding such a function call to module \( m \) where \( S_i(m) \) was empty, both \# and \( \chi \) will increase for \( S_i(m) \), thus values of \( ACG_i \) and \( WACG_i \) will be increased as well which means strict monotonicity.

4. We have to prove that \( WACG_i(m) + WACG_i(n) \geq WACG_i(m + n) \) and \( ACG_i(m) + ACG_i(n) \geq ACG_i(m + n) \). Since the proof is similar for both \( ACG_i \) and \( WACG_i \), here we will only list it for \( WACG_i \). We need to consider three possible cases.

   (a) **Disjoint modules.** In this case there exists no module \( p \) such that \( \exists f \in p \land g \in m \mid p.f \rightarrow m.g \) and \( \exists f \in p \land g \in n \mid p.f \rightarrow n.g \) and \( \# f \in m \land g \in n \mid m.f \rightarrow n.g \) or \( n.f \rightarrow m.g \) : simply, there is no module that has a connection to both \( m \) and \( n \) and there are no calls between \( m \) and \( n \). Merging such modules means that the merged module will have exactly
the same calls as \( m \) and \( n \), so \( WACG_i(m) + WACG_i(n) = WACG_i(m + n) \).

(b) Call from a common module. If there is a module \( p \): \( \exists f \in p \land g \in m \mid p.f \to m.g \) and \( \exists f \in p \land g \in n \mid p.f \to n.g \), also \( \exists f \in m \land g \in n \mid m.f \to n.g \) or \( n.f \to m.g \) meaning that there is one module that has a call to both \( m \) and \( n \), but there is no call between \( m \) and \( n \). The merged module will contain the call from \( p \) only once, so its weight will be accounted for only once, hence \( WACG_i(m) + WACG_i(n) > WACG_i(m + n) \).

(c) Call between the modules. If there is no module \( p \) in a way that \( \exists f \in p \land g \in m \mid p.f \to m.g \) and \( \exists f \in p \land g \in n \mid p.f \to n.g \) and \( \exists f \in m \land g \in n \mid m.f \to n.g \) or \( n.f \to m.g \). In this case there is a call from \( m \) to \( n \) (the proof is the same in the reverse order) and there are no common calls from outer modules. Merging such modules will have the same connections as the separate modules except for the call between \( m \) and \( n \), which will be excluded since we do not consider self calls for \( WACG \) and \( ACG \). This leaves us with the same considerations as (4b), thus \( WACG_i(m) + WACG_i(n) > WACG_i(m + n) \).

5. If \( m \) and \( n \) are disjoint, then the condition of 4.a. holds.

The coupling nature of \( WACG \) follows from a more generic theorem.

**Theorem 5.3.2.** The sum of coupling metrics is a coupling metric.

**Proof.** Let \( M_1 \) and \( M_2 \) be coupling metrics. We have to prove that \( M := M_1 + M_2 \) is a coupling metric. We check the five properties of coupling metrics.

1. \( M \) is non-negative, since it is the sum of non-negative numbers.

2. If \( M_1 = 0 \) and \( M_2 = 0 \), then \( M = 0 \).

3. Let \( M_1 \) and \( M_2 \) be monotonously increasing functions on the domain set \( D \). For all \( x, y \in D \), \( x \leq y \) we know that \( M_1(x) \leq M_1(y) \) and \( M_2(x) \leq M_2(y) \). Adding \( M_1 \) and \( M_2 \) together we have \( M_1(x) + M_2(x) \leq M_1(y) + M_2(y) \) and that is equivalent to \( (M_1 + M_2)(x) \leq (M_1 + M_2)(y) \).

4. Merging of two modules can also be derived back to the monotonicity of addition.

5. The equation derives from the monotonicity property as described in Theorem 5.3.1.
Since $WACG$ is the sum of two coupling metrics, it is also a coupling metric. We will use both $WACG$ and $ACG$ to show that the degree distribution derived from these metrics shows a scale-free property.

### 5.3.2 Hypotheses

Since several studies provided proof that real object-oriented software systems follow scale-free properties, we expect that the Scala compiler will also show the same, despite the fact that it is written in a mixed style, using functional and object-oriented paradigms. In other words, combining the functional style (immutable data structures, higher-order functions, algebraic data types) has no (or insignificant) effect on the scale-free property of object-oriented programs.

**Hypothesis 1.** The Scala compiler shows scale-free property for $WACG_i$, $WACG_o$ and $WACG$.

The less complex $ACG$ still grabs an interesting property of a software system. The lack of multiplicity of edges reveals a similar “coupling” order between nodes. We expect that this simpler metric also preserves the scale-free property of the systems under investigation.

**Hypothesis 2.** The Scala compiler preserves the scale-free property with $ACG_i$, $ACG_o$ and $ACG$.

We analyzed three consecutive major versions of the compiler to investigate whether the scale-free property changes over releases. Supposing hypotheses 1 and 2 are true for the first analyzed version, the scale-free property remains and the average change of the metric values are significant (i.e. more than 5%) for all defined metrics.

**Hypothesis 3.** The scale-free property can be observed for all analyzed versions of the Scala compiler.

### 5.4 Practical Approach

We have written a Scala compiler plugin for analyzing the call-graph of the Scala [86] compiler. The main reason for choosing the Scala compiler for our investigations is three-fold: it is written using both functional and object-oriented paradigms, it is one of the biggest freely available Scala code bases and since the authors are experts in Scala, we can consider the language use to be pragmatic. We have gathered the number of incoming function calls for each module (class in this case). The investigated Scala compiler versions are as follows: v2.10, v2.11 and v2.12. The codebase
contains about 430,000 lines of code. The compiler plugin for Scala was inserted as the last compile phase traversing all method calls and registering incoming and outgoing edges for the corresponding classes. The source code can be accessed at [3].

5.5 Evaluation

![Figure 9: WACG distributions for the Scala compiler.](image)

We used log-log plots to visualize the distributions of class dependencies. The charts can be read as the number of classes (y-axis) that have the given number of connections (x-axis). We have fitted a linear regression model (the red line in the charts) to visualize that the number of classes with a given number of connections is close to a straight line. The blue lines represent the average number of connections.

Figure 9. shows the results of the metrics for the Scala compiler v2.10. The number of Scala classes under evaluation is 2227. All of \( WACG_i \) (Figure 9a.), \( WACG_o \) (Figure 9b.) and \( WACG \) (Figure 9c.) show a highly left-skewed distribution. This observation confirms hypothesis 1., the distribution is very similar to ideal power-law distributions. It is common in all charts that the majority of the data points are below average and some of the modules are significantly higher than this average.

The \( ACG \) metrics show a cleaner distribution in the following way: with only a few outliers at the beginning, a steep decline and a long tail can be observed on the charts (Figure 10). Hence we conclude that hypothesis 2. also holds.

The fact that Scala is a multiparadigm language shows that scale-free properties of software systems are agnostic to the specific programming paradigm.

As hypothesis 3 suggests, the scale-free property is preserved throughout the different versions of the compiler. We included only the distributions deriving from the \( ACG \) metrics (Figure 11. and Figure 12.), but the incoming and outgoing distributions show similar results. The source code changes significantly: approximately 50% of the lines in the codebase of the Scala compiler were touched. The changes
Figure 10: ACG distributions for the Scala compiler.

Figure 11: WACG distributions for different versions of the Scala compiler.

in the values of the metrics are significant: there is 5.92% decrease in WACG and 11.09% decrease in ACG. These numbers also support hypothesis 3.

Figure 12: ACG distributions for different versions of the Scala compiler.

Our conclusion is that both ACG and WACG are suitable to capture the scale-free property of the investigated systems. The scale-free property is preserved and significant source code modifications do not alter this property. Since scale-freeness means that some nodes (hubs) have above-average connections, both ACG and WACG are suitable to point out heavily dependent parts of software systems.
Furthermore, the paradigm in which the system is written, is not relevant to the scale-free property.

5.6 Thesis III.

I have investigated ideal coupling levels in large software systems. I have defined two software metrics and proved they are coupling metrics. I have shown that these metrics would display scale-free properties on the Scala compiler and I have also shown that the scale-free property is kept throughout major versions of codebase of the Scala compiler.

<table>
<thead>
<tr>
<th>Thesis Title</th>
<th>Relevant Publications</th>
</tr>
</thead>
</table>
Chapter 6

Read-Copy-Update in Scala

6.1 Motivation

Ever since we have started approaching the practical limits of single-threaded CPU performance, concurrent programming has been in the focus of researchers and industry parties. Concurrent programming with shared mutable state is universally considered hard [114]. There are many reasons for this, starting from the numerous abstractions designed to solve specific problems that do not work well together, the unpredictability of scheduling the concurrent modifications, to the complexity of reasoning about temporality of concurrent software. New approaches and concurrent programming models are being invented in hopes of reducing the complexity for software developers.

6.2 Related Works

6.2.1 Amdahl’s Law

Amdahl’s law [16] is a formula describing the correlation between the theoretical speed gain of a system under a specific work load and the improvement of said system’s resources. It is often used to estimate performance gains of multiple processors in parallel computing. Amdahl’s law has been formulated [100] as seen on Figure 13, where $S_{\text{latency}}$ is the estimated speedup, $s$ is the speedup of the part that benefits from the resource improvements and $p$ is the original proportion of execution time without the improvements.

$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

Figure 13: Amdahl’s law formulated.
We also have to consider the parts of the system that do not benefit from the improvements. A good example is a software system whose task can be divided into two categories: one that does I/O operations, the other runs computations. In this case, adding multiple processor cores will not benefit the I/O parts, but will theoretically increase the performance of the concurrently ran computations.

### 6.2.2 Scalability

Amdahl’s law describes the theoretical maximum improvement in a system’s speed, but in typical real-world usage, even the most sophisticated approaches will have worse scaling. The reasons are various and arise from each concurrency model’s fundamental properties. If we consider the actor model [48], one of its main characteristics is supporting messaging-based communication between actors that do not share any data. This results in copying data when actors communicate and if the actors do not share the same address space (because, for example they run on separate machines), we also have to consider network overhead.

Looking at possibly the most frequently used concurrency model using shared mutable state between concurrently executing parts of a program, we have to make sure that each thread has a consistent and correct view of the shared mutable state. For this reason, we have to **mutually exclude** modifications. To support this, we have several possible primitives, including locks, mutexes and semaphores, but using these constructs correctly is a non-trivial problem. For this reason, we have to increase the abstraction level of these for typical use-cases.

We have evaluated the **read-copy-update** locking mechanism in Scala. The original implementations of RCU provide low-level APIs that require caution to avoid any concurrency issues, including data races or deadlocks. Our proposed API matches Scala’s idioms by providing a higher-level abstraction while it does not sacrifice performance.

### 6.2.3 Read-Copy-Update

A non-trivial synchronization approach has been introduced to the Linux kernel in 2002 called read-copy-update (RCU) [72, 71], although similar techniques have been previously used for garbage collection algorithms [58], CPU TLB implementations [99], concurrent systems with hard real-time requirements [55] and many others. RCU aims to alleviate synchronization overhead for situations when there are more read than write operations. It allows this by not blocking reads; if a write happens while there are readers, the write operation will be applied to a new memory address, keeping all readers with an intact state of the underlying data structure. When the write operation finishes, new readers will be pointed to the new memory
address. After all readers of the stale data finish reading, its memory space can be freed up.

Figure 14: RCU operations example.

On Figure 14 we have displayed an example of RCU-backed concurrent reads and writes. At first we have the initial version of the data (a). Later a reader (b) then another one start (c) reading the data. While they are executing their reads, a writer starts updating the data by copying the initial version (d) – but this does not block the existing readers: a third one joins (e). In the meanwhile, the first two readers finish their operations (f), but the third reader still refers to the old version. As the next step, the writer finishes updating the memory contents (g), so when the next reader comes along, (h) it is pointed to the new version of the data. The last reader of the old data then completes (i), meaning this memory can be discarded (j).

RCU is an example of space-time trade-off where memory space is traded for performance gains: neither readers will block writers, nor a writer will block readers until it finished a possibly expensive operation. RCU does not provide conventional temporal mutual exclusion, but its handling of concurrent data access is based on spaciality: readers and writers are separated on the memory space, henceforth they can operate even while overlapping in time.

Numerous implementations can be found in various operating systems and even a user-space library is available [123]. The C-based implementation in the Linux kernel provides a fairly low-level API that can be mapped to the following operations or methods:

- **rcu_read_lock().** Marking the beginning of a read operation so we know when this data can be freed up if it has been update by a write.

- **rcu_read_unlock().** Signaling that a read operation has finished.
• `synchronize_rcu()`. Blocks until all *pre-existing* read operations finish. In some implementations this method can be called with a callback that will be applied when the reads have finished instead of blocking.

• `rcu_assign_pointer()`. Writers use this method to update the underlying data managed by RCU.

• `rcu_dereference()`. Readers can access the RCU-guarded data via this method.

Based on the short description of this API, it is easy to see that using a read-copy-update implementation is a non-trivial task and needs special attention from its users. To help developers avoid typical pitfalls, an implementation with a higher-level API has been completed in C++ that uses some of the new features of C++14 [69].

### 6.3 Theoretical resolution

Formalizing correctness proofs of concurrent algorithms requires a rigorous mathematical model that can help express the temporality of data reads and writes. There have been a handful of mathematical models developed for such uses, including the temporal extension of formal logic systems [35]. We have chosen to use the *happens-before* relation for our analysis as it provides an easy-to-understand framework while assures rigorosity.

#### 6.3.1 The happens-before relationship

The happens-before relationship establishes a guarantee that the result of an operation performed by a thread is visible to an operation in another thread. For our intents and purposes, it is enough to consider reads and writes to memory, but one can extend the model to other operations and types of results. Happens-before defines a strict partial ordering of operations in the program. The rules listed here are formulated at the level of the language running on the JVM, thus it is the responsibility of the JVM to hide any internal implementation detail – be it related to memory management, garbage collection or anything platform-specific.

**Single-threaded rule**

Every operation in a single thread happens-before any other operation defined later in the program order.

If we consider the operations on Listing 35, we can define the happens-before relationship as follows: $operation_1 \xrightarrow{\text{happens-before}} operation_2 \xrightarrow{\text{happens-before}} operation_3$. 
```
def singleThreaded() = {
  <operation1>
  <operation2>
  <operation3>
}
```


**Monitor lock rule**

Unlocking a monitor happens-before every subsequent lock on the monitor. A monitor can be expressed in Scala by various language constructs, on Listing 36 we give the example of the `synchronized`-block. Let’s assume that the execution of `threadA` reaches `synchronized(object1)` first resulting in `threadA` acquiring the lock on `object1`. In this case, `threadB` will not be able to acquire the lock until the execution of `threadA` reaches the end of the synchronized block on line 4. After this is complete, `threadB` can acquire the lock and start executing the operations in its own `synchronized`-block.

```
def threadA = {
  synchronized (object1) {
    ...
  }
}
def threadB = {
  synchronized (object1) {
    ...
  }
}
```

Listing 36: Synchronized-block happens-before relation.

A thread must also first acquire the monitor, before releasing it. The roles of each thread can be changed. The following happens-before ordering can be formalized:

```
threadA : monitor.lock happens-before threadA : monitor.unlock
```

**Volatile rule**

Scala’s main runtime environment, the JVM supports a more lightweight synchronization construct than monitors in the form of `volatile` variables. In Scala, this feature can be accessed through the `@volatile` annotation. Using a volatile variable ensures that every write operation to the variable will be completely visible to every read operation in any other thread that was started after the write operation started. The reason for supporting such a low-level synchronization construct is that it has hardware support on many platforms, making it very efficient. Volatile variables naturally only support primitive values, such as integers, doubles or references.

The example on Listing 37 shows a typical use case of volatile variables: an integer with a static lifetime is shared between two threads. It also shows a possible common
object O {
    @volatile var sharedVolatile: Int = 0
}

def threadA = {
    ...
    O.sharedVolatile = 3
}

def threadB = {
    O.sharedVolatile =
    O.sharedVolatile + 1
}

Listing 37: Volatile happens-before relation.

pitfall of such low-level synchronization constructs: in threadB the value is both read and written, but these two operations are not guaranteed to be atomic. The right-hand side of the assignment will read the value, but before the assignment actually happens, another thread can very well overwrite the value, then the assignment will take place, losing the intermediate value. This is completely correct from the perspective of the volatile variable. For reads in any thread and writes started before the read, volatile assures only the following: volatile.write \(\xrightarrow{\text{happens-before}}\) volatile.read.

Thread start rule

On the JVM, the basic concept of concurrent execution is a thread. A thread can be spawned by other threads and the new thread must be able to observe the results of every operation that happened before on the spawning thread. Listing 38 displays this requirement.

def threadA = {
    val t = new Thread () {
        override def run ():
            Unit = threadB ()
    }
    t.start ()
}

def threadB () = {
    ...
}

Listing 38: Volatile happens-before relation.

The thread start rule states that threadB.start \(\xrightarrow{\text{happens-before}}\) operations in threadB. As the result of the single-threaded rule, this also implicitly states that threadB observes the results of all operations that happened-before in threadA.

Thread join rule

Besides the creation of threads, we also need to be able to express requirements for the end of threads. A thread that spawns another thread can join on the spawned thread, meaning it will wait for the operations in the spawned thread to complete.
Furthermore, we also require that all operations in the spawning thread after the spawned thread is joined will be observable. Listing 39 illustrates this requirement.

```scala
def threadA = {
  val t = new Thread() {
    override def run(): Unit = threadB()
  }
  t.start()
  ...
  t.join()
  <operation1>
}
```

```
def threadB() = {
  ...
}
```

Listing 39: Volatile happens-before relation.

We can formalize this as $\text{operations in threadB} \xrightarrow{\text{happens-before}} \text{threadB.join}$ $\xrightarrow{\text{happens-before}} \text{operations in threadA after threadB.join}$.

Finally rule

Exception handling can introduce non-local gotos into structured code that can lead to various resource-management problems, including ones related to synchronization constructs. On Listing 40 the resource the code is leaking is a lock. Since the method `leaksResources` throws an exception, control never gets to unlock the lock that was acquired before, so all later attempts done by other threads will wait indefinitely causing a classical deadlock.

```scala
object Deadlock {
  Lock l = new Lock

  def leaksResources = {
    l.lock
    throw new RuntimeException
    l.unlock
  }

  def willTryToLock = {
    l.lock
  }
}
```

Listing 40: Deadlock caused by throwing an exception.

To resolve such problems, the JVM has the concept of `finally-blocks` that are guaranteed to be executed. They always need to accompany a try-block and the JVM will make sure they are executed no matter how the execution completes in
the try-block. The code using a finally-block resolving the previous issue can be seen on Listing 41.

```java
object Deadlock {
    Lock l = new Lock

    def leaksResources = {
        l.lock
        try {
            throw new RuntimeException
        } finally {
            l.unlock
        }
    }

    def willTryToLock = {
        l.lock
    }
}
```

Listing 41: Possible deadlock resolved with using a finally-block.

Finally-blocks provide a powerful guarantee that can be leveraged to work around complex problems. The happens-before relation for them can be expressed as $finally-block \xrightarrow{happens-before} try-block returns control$.

**Transitivity rule**

It feels natural, but has to be explicitly stated that happens-before is transitive: $A \xrightarrow{happens-before} B \land B \xrightarrow{happens-before} C \Rightarrow A \xrightarrow{happens-before} C$

**Irreflexivity rule**

Any operation in any given program cannot observe its own result: $\forall a \in \{operations\} : (a,a) \not\in \xrightarrow{happens-before}$

**Asymmetry rule**

Happens-before is a one-way relation: $\forall (a,b) \in \{readers\} : (a,b) \xrightarrow{happens-before} \Rightarrow (b,a) \not\in \xrightarrow{happens-before}$

### 6.3.2 Guarantees of a read-write lock

A read-write lock allows unlimited simultaneous access to readers, while guarantees that writes are mutually exclusive. Because of this, there is no happens-before
relation between reads, only between writes. The guarantees can be formalized as follows:

1. Reads can be run simultaneously. \( \forall (a, b) \in \{\text{readers}\} : (a, b) \not\in \text{happens-before} \)

2. All reads finish before writes. \( \forall \text{readLock} \in \{\text{readers}\} : \text{readLock.release} \xrightarrow{\text{happens-before}} \text{writeLock.acquire} \)

3. All writes are mutually exclusive. \( \text{writeLock.release} \xrightarrow{\text{happens-before}} \text{writeLock.acquire} \)

6.3.3 Guarantees of RCU

The RCU implementation is similar to a read-write lock in the sense that it allows unlimited simultaneous reads, while the locking scheme guarantees mutual exclusion for write operations. The main difference is that RCU continues to allow reading the old data while the write operation is in progress. This is extremely useful for cases when writes can take a significant amount of time. This is the reason why we need to distinguish between the beginning of a write operation and its final step, updating the underlying data reference – we will refer to these as \textit{writes} and \textit{reference updates}. Our implementation (as seen on Listing 44) further guarantees that write operations are serialized: writes initiated in a later point are guaranteed to observe the results of previous writes. This consideration further simplifies the usage as clients do not need to synchronize their writes. This leads us to the following guarantees:

1. Reads can be run simultaneously. \( \forall (a, b) \in \{\text{readers}\} : (a, b) \not\in \text{happens-before} \)

2. Write operations can be run in parallel with reads. \( \forall a \in \{\text{readers}\}, b \in \{\text{writes}\} : (a, b) \not\in \text{happens-before} \)

3. Reference update finishes before new reads start. \( \forall \text{read started after} \in \{\text{readers started after}\} : \text{referenceUpdate} \xrightarrow{\text{happens-before}} \text{read} \)

4. All writes are mutually exclusive. \( \text{writeA} \xrightarrow{\text{happens-before}} \text{writeA.referenceUpdate} \)

6.4 Practical Approach

As Scala provides high levels of abstraction, one of the goals of our read-copy-update implementation was to hide the low-level details from the users. This would require us to handle all logic related to the internal locking mechanisms of RCU, but in the meanwhile support all use-cases without performance penalties. We have also been trying to provide an idiomatic Scala API that borrows patterns from similar holder
objects in the standard library, such as `Option[A]` [106]. This would allow users to easily understand a familiar API and would prevent them from implementing concurrency-related bugs.

### 6.4.1 The RCU API

We show the proposed API of our RCU implementation on Listing 42. This API covers the operations in other RCU implementations, but hides the low-level locking mechanisms needed for correctness. The `get()` and `set()` methods cover the basic reference operations of getting and setting values of the guarded memory location. To update the RCU data structure, users need to provide a modifier function that receives the current state as its input and produces the new data. This passed-in function or lambda represents the actual `write` operation and the `modify()` method handles all necessary logic internally. `map()` is a simple convenience function that acts as a reader and transforms the data.

```scala
trait ReadCopyUpdate[A] {
  def modify(modifier: A => A): A
  def get(): A
  def map[B](f: A => B): B
  def set(value: A): Unit
}
```

Listing 42: RCU Scala API.

### 6.4.2 The Naïve Implementation

The naïve version uses an `AtomicReference` for the backing data since it provides lock-free reference updates. Writes themselves need to be synchronized among each other since a new write operation can be requested while another one is still working on modifying the underlying data. To prevent data collisions or overwriting using an older version at a later time, the complete write operations need to have a mutual exclusion. Setting the reference can happen outside of the write synchronization as the atomicity is guaranteed by `AtomicReference`.

### 6.4.3 Using Read-write Locks

We have also implemented RCU using a more advanced, ReadWriteLock-based locking scheme. The only difference compared to the naïve version is that we use two different locks for reading the reference and setting it. Theoretically this should favor readers even more.
trait ReadCopyUpdateNaive[A] extends ReadCopyUpdate[A] {
  protected val data: AtomicReference[A]
  private val modifierLock = new AnyRef

  def modify(modifier: A => A): A = {
    val newData = modifierLock.synchronized {
      modifier(data.get())
    }
    data.set(newData)
    newData
  }

  def get(): A = data.get()
  def map[B](f: A => B) = f(get())
  def set(value: A): Unit = data.set(value)
}

Listing 43: RCU using basic locking logic.

trait ReadCopyUpdateRWL[A] extends ReadCopyUpdate[A] {
  protected var data: A
  private val l = new ReentrantReadWriteLock(false)
  private val rl = l.readLock()
  private val wl = l.writeLock()
  private val mLock = new AnyRef
  def modify(modifier: A => A): A = mLock.synchronized {
    val newValue = modifier(data)
    try {
      wl.lock()
      data = newValue
      newValue
    } finally { wl.unlock() }
  }

  def get(): A = try {
    rl.lock()
    data
  } finally { rl.unlock() }
  def map[B](f: A => B) = f(get())
  def set(value: A): Unit = try {
    wl.lock()
    data = value
  } finally { wl.unlock() }
}

Listing 44: RCU using read-write locks.
6.4.4 Correctness of the Read-Write Lock RCU

As we have seen in Section 6.3.3 there are certain guarantees the abstract RCU locking scheme provides. We will show that our implementation on Listing 44 meets these requirements.

1. $\forall (a, b) \in \{\text{readers}\} : (a, b) \not\in \text{happens-before}$
   
   Since reads are guarded by the underlying ReentrantReadWriteLock's read lock implementation, this is trivially true.

2. $\forall a \in \{\text{readers}\}, b \in \{\text{writes}\} : (a, b) \not\in \text{happens-before}$
   
   As lines 8 and 9 show, the modifier operation is not synchronized by the ReentrantReadWriteLock, meaning that while this operations is running, neither reads or writes are synchronized.

3. $\forall \text{read} \in \{\text{readers started after}\} : \text{referenceUpdate} \xrightarrow{\text{happens-before}} \text{read}$
   
   Reference updates are synchronized by the ReentrantReadWriteLock's write lock on lines 11 and 32. Since the ReentrantReadWriteLock originally guarantees that read and write operations are mutually exclusive, this means that reference updates will be mutually exclusive to reads in our RCU implementation. Since the lock is in a non-fair mode, the write lock will be acquired as soon as possible.

4. $\text{writeA} \xrightarrow{\text{happens-before}} \text{writeA.referenceUpdate} \xrightarrow{\text{happens-before}} \text{writeB}$
   
   Write operations are serialized using a monitor. This is the reason why reads can still be running while writes are in progress.

The implementation is free of deadlocks for two reasons: it uses finally-blocks for every explicit unlock operation and there is only a single place of locking while already holding a lock, thus there is no way of locking in the reverse order.

6.5 Evaluation

To evaluate our read-copy-update class, we have implemented one the most fundamental data structures used in software, the HashMap in a thread-safe way. Both the Java and Scala standard libraries provide a version of this collection, helping us compare both the ease-of-use and the performance of the proposed solution. We have analyzed the implementations of the two basic operations of a HashMap – get() and += – between the standard version and our RCU-based implementation. We will also discuss the performance characteristics of these two different versions.
6.5.1 Concurrent HashMap in Scala

Scala’s concurrent HashMap is a wrapper around the Java java.util.concurrent.ConcurrentHashMap<K, V> class, providing a Scala-like interface. Converting the Java implementation is fairly simple based on the decorators found in scala.collection.convert.decorateAsScala: one can simply call new ConcurrentHashMap[K, V]().asScala to get a scala.collection.concurrent.Map object.

ConcurrentHashMap provides a lock-free retrieval operation while writes never lock the whole internal table. This is achieved by introducing segments to the entry array, where each segment represents a range of the possible hash values. When write operations happen simultaneously, in ideal cases they will only touch different segments, avoiding lock contention. This of course might not be the case in real-world applications. The number of segments – or as the class documentation refers to it, the level of supported concurrency – can be controlled from client code, providing the possibility of fine-tuning performance characteristics of the class.

6.5.2 RCU-based Concurrent HashMap in Scala

Our concurrent.Map is based on the non-thread-safe Java HashMap class [91]. Most of the considerations put into the implementation details of that class are copied over to our version, including bucket implementation and hash-distribution. The main difference is that our hash table is now a ReadCopyUpdate instance, and both HashMap reads and writes will be delegated through the RCU API. We have listed the code snippets for both of these methods for reference.

Comparing these methods to the original ones reveals how simply one can convert single-threaded code to use RCU. Furthermore, since all locking-related logic is handled in the ReadCopyUpdate class, the readability of the client code does not decrease significantly, allowing developers to focus on the domain problem.

6.5.3 Benchmarks

We have used JMH [92] as our benchmark harness to reduce possible variability to the minimum. The test code concurrently reads and writes to the map object using multiple threads. Since the goal was to benchmark RCU performance, we were focusing on cases where there is minimal hash collision, resulting in a flat bucket structure. This allows us to test the characteristics of the RCU data structure.

The test machine had 80 cores of Intel Xeon E5-2698 v4 2.20 GHz with 512 gigabytes of memory. The benchmarked method creates 50000 Future instances, of which a varying number will be writers, the rest of the operations will be readers and it will block until all Futures complete.
```scala
protected val table: ReadCopyUpdate[Array[Entry[K, V]]]

override def +=(kv: (K, V)): RcuHashMap.this.type = {
  val (key, value) = kv
  table.modify{ t =>
    val index = indexFor(hash(key.hashCode()), t.length)
    val entry = t(index)
    if(entry != null) {
      entry.addNext(key, value)
      noOfElements += 1
    t
    } else {
      val res = addEntryToTable(t, key, value, index)
      noOfElements += 1
      res
    }
  }
  this
}

private def addEntryToTable(t: Array[Entry[K, V]], key: K, value: V, index: Int): Array[Entry[K, V]] = {
  val tableLength = t.length
  val array = if((tableLength * loadFactor) <= noOfElements) {
    val newArray = new Array[Entry[K, V]](t.length * 2)
    Array.copy(t, 0, newArray, 0, tableLength)
    newArray
  } else { t }
  array(index) = new Entry[K, V](key, value)
  array
}
```

Listing 45: RCUHashMap += implementation.

```scala
override def get(key: K): Option[V] = {
  val entry = table.map{t => t(indexFor(hash(key.hashCode()), t.length))}
  if(entry != null) entry.get(key)
  else None
}
```

Listing 46: RCUHashMap get() implementation.
6.5.4 Results

On Figures 15 and 16 we show how Java’s ConcurrentHashMap implementation compares to the RCU-based one. We have run these benchmarks with a fixed number of threads and changed the ratio of read to write operations. The Y-axis shows values of ms/op, while the X-axes show that every \( n \)th operation is a write.

![Figure 15: Benchmark results I.](image)

As the charts show, the performance of our implementation is comparable to ConcurrentHashMap’s. RCU proves to be usually faster when read operations dominate, and it usually keeps up pace even when write operations become more frequent.

![Figure 16: Benchmark results II.](image)

We have also tested how well RCU scales if we increase the worker threads. These results can be found on Listing 16. Here, on the X-axis we displayed the number of threads with every 8000th operation being a write.

There is no realistic way of eliminating all variance of real-world benchmarks; as our data show, we have observed some discrepancies in our measured values, even
though we used JMH’s warm-up capabilities as well as calculated average results over 15 iterations of the benchmarked method. We have also calculated the standard descriptive statistics properties of our benchmark results and displayed them on Table 4.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>4 Threads</th>
<th>32 Threads</th>
<th>64 Threads</th>
<th>8K Writers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CMap</td>
<td>RCU</td>
<td>Diff</td>
<td>CMap</td>
</tr>
<tr>
<td>Mean</td>
<td>97.14</td>
<td>100.67</td>
<td>-3.53</td>
<td>65.13</td>
</tr>
<tr>
<td>Std Error</td>
<td>0.91</td>
<td>1.01</td>
<td>1.64</td>
<td>3.42</td>
</tr>
<tr>
<td>Median</td>
<td>97.18</td>
<td>100.66</td>
<td>-3.46</td>
<td>63.12</td>
</tr>
<tr>
<td>First Quartile</td>
<td>95.32</td>
<td>98.92</td>
<td>-3.53</td>
<td>57.12</td>
</tr>
<tr>
<td>Third Quartile</td>
<td>99.04</td>
<td>103.57</td>
<td>0.72</td>
<td>72.27</td>
</tr>
<tr>
<td>Variance</td>
<td>9.10</td>
<td>11.32</td>
<td>29.59</td>
<td>128.79</td>
</tr>
<tr>
<td>Std Deviation</td>
<td>3.02</td>
<td>3.36</td>
<td>5.44</td>
<td>11.35</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>-0.14</td>
<td>-0.06</td>
<td>-0.84</td>
<td>-0.77</td>
</tr>
<tr>
<td>Skewness</td>
<td>0.13</td>
<td>-0.39</td>
<td>-0.06</td>
<td>0.44</td>
</tr>
</tbody>
</table>

Table 4: Descriptive statistics of the benchmark results.

6.6 Thesis IV.

I have implemented the read-copy-update locking scheme in Scala providing a high-level API that fits into the standard library of the language. As a validation of the RCU method, I have used the happens-before relation to prove the soundness of the implementation. To analyze the performance characteristics of the approach I have created a concurrent hash map implementation using my RCU API and investigated how it compares to the one in the Java’s standard library. It has comparable performance, so I have concluded that RCU is a viable locking strategy even in high-level language constructs.

<table>
<thead>
<tr>
<th>Thesis Title</th>
<th>Relevant Publications</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Improvements to The Type System of Scala</td>
<td>○ ○ ○ ○</td>
</tr>
<tr>
<td>(2) Effects of Error Handling on Software Complexity</td>
<td>○</td>
</tr>
<tr>
<td>(3) Scale-free Properties of Software Systems</td>
<td>○ ○</td>
</tr>
<tr>
<td>(4) Read-Copy-Update in Scala</td>
<td>• •</td>
</tr>
</tbody>
</table>
Chapter 7

Summary

In this dissertation we have investigated how different programming paradigms are represented in Scala. After creating the theoretical foundations, we have investigated four different areas of interest.

7.1 Improvements to The Type System of Scala

Summary

As part of the many advanced typing-related features, Scala supports omitting certain code elements that can be deducted by the compiler. The two main aspects of this feature are type inference and implicits.

The type inference algorithm of Scala is one of its highly-praised features. It helps programmers reduce clutteredness of their code by letting them omit tedious type declarations. Due to Scala’s object-oriented features, the type inference algorithm is not a simple implementation of the Hindley-Milner algorithm used in many functional languages. Partly due to this reason, Scala does not support type inference for recursive definitions. This goes against the goals of Scala and exposes unnecessary constraints on the users of the language. Furthermore, there can be cases when programmers do not define types of these recursive definitions correctly, leading to potential bugs. We have looked at how humans solve this problem and applied the same heuristics to the inference algorithm. Although more testing would be needed before merging the changes to the Scala compiler’s main line, we have created a fully working implementation.

Using implicits can help with reducing code repetition, increasing readability and allowing programmers to write terser code and more natural domain-specific languages. Due to the wide range of features in the compiler, adding support for implicits had a significant effect on the performance. This degradation can lead
to great frustration for day-to-day users of the language. We have analyzed the performance characteristics of the compiler through various methods and applied different approaches to alleviate the problem. The analysis we have conducted was originally requested by a large industry party exactly because of the aforementioned frustrations. The results we have presented here are still being used daily and the attention of every programmer working on the large Scala codebase is drawn to these findings.

7.1.1 Thesis I.

I have analyzed how the type system of Scala supports providing succinct and elegant solutions to problems and I have found two areas of possible improvement.

Firstly, I have created a novel approach to type simple recursive functions. I have proved its correctness by defining a new formal language based on typed lambda-calculus and used typing rules to prove the correctness of the theorem. I have implemented this type inference algorithm as an extension to the Scala compiler and tested the implementation on various cases.

Furthermore, I have investigated how implicit resolution affects the performance of the compiler. I have created a specialized compiler plugin to make meaningful measurements and created a test suite to generate synthetic code to test my hypotheses. I have worked out easy-to-follow guidelines for programmers that help reduce the potential errors and the time compilation takes when using implicits.

7.2 Effects of Error Handling on Software Complexity

Summary

Although handling errors is a key aspect of creating robust software, its effects on software complexity had not been widely investigated. There are several methodologies developed for error handling, starting from the most trivial ones, such as using return values or GlibC’s ERRNO to the more modern exception handling. With the advent of functional programming constructs, a new way of handling exceptional cases was added to the generic toolbox: using monads. There had been many unanswered questions around these error handling methods and how they affect code complexity. I have analyzed and measured them with scientific rigour.

One of my publications on this topic has been cited by [81].
7.2.1 Thesis II.

I have investigated how different error handling methods affect software complexity. I have extended two standard complexity metrics – McCabe’s cyclomatic complexity and A-V – to the case of exceptions and evaluated how these metrics measure complexity for exceptions. I have compared traditional error handling methods (using ERRNO and return values) to modern exception handling constructs. I have concluded that exception handling helps reduce software complexity.

I have also analyzed monadic error handling comparing it to using exceptions and through examples I have shown that in general, monadic error handling does not increase complexity while it helps reduce it for non-structured programming constructs.

7.3 Scale-free Properties of Software Systems

Summary

Coupling between software components has been a main consideration when designing complex systems. The general consensus is that the lower the coupling, the better. Unrelated to software metrics, there have been investigations to many properties of networks that can be found in nature. A key property of such systems is scale-freeness. This property describes that there is a natural organization to these networks, and it is displayed by the coupling between the nodes in these networks. Most of the nodes have a low number of connections to other nodes, while there is a small number of nodes – called hubs – with many connections. These hubs represent both the strengths and weaknesses of these networks.

We have theorized that the same properties of natural networks may be applicable to software systems and that these properties are invariant to large-scale systems. Furthermore, we have analyzed software implemented not only in different languages, but different paradigms to show the universal nature of scale-freeness.

7.3.1 Thesis III.

I have investigated ideal coupling levels in large software systems. I have defined two software metrics and proved they are coupling metrics. I have shown that these metrics would display scale-free properties on the Scala compiler and I have also shown that the scale-free property is kept throughout major versions of codebase of the Scala compiler.
7.4 Read-Copy-Update in Scala

Summary

Concurrent programming is considered one of the hardest problems in computing. While humans can fairly easily argue about correctness of single-threaded applications even through multiple levels of abstractions, most of us have a hard time to write bug-free concurrent code. We can avoid this problem less and less though. There is an increasing demand to solve computationally intensive problems through software implementations and we are approaching the practical limits of single-threaded performance of hardware. For these reasons, the generic attention is drawn away from low-level synchronization constructs to higher-level approaches. One of these is read-copy-update, where we exchange temporal overlapping of data for spatial overlapping. With RCU, we can effectively solve problems where reading data happens orders of magnitudes more frequently than writing.

RCU has so far been only implemented as a low-level library that is hard to access. It has a low-level API that requires disciplined usage. I wanted to investigate if this is inherently required or if RCU is a capable locking strategy even in environments with higher-levels of abstraction.

7.4.1 Thesis IV.

I have implemented the read-copy-update locking scheme in Scala providing a high-level API that fits into the standard library of the language. As a validation of the RCU method, I have used the happens-before relation to prove the soundness of the implementation. To analyze the performance characteristics of the approach I have created a concurrent hash map implementation using my RCU API and investigated how it compares to the one in the Java’s standard library. It has comparable performance, so I have concluded that RCU is a viable locking strategy even in high-level language constructs.
My Publications


My Lectures


Bibliography


[124] Todd L Veldhuizen. “C++ templates are turing complete”. In: Available at citeseer. ist. psu. edu/581150. html (2003).


[130] Why does this implicit function fall into stackoverflow? 2013. URL: https://groups.google.com/forum/#!topic/scala-language/RejIFN5bLrA.


Appendices
A.1 Multiparadigm Programming Language
Elements in Scala

Scala *(SCAlable LAnguage)* is a general-purpose, multiparadigm programming language that supports object-oriented and functional constructs [84]. It has a strong, static type system with features coming from functional languages, such as type inference, functions being first-class citizens and immutability. Other functional properties include lazy evaluation, statements as expressions, currying, pattern matching, as well as implicit values and functions and support for closures and algebraic data types. Scala is an object oriented language as well, since all Scala values are objects. Data types and their behavior are described by traits and classes with similar semantics to other object-oriented languages. Based on these characteristics Scala is known for its terseness, its support for domain-specific languages *(DSLs)* and being a language that can support solving a wide variety of problems. Its compiler is open-source that natively supports extensions through macros and compiler plugins. In this Chapter we will discuss some of the main features of Scala.

A.1.1 A Unified Type System

Due to the functional aspects of Scala, all values have types, including objects, numerical and String primitives as well as functions and expressions. On Figure 17 we present the complete type system of the language.

![Figure 17: The unified type system of Scala.](image)

Compared to Java, the main difference is that there is no distinction between primitives and reference types, all of them are included in the hierarchy. The top type is *Any* which means that all types in Scala programs will extend it, while at the bottom there is *Nothing*, which in turn is a subtype of all – including user-defined ones as well. Immutable values of types are created with the *val* keyword, while using *var* will declare them mutable.
A.1.2 Classes and Traits

Type hierarchies in Scala are built using classes, objects, abstract classes, traits and a special abstraction, case classes (we will discuss case classes separately). Classes and abstract classes are similar to other object-oriented languages, with one notable difference. Each class must exactly have one class constructor – if the programmer has not defined one, the compiler will generate it. An example for a class constructor can be seen on Listing 47. Programmers can define constructor overrides – using the this() method –, but all of them will have to eventually call the class constructor. Values or variables defined in the class constructor will behave like regular fields of the class, but they will also define the constructor’s signature.

Scala supports single inheritance, thus a class can only extend one other class or abstract class. Traits are class-like entities, but no instances of them can be instantiated and they cannot define constructors. They can contain field and method declarations and definitions as well and more than one can be mixed into any class. To resolve the classical diamond inheritance problem [80], Scala applies type linearization: it will flatten type trees by walking them in a specific order: from right to left as they are listed in the class definition. All hidden fields and methods are still available to use in the class by specifying the exact type using this[TypeName]. One can also create final and sealed classes and traits. Final classes cannot be extended further, while sealed ones will have to have all their subtypes defined in the same file.

Instances of Objects are entities with a static lifetime; they cannot be instantiated and will be loaded by the class loader at the occurrence of the first reference to them.

A.1.3 Case Classes

Case classes are similar to regular classes, but they provide an easy-to-use value type-like interface. The compiler will automatically generate the following methods for case classes based on their class constructors: getters and setters, hashCode, equals, copy, toString, apply and unapply (the last two are used with pattern
matching). These methods will behave as one would expect from a value-like object. Similarly to regular classes, one can also define case objects, which will have the same properties as case classes but they will also have a static lifetime. Because of these properties, case classes can be used as building blocks for algebraic data types. We demonstrate defining sum types on Listing 48 and product types on Listing 49.

```scala
case class Pair[A, B](val first: A, val second: B)
```

Listing 49: A product type using case classes.

### A.1.4 Functions as First-class Citizens

Functional programming requires that functions can be treated as simple values. Function types (their full signature with the return type) are part of the type hierarchy and they can be referenced via vals and vars, they can be passed as parameters to other (higher-order) functions and anonymous functions can be created as well. Listing 50 shows some examples to these usage patterns.

```scala
class GenericFunctions(
  protected val computeInts: (Int, Int) => Int =
    (a, b) => a + b
) {
  protected def doubleInt(a: Int) = a * 2
  def map[A](s: Seq[Int]): Seq[A] = a. map (doubleInt)
}

val subtraction = new GenericFunctions((a,b) => b - a) {
  override def map[A](s: Seq[Int]): Seq[A] =
    a.map { _.toString }
}
```

Listing 50: Usage patterns for functions.

Additionally, Scala supports currying, multiple argument lists for functions and nested function definitions.

### A.1.5 Generics, Variances and Structural Types

Scala supports parametric polymorphism in the form of generic classes and functions. The type parameters can have lower and upper bounds defined for them using the >: and <: operators, respectively. On top of this, there is support for three kinds of type variance: invariance, covariance and contravariance. When a type parameter is marked with a + sign, it is covariant, - means contravariance and no marks refers
to invariance. Types do not necessarily need to be referred to by their symbols, programmers can also use structural types where they only define the required shape of the type and not its name. An example can be seen on Listing 51.

```scala
class NeedsStructuralType[A <: 
  { def compare(a: A, b: Object): Int }
]
```

Listing 51: Structural type in a generic class definition.

### A.1.6 Type Inference

Having a strong and static type system enables the compiler to infer almost all types of locally defined constructs. This includes local variables and values and in most cases, return types of methods. When no type is explicitly specified, the compiler will try to run a modified version of the Hindley-Milner type inference algorithm to figure out the omitted types. If it succeeds, it will annotate the corresponding AST with the inferred type and continue the compilation process. If the algorithm cannot return a single type or diverges, it will throw a compilation error and the programmer needs to annotate the code manually. One of the most notable cases is recursive functions that always require explicit return types.

In most cases, the lastly evaluated expression will define the type of a function. When the type inference algorithm meets conditionals with multiple types, the least upper bound of those types will be calculated.

### A.1.7 Type Fields and Self Types

Apart from values and functions, classes and traits can contain types as fields and they can be overridden in derived classes. In case of abstract classes and traits, these can also be abstract. Type fields can be used just like any other type symbol: they can be types of values, parameters or return types of functions or can be passed as parameters to generic types. They can also have structural types as their definitions. For an example, see Listing 52. Using type fields, it is fairly easy to solve problems like family polymorphism [37], among others.

There is a special type field in traits, called the *self type* which can define mandatory requirements for the type hierarchy. If a self type is defined, it can dictate what other traits need to be mixed in with the given trait without explicitly extending them. This enables traits to depend on other traits without explicitly injecting them into their own type hierarchy.
trait TypeFieldBase {
  type ReturnType <: { def parseFrom(s: String): Int }
  def returnsAValue: ReturnType
}

class TypeField {
  override type ReturnType = Int
  override def returnsAValue = 3
}

Listing 52: Structural type in a type field definition.

A.1.8 Implicits

Scala supports implicits in two ways: as implicit parameter lists and implicit converter functions. In the first case, a function definition can mark at least one of its argument lists with the keyword \texttt{implicit} and in this case, providing these values \textit{explicitly} is not required as long as the compiler can find eligible values at the place of the function application. Eligible values are ones that

- can be accessed without any scope modifiers and are marked with the \texttt{implicit} keyword or

- are part of a companion object (an object with the same name as the given class or trait) and are labeled implicit.

Implicit conversions come into picture when there is a specific type error in the code. If either the expression’s return type does not conform with the place of the expression or there is a member selection (by the dot-operator) for a member that is not available on the given type, instead of simply throwing a type error, the compiler will look for eligible implicit converters that would resolve this error. If it finds exactly one in scope, it will apply it implicitly, or if there are more or less than exactly one, it will throw a compilation error. The method needs a specific signature to be selected for a conversion: in case of an \texttt{A => B} conversion it needs to be \texttt{implicit def a2b(value: A): B}. To keep the problem of implicit selection practical and solvable, the compiler only tries to apply one implicit conversion, and not a chain of them (i.e. if there is a conversion required for \texttt{A => B} but that can only achieved by \texttt{A => C => B}, this will not be applied).

A.1.9 Pattern Matching

The language has powerful pattern matching capabilities. The syntax can be familiar to programmers using functional languages; an example can be seen on Listing 53. The supported main features are as follows:
- Values: in a pattern, programmers can define simple values to match against.

- Types: patterns can include simple types.

- Guards: arbitrary boolean expression can be evaluated in a case expression. Previously extracted values are available.

- Unapply methods: if a type defines unapply methods, they can be used in patterns. This functionality can be extended by extractor objects for types the programmer cannot modify.

- Omitting values: using _ one can omit unused parts of a pattern and they will match any value.

- Matching any value of any type.

- Binder: values can be bound to variables using the @ sign. These can be referred to in the case block.

- Default case: a simple _ will match anything without creating a reference to the value.

```scala
val p: Any = Pair(42, "Scala")

val matched = p match {
  case 42 => "Found 42"
  case Pair => "Found a pair"
  case p: Pair if p.first == 42 => "Found a pair with 42 and Scala"
  case Pair(42, "Scala") => "Found a pair with Scala"
  case x => s"Found a ${classOf[x].getName}"
  case Pair(f @ _, _) => s"Found a pair of $f as its first"
  case _ => "Found something"
}
```

Listing 53: A product type using case classes.

The compiler checks for pattern completeness when all possible unapply methods are known at compile time or when a match only lists subtypes of a sealed trait. It also warns against impossible cases, for example when adding a case for a type that is not a subtype of the expression that is being matched against.
A.1.10 Extending the Compiler

The current version of the Scala compiler, *scalac* is open source and has a very open architecture. It can be easily extended by compiler plugins or macros. These extensions are written in Scala, they run at compile time and they have the ability to observe and transform the abstract syntax tree (AST) [22]. Compiler plugins are more flexible, as they can be inserted after any compilation phase and the only constraint on them is that they need to produce an AST that is accepted by later phases. Macros are run as the part of the typer phase and use the compiler’s low-level reflection API as well as some macro-specific infrastructure, for example *reify*, *splice* and *quasiquotes*. These enable the programmer to easily convert between literal code blocks and ASTs. Since the revamped version of macros that appeared in Scala 2.12, almost all practical use cases can be solved by macros, including new type generation via materializers.
A.2 Dissertation Summaries

A.2.1 Dissertation Summary

The dissertation contains novel research results on four aspects of software development: type systems, managing complexity related to error handling, levels of component coupling and concurrent programming at high abstraction levels.

The first thesis represents my research on approaches where we can compute the return types of simple recursive functions without a fixed-point search and how the compilation performance is affected by implicit resolution and how this can be improved.

Analyzing and managing the complexity of software systems is a core question of every software project. Error handling has been almost completely neglected in this research area, despite the fact that it is an essential part of programs. The advancements in the error handling approaches promise reduced complexity while providing an increase in usability and feature set. I analyze how different error handling methods compare to each other with a scientific rigour. I also investigate these error handling methods in paradigm-agnostic and multiparadigm settings to gain an understanding of their usefulness in different programming methodologies (Thesis II).

Software complexity can be of course observed at higher abstraction levels as well. A widely accepted metric of complexity is the coupling between components. It is generally accepted that lower coupling helps reduce complexity and overall software development costs. In the dissertation I investigate what properties coupling metrics present in multiparadigm languages and if there are special attributes we can observe. Furthermore I try to find an answer if there is a natural limit to reducing coupling (Thesis III).

As we are reaching the physical limits of single-threaded performance of CPUs, the focus shifts towards concurrent and parallel computing. Parallel computing increases the complexity of programs in a completely new aspect: most programmers find it extremely hard to argue about correctness of their code if it can run on multiple threads. This is the motivation behind finding high-level abstractions for parallel computing problems. A fairly new abstraction is called Read-Copy-Update, but its implementations so far have been only available in lower-level languages. I examine if this can be changed and if RCU can be implemented in a high-level language efficiently. I test the performance of different approaches by implementing a common data structure, a concurrent hash map and evaluate it in a wide range of use cases (Thesis IV).
A.2.2 Disszertáció Összefoglaló

A disszertáció új kutatási eredményeit tartalmaz a szoftverfejlesztés következő területein: típusrenderszerek, szoftverbonyolultság hibakezelés esetén, komponensek kapcsolódási szintje valamint konkurens programozás magasz absztrakciószinteken.

Az első tézis azon kutatás eredményeit mutatja be, amely rekurzív függvények típuskövetkeztetését fixpontkeresés nélkül teszi lehetővé, valamint az implicit hibakezelés esetén és konkurens programozás magass absztrakciószinteken.

A szoftverrendszerek komplexitásának elemzése és kezelése alapvető kérdése minden szoftverprojektnek. A hibakezelés kérdése majdnem teljesen kimaradt eddig ebből a kutatási területből, annak ellenére, hogy egy meghatározó része a programoknak. A hibakezelésben elért fejlesztések a csökkenő komplexitás mellett a korábbi módszerek használhatóságának és a támogatott tulajdonságainak körének bővítését ígéri. Tudományos alapossággal vizsgáltuk meg, hogy a különböző hibakezelési módszerek hogyan viszonyulnak egymáshoz. A kutatásunk paradigmaagnosztikus és multiparadigma környezetben végeztük, hogy fényt derítsünk a használhatóságukra különböző programozási metodológiák használata közben (Tézis II).

A szoftverkomplexitás természetesen magasabb absztrakciós szinteken is megfigyelhető. Egy széleskörűen elfogadott mértéke a komplexitásnak a komponensek közötti kapcsolódás. Az általános vélekedés szerint a kapcsolódások számának alacsony szintje segíti a komplexitás és az egyéb feljesztési költségek alacsonyan tartását. A doktori értekezésben megvizsgáljuk, hogy a kapcsolódási metrikák milyen tulajdonságai találhatóak meg multiparadigma nyelveken és hogy találhatunk-e különleges jellemzőket. Továbbá arra is keressük a választ, hogy van-e természetes alsó határa a kapcsolódási mértéknek (Tézis III).

Ahogyan közelítünk az egyszálú feldolgozás fizikai határához, a fókusz egyre inkább a többszálú programozás felé irányul. A párhuzamos programozás egy teljesen új aspektusból növeli a komplexitást: a legtöbb programozó a hatalmas kihívást jelent egy program helyességét bizonyítani, ha az több szálban hajtódik végre. Ez a fő motiváció magasszintű párhuzamos programozási absztrakciók megalkotására. Egy meglehűséges új absztrakció a Read-Copy-Update, de az eddigi implementációi mind alacsonyszintű nyelveken voltak elérhetőek. Megvizsgáljuk, hogy ezt meg lehet-e változtatni, és hogy az RCU hatékonyan implementálható-e egy magasszintű nyelvben. A teljesítménytesztelést egy gyakran használt adatszerkezet, a konkurens hash map implementációján végezzük, amit használati esetek széles skáláján vizsgálunk meg (Tézis IV).
ADATLAP
a doktori értekezés nyilvánosságára hozatalához

I. A doktori értekezés adatai
A szerző neve: Nagy Gergely Attila
MTMT-azonosító: 10066352
A doktori értekezés címe és alcíme: Elements of Multiparadigm Programming in Scala
DOI-azonosító: 10.15476/ELTE.2020.143
A doktori iskola neve: ELTE Informatika Doktori Iskola
A doktori iskolán belüli doktori program neve: Informatika Doktori Iskola
A témavezető neve és tudományos fokozata: Dr. Porkoláb Zoltán, egyetemi docens
A témavezető munkahelye: Programozási Nyelvek és Fordítóprogramok Tanszék

II. Nyilatkozatok
1. A doktori értekezés szerzőjeként
   a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.
   b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közvetíteléig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;4
   c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (datum)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;5
   d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerzőjéhez tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerződést követő egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.6
2. A doktori értekezés szerzőjeként kijelentem, hogy
   a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szerzői munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;
   b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.
3. A doktori értekezés szerzőjének aláírása


2 A kari hivatal ügyintézője tölti ki.
3 A megfelelő szöveg aláhúzandó.
4 A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácsot a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.
5 A doktori értekezésben benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.
6 A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.