AN ABSTRACT OF THE DISSERTATION OF

Ameya Ketkar for the degree of Doctor of Philosophy in Computer Science presented on August 18, 2021.

Title: Towards Automating Type Changes

Abstract approved: _____

Danny Dig

Developers frequently change the type of a program element and update all its references for performance, security, concurrency, library migration, or better maintainability. Despite type changes being a common program transformation, it is the least automated and the least studied. Manually performing type changes is tedious since the programmers have to reason about propagating the type constraints of the new type over assignments, method hierarchies and subtypes. Existing automation approaches for type changes are inadequate for large-scale code bases. Moreover these techniques require the user to encode the adaptations for the type change in a DSL, which might not be straight-forward if she is unfamiliar with the types. These challenges introduce barriers in the adoption of these techniques. The thesis of this dissertation is: *it is possible to design and implement type change techniques and tools that are scalable and usable.* For this purpose, we developed: (i) TypeChangeMiner: a tool that accurately and efficiently detects type changes performed in the version history of a project (ii) T2R: a ultra-scalable MapReduce amenable framework to perform type changes safely and accurately, and (iii) TC-Infer: a technique that learns the task of performing type changes by analyzing how other developers have previously performed the same type change.

Towards Automating Type Changes

by

Ameya Ketkar

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 18, 2021
Commencement June 2022

Doctor of Philosophy dissertation of Ameya Ketkar presented on August 18, 2021.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Head of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Ameya Ketkar, Author

ACKNOWLEDGEMENTS

The research presented in this dissertation would not be what it is without the guidance and support from many people.

I would like to thank my advisor and mentor Danny Dig, without whom this work would not have been possible. He believed in me five years ago when I had absolutely no research experience of any sort, motivated me to pursue research in Software Engineering as a masters student and then as a PhD student. I am very grateful to him for taking me to multiple conferences and introducing me to other researchers in software engineering community. I would thank him for being a patient listener when I would rant about the extremely lower level details of the work, while gently guiding me to look at the bigger picture. I am the researcher I am because of him.

Nikolaos Tsantalis (a.k.a. Nikos) has been my mentor, friend and collaborator throughout my PhD. His work ethics, his research ideas and his strive for perfection has been a constant source of inspiration for me. I thank him for inviting me to Montreal for the fall term, accepting my contributions to RefactoringMiner and being my critical-thinking partner throughout the writing of this dissertation. The most value-able lesson he imparted to me was - "*If something can be done in two minutes, do it now!*"

I must also thank Davood Mazinanian for being the best student mentor a graduate student can wish for. I have always been in awe of your patience, attention to detail, and artistic skills. I am very grateful to him for introducing me to all tools and techniques needed to excel in research and catalyzing our collaboration with Google. I am also grateful to Ali Mesbah for believing in me then (a first year graduate student) and spending many Fridays to adapt my project to the Google Infrastructure. Thank you for being patient with me, guiding me to write very high quality code, and working tirelessly to during the writing of the paper. This dissertations would not be what it is, without your valueable contributions. I am also grateful to Eddie Aftandilian for enabling this collaboration. I would like to thank Oleg Smirnov and Timofey Bryskin for collaborating with us and producing a wonderful plugin that can takes our research to the hands of real world developers. I would specially thank Oleg for being excited about type changes and tirelessly developing the plugin with enabling such a smooth user experience. I would also like to thank Gustavo Soares, Arjun Radhakirhsnan and Ashish

Tiwari for being wonderful mentors during my internship at Microsoft, and enriching my PhD journey. The work I did at Microsoft has been an inspiration for the last chapter of this dissertation. I would also like to thank Raffi Khatchadourian for helping me initially traverse the domain of type-related changes and then for providing extensive feedback for my work. I would also like to thank Martin Erwig and Eric Walkingshaw for the amazing courses they offered, constructive feedback they provided on my research, and specially for creating an inclusive environment in the *lambda reading group*. Further I would like to extend my gratitude to Malinda Dilhara for being a wonderful colleague and thoughtful friend. My experiences at various conferences would have been less fun without him.

Surviving graduate school would have been impossible had I not adventured each weekend into the unholy dungeons fighting demons, devils and warlords with the *Wild Stallions* - Jeffery Young, William Maxwell, Colin Shea-Blymer and Keeley Abott. I would thank Jeff for introducing me to *DnD*, co-programming nursery rhymes in Euterpea, and being a wonderful friend. I would also take this opportunity to thank Ganesh, Aashutosh, Abhishek, Mihir and Prashant for their constant support during my time at OSU.

I would like to thank my partner Meghamala Sinha for her constant support and being my best friend through the thicks and thins. I would also extend my gratitude to my brother Anjaneya and my cousin Aditya for always being supportive of my actions and adding a different perspective to my life. Last but not the least I am eternally grateful to my parents Sanjay Ketkar and Ashwini Ketkar for instilling in me values and principles that taught me to be a good human being more than anything else.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

Page

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

In loving memory of Bhaskar and Padmavati Ketkar, and Bhau Joshi.

Chapter 1: Introduction

As programs evolve, the type of a variable or a method may be updated from type T to type R, because T is deprecated, R is more efficient or R has some desirable property. For example, updating `HashMap` to `ArrayMap` improves runtime performance [124]. To perform a type change, developers update the declared type of a program element (local variable, parameter, field, method return type) and then adapt the code referring to this element (within its lexical scope) to the API of the new type. Due to assignments, argument passing, or public field accesses, a developer might adapt several expressions (like new instance creation, static method invocations, lambda expressions) and perform a series of type changes to propagate the type constraints for the new type. This task can easily overwhelm the developers. For example, when CoreNLP developers replaced generic with specialized Java 8's Functional Interfaces, the type changes involved 34 files containing 75 declarations, 74 allocation sites, 35 call sites, 3 subclasses and 39 lambda expressions. With the size of the projects and extensive use of a type, the complexity of type change increases. Moreover, a type change is a foundational step in class/library migration [133], fixing API-breaking changes in clients (e.g., updating a method's signature [30]), or correcting inefficient uses of an API [89]. Type changes may be also applied for other reasons such as improving the maintainability of the code [27] (`String`→`Path`), improving performance[124], introducing concurrency [36] (`Map`→`ConcurrentMap`) or immutability (`List`→`ImmutableList`), changing properties `LinkedList`→`Deque` or even improving security(`Random`→`SecureRandom`). Despite type changes being a common program transformation, it is the least studied and has negligible automated tool support.

Most of the prior work studied type changes in the context of other evolution tasks such as API updates [24, 31, 32] and library migration [4, 69, 134]. However, there is a gap in understanding type changes in the general context. With this knowledge gap, researchers miss opportunities to improve the state of the art in automation for software evolution, tool builders do not invest resources where automation is most needed, language and library designers cannot make informed decisions when introducing new

types, and developers fail to use common practices when changing types.

Type changes are considered to be hard to automate[23]. Developing automation techniques for *type changes* was an active area of research more than a decade ago, when researchers Tip et al. [13, 138] made a breakthrough for the program analysis community with their work *Refactorings using type constraints*. Type changes have gained little traction from researchers [36, 74, 77, 149] since then. However, type constraints analysis is resource intensive [13, 138] since it has to extract constraints for *all* program types, thus will not scale to today's large open source projects containing hundreds of thousands of lines of code or Google's code base of 300M LOC.

Modern IDEs provide very little support for type changes. It is not only limited to few pre-defined types but it is also not extensible because they do not allow the user to configure the tool with custom mappings between the new and old types' idioms. Unlike IDEs, the state-of-the-art technique [13] allows the user to express the adaptations required for the type change as a program in a DSL. Though tedious, developers can easily encode the transformation in the DSL if they are familiar with the types involved in the type change. However, if they are unfamiliar with the types they would have to ask a co-developer or look up the documentation (could be outdated or unavailable), release notes or Q&A forums to understand how correctly to adapt the code to perform the type change and encode it in the DSL. Moreover, handcrafting such rewrite rules is awkward [17] and often involves multiple trial and error iterations. This introduces a barrier to the adoption of these techniques.

1.1 Goal Statement and Research Questions

Previously researchers Negara et al. [101] surveyed 420 developers and ranked type changes as the most highly desired feature (among commonly applied transformations) for IDE automation. Moreover, researchers Nishizono et al. [110] found that among other source code maintenance tasks, changing the type of a variable requires the longest comprehension time. Our goal is to *reduce the burden of software developers* by providing automated assistance to perform type changes. The thesis of this dissertation is:
It is possible to design and implement type change techniques and tools that are scalable and usable.

To be practical, our solution must address the problems that developers encounter

in practice when performing type changes. Our solution to better understand the space of type changes, is to perform a large scale empirical study that mines type changes performed in the version history of large open source projeccts. This study compares the practice of type changes with that of other refactorings, explores the various characteristics of these type changes, investigates the edit patterns applied to adapt the references, and identifies the most popular type changes applied in practice.

Our solution for *scaling* type change techniques is to make the core analysis that reasons about how to propagate type changes safely, *distributed*. However the existing techniques cannot adapt to distributed processing frameworks as they access single source code files one at a time in an arbitrary order, during which no global state is maintained. To overcome this, we designed a three-step algorithm that takes as input the source code and the transformation specifications. The specifications define the source and target types ($\texttt{T} \rightarrow \texttt{R}$) and the rewrite rules for adapting the references (and other expressions). Dividing the type migration technique into three separate phases enables scalability in large codebases where the entire code spans hundreds of millions of LOC. Each of the steps becomes amenable to distributed execution internally through, for example, MapReduce [28] The output of each step is fed as input to the next step.

Our solution to make type change techniques more usable, is to automatically infer the transformation specifications from previously applied type changes and then provide these specification as input to the type change techniques. For this purpose, we extend RefactoringMiner to identify type changes performed in the commit history of a project and then develop an AST difference algorithm to deduce the fine-grained rewrite rules required to adapt the source code to the type change.

In this dissertation, we investigate the following key questions regarding automating type changes:

1. How common are type changes in practice? What are the most popular type changes? What common edit patterns are applied to adapt to these type changes?

2. How can we develop a technique to apply type changes in ultra-large-scale code bases? Is the accuracy of such a technique good enough to be used in practice?

3. Can the transformation specifications for popular type changes be inferred from the version histories of open source projects? Are these specifications accurate and effective?

4. Do real world developers find automated type change techniques useful? Can our automation make them more productive?

To answer these questions, we developed

1. TypeChangeMiner: for accurately and efficiently detect type changes performed in the version history of a project
2. T2R: a MapReduce amenable framework to express and perform type changes
3. TC-Infer that deduces the specifications from previously performed type changes
4. IntelliT2R: An adaptation of *IntelliJ IDEA*'s type migration technique that can be configured through transformation specifications inferred by TC-Infer.

1.2 Towards Automating Type Changes

1.2.1 Understanding Type Changes in Java

To understand the challenges the developers face when performing type changes, we present the first longitudinal, large-scale, and most fine-grained empirical study on type changes performed in practice. We analyze the commit histories of 129 large, mature, and diverse Java projects hosted on GitHub. For this purpose, we extended the state-of-the-art refactoring mining technique RefactoringMiner to efficiently and accurately mine (in a refactoring aware manner) 416,652 commits and detect 297,543 instances of type changes. We thoroughly evaluated our tools and they have 99.7% precision and 94.8% recall in detecting type changes. We also developed TypeFactMiner, for efficiently and accurately resolving the type binding information of a source code elements, for the entire commit history of a project. To advance the science and tools for type change automation, we use this rich and reliable dataset to answer six research questions, and present empirically-justified implications for researchers, tool builders, software developers, library designers and educators. Some of our key surprising findings are:

1. Type changes are *more common* than renaming. Developers often complement a type change with a rename to maintain consistency.

2. A large variety of rewrite rules are applied to adapt the source code to the type change. Too large to be simply enumerated manually. Moreover to adapt the code, developers often perform secondary *cascade* type changes, which are different than the primary type change.

3. Developers more often perform type changes on `public` program elements rather than `private`, `package-private`, and `protected` elements, introducing potential breaking changes. Although the raw number of type changes on `private` elements is less than the number of type changes on `public` elements, developers tend to change more often the types of `private` elements compared to `public` ones, indicating possible considerations for preserving backward compatibility.

The key empirically-justified implications of this study were:

1. *Inferring transformation specifications from version history*: We found that type changes are highly repetitive, within individual commits (7.3 TCIs per commit) but also across multiple commits from distinct projects. This confirms the findings of others on the repetitiveness of code changes [101, 104, 108, 121, 122], and calls for new research on inferring type change mappings from previously applied type changes.

2. *Automate Reference Adaptation* Type change is a very commonly applied transformation. This highlights that IDEs should provide support for advanced composite refactorings, which perform a type change and adapt the code referring to the variable whose type changed. While current IDEs support refactorings like *Change Method Signature* [50] or type migration [67], they only update the declaration of the method or variables, but do not adapt the references. The *adaptation process* requires identifying the syntactic transformation (edit patterns) required to adapt the references of the variable (or method) and type dependent expressions (or statements) for a particular type change.

1.2.2 Type Migration in Ultra-large-scale Codebases

A safe type migration technique requires *whole-program analysis*, therefore it is challenging to make it scalable. One possible solution is to make the analysis distributed, but this implies accessing source code files one at a time, possibly in an arbitrary order, thus

making it unsuitable for whole-program analysis. To overcome this we propose T2R, a three-step process (shown in Figure 2) that, given the source code and transformation specifications for a type change, 1. it collects relevant type and syntactic information from the source code, 2. then analyzes this information to narrow down modification sites, and 3. finally applies code transformations. Each of these steps are amenable to MapReduce processing, thus making the approach ultra-scalable. T2R bears several intricacies - 1. collecting complete source code information for accurate analysis, 2. constructing an expressive graph based model to facilitate type migration analysis, 3. applying preconditions to identify *safe* migrations, and 4. effectively translating analysis results into source code transformations.

We evaluated T2R for its *accuracy*, *usefulness* and *scalability*. To evaluate T2R, we instantiated it with specifications to migrate seven generic functional interfaces in Java to 35 specialized alternatives. We run T2R on Google's codebase with 300M lines of Java code and on seven performance-critical open-source projects like CoreNLP, Cassandra, SonarQube and Neo4j, totaling 2.6M LOC. T2R could efficiently analyze the ultra-large scale code base of 300MLOC and performed 59 large migrations in just 33 minutes, where it took 30 minutes to parse the code on their cloud infrastructure and 3 minutes for the actual analysis. T2R generated 130 patches in total, of which 126 compile and pass tests successfully. The original developers of these leading open source projects and at Google accepted 98% (114/126) of these patches.

1.2.3 Inferring the Transformation Specifications

The Achilles heel of techniques automating type changes is that the user has to manually encode the syntactic transformations required to perform the desired type changes. While these techniques allow the transformations to be expressed as transformation specifications, they still require significant human effort; introducing the barrier for the adoption of these techniques. To overcome this challenge, we introduce TC-Infer, a technique that infers the specifications required for performing a type change from the previously performed instances in the commit history of a project(s). Particularly, our technique 1. crawls over the commit history to identify instances of type changes and other refactorings, 2. analyzes the adapted statements that reference the element whose type is changed, 3. for each adapted statement, establishes the mapping between

the sub-trees (sub-expressions/statements) and deduces the rewrite rules capturing the adaptation in the COMBY language, and 4. groups the rewrite rules inferred for a type change kind as a transformation specification The rules produced by our technique can be used by existing state-of-the-practice tools like IntelliJ's Type Migration, or state-of-the-art tools that use type constraints [13] or by T2R that uses type-fact graphs [74].

We evaluate the *applicability*, *effectiveness* and *usefulness* of the transformation specifications reported by TC-INFER. We found that our technique can infer rules for 93% of the *popular* type changes. We evaluate accuracy on dataset of 245 commits containing 3060 instances of 60 diverse type change kinds. We manually validated our the changes, and our results show that rules produced by TC-INFER have average precision of 99.2% and average recall of 93.4%. We also demonstrate the usefulness of TC-INFER, by developing a plugin for the INTELLIJ IDE that provides assistance to developers to perform type changes. This plugin extends the current IntelliJ's Type Migration framework to perform custom type changes with the rules produced by TC-INFER as input. We evaluate the usefulness of the rules inferred by TC-INFER, by assisting the developer to perform type changes in the IDE by suggesting these rules.

1.3 Contributions

This dissertation makes several important contributions.

1. **Problem Description.** This dissertation describes an important problem, *automating type changes*, a very frequently performed, tedious and repetitive transformations that has been ignored by researchers and tool builders

2. **Insight into the type changes performed in practice.** To the best of our knowledge, ours is the first longitudinal, large-scale and most fine-grained empirical study on type changes in practice. While other researchers focused on type changes appearing in the context of library migrations only.

3. **A rigorous approach and framework.** We propose a framework that uses a graph modelling the type structure of the program and employs a three step MAPREDUCE amenable approach to perform type migrations *safely* in *ultra-large code bases*.

4. **A practical tool set.** We develop a tool-set to automate the the task of performing

type changes that is safe and practical (no overhead on for the users to construct the required specifications) - (a) T2R is a IDE-independent tool that can perform type migrations in ultra-large scale codebases, (b) The latest RefactoringMiner now has the capabilities to accurately report type changes performed in the commit history, and (c) TC-Infer a tool to infer the transformation specifications for the type changes from the version history of a project. These specifications can be provided as input to the type migration techniques. (d) An extension of IntelliJ's Type Migration that can use custom specification to perform type changes.

5. **Empirical evaluation.** We evaluate all the tools in the tool-set through thorough manual validation and show they have high accuracy (precision and recall).

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows : Chapter 2 briefly describes our formative study [89] that explores practice of using lambda expressions in Java, where we found the misuse of functional interfaces that could be eliminated by appropriately configuring and applying type change technique. The findings from this formative study have motivated many contributions in this dissertation.

Chapter 3 presents our large scale empirical study on type changes in Java. The first part presents the tools that we developed to identify and thoroughly analyze type changes performed in the version history of a project . The second part answers six research questions about the practice of type changes based on the fine-grained changes collected from a large corpus of 130 large open source projects, making our findings representative. The third section presents an *actionable, empirically justified set of implications* of our findings from the perspective of four audiences: Researchers, Tool Builders, Language Designers, and Developers. This chapter was published at FSE 2020 [76].

Chapter 4 presents T2R, a technique that can perform type changes (migrations) in ultra-large scale codebases. The first section motivates the problem and highlights the challenges for automating type migration . The second section thoroughly discusses the MapReduce amenable three-step process. The third section evaluates T2R for its *accuracy*, *scalability* and *usefulness* at eliminating misuses of functional interfaces. This chapter was published at ICSE 2019 [74].

Chapter 5 presents TC-Infer, that infers the transformation specifications from the version history of the project. The first section highlights the intricacies related to capturing the adaptations as rewrite rules and particularly describes algorithm InferRules that produces these rewrite rules for two input code snippets. The second section evaluates the effectiveness and usefulness of the specifications produced by TC-Infer through an extension of IntelliJ's Type Migration framework. This chapter is under-review at ICSE 2022.

Chapter 6 surveys related work and Chapter 7 concludes and talks about future work.

Chapter 2: Formative Study

Prior to studying type changes, we performed a large-scale, quantitative and qualitative empirical study that shed light on how imperative programmers use lambda expressions as a gateway into functional thinking [89]. Interestingly, we found out that developers are using Java's built-in functional interfaces inefficiently, i.e., they prefer to use generic functional interfaces over the specialized ones, overlooking the performance overheads that might be imposed. For instance, they use `Function<Double,Double>` instead of `DoubleUnaryOperator`. This study [89] extensively investigates the practice of using lambda expressions by answering six research questions. In this chapter we will briefly describe the methodology(Section 2.1), and results(Section 2.2) and implications(Section 2.3) relating to the one research question that is relevant to the theme of the dissertation - *What features of Lambda Expressions are Adopted by Java Developers?*. The results in this chapter serve as motivation for this dissertation.

In this work, we perform the first large-scale study on 241 open-source projects hosted on GitHub, containing 100,540 lambda expressions. In addition, we study the evolution history of these projects to discover trends in the adoption rate of lambdas. Moreover, to better understand the reasons motivating developers to use lambdas, we survey 97 developers of these projects right after introducing a lambda expression. Using the collected information, we answer the following research questions:

RQ1: *What* features of lambda expressions are adopted by Java developers? We found that developers mostly create their own custom functional interfaces rather than using the ones provided by Java in the `java.util.function` package, mainly due to the need for exception handling capabilities and extending other interfaces. In addition, we found that 20% of developers use unnecessary boxing and unboxing without thinking about the performance overhead. Moreover, test code has significantly higher lambda density than production code.

RQ2: *How* do Java developers introduce lambda expressions? We investigate the introduction rate of lambdas over time, starting from the first commit introducing a

lambda until the last commit of the project. We found an increasing trend in the adoption rate of lambdas.

RQ3: *Who* introduces lambda expressions? We found that a small to moderate percentage of developers introduce most of the lambdas. Moreover, core contributors introduce slightly more lambdas than outsiders.

RQ4: *How* do Java developers introduce lambda expressions? We found evidence of migration efforts in several projects by converting anonymous classes to lambdas, replacing loops/conditionals with streams, and enhancing functionality by wrapping existing code to lambdas. Moreover, we found that most lambdas in new code are introduced without tool support.

RQ5: *Why* do Java developers introduce lambda expressions? We found 14 common reasons motivating developers to introduce lambdas, including improved readability, class creation avoidance, behavior parameterization, simulating lazy evaluation, callback implementation, and improved testability. Developers also employ lambdas to implement object-oriented design patterns and replace class inheritance with object composition.

Using this rich data, we reveal several actionable implications. Researchers can create novel applications of lambda expressions (e.g., in the GoF design patterns [56]) and focus as well on code migration scenarios, thus increasing the adoption of lambdas. Language and library designers can find several pain points that developers wrestle with, which call for better language support. Among others, we highlight support for checked exception and optimization of generic boxed versions into specific primitives. Tool builders can improve tools to provide relevant recommendations and assistance intelligently, for example by suggesting currying refactorings. Developers can educate themselves on how to use lambdas wisely and prudently, for example by becoming aware of the implications on performance or code readability.

2.1 Research Methodology

In this study, we employ both quantitative and qualitative methods for answering our research questions. The qualitative approach answers *why* Java developers use lambda

expressions, while the quantitative approach allows answers research questions. In this chapter we will describe only the quantitative approach that was used to answer the first research question which is relevant to the dissertation.

2.1.1 Subject Systems

We collected the top 2000 Java projects on GitHub, ranked based on the number of stars (i.e., the *stargazers count*). Our dataset covers a wide range of Java projects, allowing to have diversity with respect to the application domain, size, development ecosystem, contribution governance, testing practices, and so on. This diversity is important to make sure that the collected data is representative for the study. First, we conducted an initial analysis on the latest revision of these projects to understand which ones have already started adopting lambda expressions. Particularly, we parsed the source code of the projects to generate Abstract Syntax Trees (ASTs), and then queried the ASTs to find uses of lambda expressions. All projects without any use of lambda expressions were filtered out, as they are not suitable subjects for our study. After the filtering phase, 241 projects remained for further analysis. Due to the space constraints, we cannot provide the complete list of these projects; however, the list is available online [90].

2.1.2 Static Source Code Analysis

We performed a thorough static analysis of the latest revisions of the source code, in order to investigate:

- the characteristics of the lambda expressions introduced by the developers, and discover good or bad practices in the way Java developers use lambdas
- the locations that developers tend to use more lambda expressions, e.g., production or test code

We implemented our static analysis tools on top of two AST Parser libraries, namely Eclipse JDT [29] and JavaParser [64]. Our static analysis extracts the syntactic information of the lambdas from the source code, namely the number of parameters, the information about the parent node of the lambda in the AST, and the location of lambda in source code. Eclipse JDT gives us further in-depth information from the

compiler, namely the fully-qualified name of the type binding resolved for the functional interface in context, the exception handling capability of the functional interface, whether the functional interface is built-in (provided in Java 8 APIs), and the signature of the abstract method defined in the functional interface, including the type of its parameters and the return type.

To obtain precise binding information using Eclipse JDT, we had to build the studied projects. The majority of the projects in our corpus use a build system, such as Maven and Gradle, which makes the project building process rather straightforward by automatically downloading all required libraries. However, in several cases we found missing libraries that required considerable manual effort to be resolved. We successfully built 147 projects, making it possible for us to use Eclipse JDT to collect in-depth information from 59,984 lambdas. We used JavaParser to analyze the 94 remaining projects, giving us just the syntactic information for over 40,556 lambda expressions. Overall, our study covers 100,540 lambdas found in the 241 studied projects, which gives us a large dataset to answer our research questions comprehensively.

2.2 Results

2.2.1 *What* features of Lambda Expressions are adopted by developers?

Use of Custom Versus Build-in Functional Interfaces:
Java 8 provides 43 built-in functional interfaces (in the `java.util.function` package) that can be used as types for lambda expressions. These functional interfaces cover a large number of input/output parameter combinations, so that the developers do not need to introduce their own custom functional interfaces. For example, the functional interface `Supplier<T>` represents lambdas that accept no parameter and return an object of type `T`. Conversely, `Consumer<T>` is used for lambdas that accept an object of type `T`, and return nothing. The question is, to what extent these built-in functional interfaces satisfy the developers' needs, and are they underused or misused?

We studied 59,984 lambda expressions used in 147 projects, by resolving the type that they are bound to at compile time. We found that 32% of lambda expressions use built-in functional interfaces, and 11% implement Single Abstract Method interfaces

provided by the Java API, such as `Comparator` and `Runnable`. While the majority of the built-in functional interfaces (36 out of 43 in total) are specialized for primitive types (e.g., `IntToDoubleFunction` is a functional interface which accepts an `int` and returns a `double`), we found that only 7% of the lambdas are using this kind of specialized functional interfaces. However, our analysis revealed that 20% of the studied lambda expressions could have been replaced with a functional interface specialized for a primitive type. For example, `Function<Integer, T>`, where `T` can be any type, was used more than `IntToDoubleFunction`, `IntToUnaryFunction`, `IntToLongFunction` and `IntFunction` combined. We found similar trends for other primitive data types and functional interfaces.

In essence, Java 8 provides these specialized interfaces to improve performance. These functional interfaces use primitive types in the signature of the abstract method, consequently, when using these interfaces, there is no need for auto-boxing by converting primitive types to their corresponding object wrapper classes.

The data shows that developers are using functional interfaces *inefficiently*. We hypothesize different reasons for this inefficient use of built-in functional interfaces, e.g., lack of awareness of the available options, or the tediousness of finding the correct type of functional interface among the 43 options. To the best of our knowledge, no IDE can detect inefficient uses of functional interfaces, or allow refactoring an instance of a general functional interface to a specialized one.

We also found that 57% of the lambdas are instantiated from *custom functional interfaces*, i.e., the ones that are defined by the developers. Why do developers need to define custom functional interfaces, given that Java provides so many built-in options?

One possible reason is that developers are *forced to*. As an example, there is no functional interface of which the single abstract method accepts more than two arguments, in the list of Java's built-in functional interfaces. The number of built-in functional interfaces would obviously grow exponentially, if Java language designers wanted to include functional interfaces capable of accommodating more parameters. However, we found that only 10% of the lambdas needed more than 2 parameters.

In functional programming, the need for larger arity is handled by currying, i.e., by converting functions of high arity to a chain of functions that accept only one parameter and return a function that partially applies that single parameter on the functions' body. To detect instances of currying, we first captured all the functional interfaces (built-in

Table 2.1: Reasons for defining a custom functional interface

Reason	**Description**	% **Interfaces**[1]	% **Lambdas**[2]
Throwing exceptions	The single abstract method of the functional interface needs to throw an exception, which is not handled by built-in functional interfaces.	23.01	42.04
Extending other interfaces	Functional interface extends another interface (e.g., `Serializable`)	25.15	63.53
Using annotations	Annotations (e.g., `@Nullable`) are defined for parameters/return type of the single abstract method or at method-declaration-level.	14.78	4.41
Default methods in the interface	Functional interface needs to have default methods besides the single abstract method.	12.95	2.61
Static methods in the interface	Functional interface needs to have static methods besides the single abstract method.	3.96	1.06
Static fields in the interface	Functional interface declares static fields besides the single abstract method.	1.98	1.01
External API SAM interface	The functional interface is declared in an external library to which the code depends, and thus cannot be migrated to built-in functional interface.	19.51	10.00
Other	Cases where a built-in functional interface could be used instead, but the reason for not doing so cannot be not known from the source code.	54.42	56.39

[1] Shows the percentage of *custom functional interfaces* introduced due to each reason. Note that, the percentages are not supposed to sum up to 100%, since there could be a custom functional interface that extends a built-in functional interface and at the same time declares a default method as well.
[2] Shows the percentage of all *lambdas* that were instantiated from the custom functional interfaces.

and custom) used in each project, using the binding information of the lambdas defined in that project. Then, we queried the ASTs of the project for identifying functional interfaces which accept one parameter and return another functional interface, which in turn accepts one parameter, but can return anything. Each implementation of these functional interfaces in the form of a lambda expression is essentially an application of currying. We found that only 1% of the total lambdas use currying.

In order to find other reasons for defining custom functional interfaces, we manually investigated 656 custom functional interfaces from a randomly-selected subset of the dataset (particularly, covering 32,268 lambda expressions), and summarized the results in Table 2.1. Notice that the most prominent reasons that *force* the developers to use custom functional interfaces are the lack of exception handling in the Java's built-in functional interfaces, and the need for extending other interfaces (e.g., `java.io.Serializable`).

Particularly, it turned out that 62.91% of the custom functional interfaces of which the single abstract method needed to throw a checked exception (accounting for 14.48% of all the studied functional interfaces), were exactly similar to built-in functional interfaces

Table 2.2: Where are lambda expressions used in code?

Parent type	**Frequency**	**Percentage**
Method invocation	87821	87.35
Initializer of variable declaration fragment	5165	5.14
`return` statement	3930	3.91
Class instance creation	2328	2.31
Assignment	680	0.68
`super()` constructor invocation	280	0.28
Lambda expression[1]	223	0.22
Enum constant declaration	54	0.05
Conditional expression	32	0.03
Array creation	27	0.03
Total	**100,540**	**100%**

[1] Cases like `a -> b -> c`, i.e., nested lambda expressions.

with an additional requirement to handle exceptions. In other words, these custom functional interfaces could be replaced with the built-in ones, if they had exception handling capability. We observed very little use of third-party functional interfaces (i.e., external libraries) to handle exceptions. We hypothesize that the developers find it easier to maintain their own checked functional interfaces, since they will have the freedom to extend other interfaces, add default methods, annotations, and static methods/fields.

In Table 2.1, for a significant portion of functional interfaces (i.e., 54.42%) we were not able to understand from the source code why the developers did not use built-in functional interfaces, while they definitely could (i.e., the single abstract method defined in the interface did not throw an exception, had less than 3 parameters, and there was no default or static methods or static fields declared in the interface). We can hypothesize different reasons for this behavior, for example, the developer might be unable to use a built-in functional interface because it is declared in legacy code that is difficult to be migrated, or because it is part of the public API of the system that should remain intact. Another possible reason is that the developer seeks a more expressive name for the interface, e.g., `Processor` instead of the built-in `Consumer`.

We were interested to see in what kind of expression or statement lambda expressions are used in the code. This can give us some insight into *why* lambdas are used, which will be further complemented by the qualitative study. Table 2.2 depicts the distribution of the type of the parent AST node of the lambda expressions.

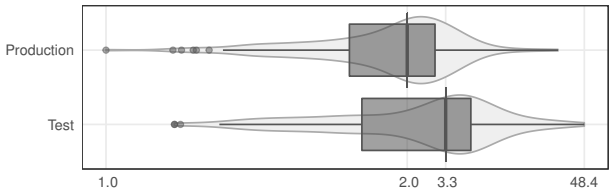


Figure 2.1: Density of lambdas in production and test files

Notice that most of the lambdas (89.94% accumulated) are passed as arguments to method invocations, class instance creations and super constructor invocations.

The advantage of using lambdas at these locations is *behavior parameterization*, an important technique to eliminate duplicated code fragments having some behavioral differences. A recent study investigated the applicability of lambda expressions on a large dataset of Type-2 clones (i.e., structurally/syntactically identical code fragments with variations in identifier names, literal values, and types) and Type-3 clones (i.e., copied fragments with statements changed, added or removed in addition to Type-2 differences), and found that 58% of them could be merged and parameterized (i.e., refactored) only by using lambda expressions [140]. Moreover, behavior parameterization was more applicable for the clones located in test code than production code (72% vs. 51% applicability).

Therefore, we studied the location of lambdas in the source code. We found that 60% of the lambdas are used in test code as opposed to production code. Figure 2.1 shows the distributions of the density of lambdas, as measured by the number of lambdas per file, corresponding to the production code and test code of the examined projects, excluding the files not containing lambdas. The violin plots show that the density of lambdas in the test code is significantly higher than the production code. To assess if there is a statistically significant difference, we applied the Wilcoxon Signed-Rank test on the paired samples of lambda density in production and test code for each project. The test rejected the null hypothesis that the density of lambdas in production code is more than test code at the significance level of 5% (p-value = 1.835×10^{-10}).

We used the *Hodges-Lehman estimator* to quantify the difference between the density of lambda expressions in the test and production files, as it is appropriate to be used with the Wilcoxon Singed-Rank test. The value turned out to be 1.15, which is equal

to the estimated median of the difference between the density of lambdas in a sample taken from the test files and a sample from the production files.

RQ Conclusions: Developers use built-in functional interfaces inefficiently. Java developers are forced to create custom functional interfaces for different reasons, e.g., where throwing checked exceptions or extending other functional interfaces is needed. However, developers do not use techniques from functional programming, like *currying*, to avoid creating custom functional interfaces.

2.3 Implications

I1. In most cases, Java developers prefer to use generic functional interfaces instead of specialized ones for primitive types. This could be due to the lack of awareness of all the available options, or the tediousness of finding the correct type of functional interface among a long list of options. Thus, developers use auto-boxing unnecessarily, which degrades the performance of the software. The Java compiler could perhaps optimize the code by replacing general functional interfaces to more specialized ones at bytecode level.

I2. We found that 42% of the custom functional interfaces implemented by developers contained methods throwing a checked exception. 63% of these functional interfaces meet the input/output requirements of the existing Java built-in functional interfaces. However, they cannot be currently replaced with built-in functional interfaces because none of them supports exception throwing. This finding calls for Java API designers to extend built-in functional interfaces in order to support exception throwing. Currently, some external libraries [139] add this missing feature to functional interfaces.

I3. The under-utilization of the Java built-in specialized functional interfaces supporting primitive types (RQ1), clearly points out the need for tools that can convert generic functional interfaces to more specialized ones, thus avoiding unnecessary auto-boxing.

I4. We found that Java projects use currying, a fundamental functional technique, sparingly. Since a developer needs to explicitly specify the type of functional interfaces

involved in currying, this becomes very verbose. Language designers can make the syntax currying-friendly by applying type inference techniques [113].

I5. RQ1 highlights that 10% of the studied lambdas use custom functional interfaces requiring more than 2 parameters. Such cases could be alternatively implemented with currying of existing built-in functional interfaces, thus avoiding the need to create custom interfaces. Automated code assistance and refactoring could suggest appropriate built-in functional interfaces for currying, and convert existing custom functional interfaces to alternative implementations taking advantage of currying.

2.4 Summary

Java 8 retrofitted lambda expressions, one of the hallmarks of functional programming. However, without understanding the adoption of lambdas among Java developers, there can be no improvement. In this study we use complementary empirical methods (mining 241 software repositories containing over 100,000 lambdas, and conducting 97 firehouse surveys) to answer fundamental questions such as *when, what, who, how*, and *why* are Java developers adopting lambdas. We found that Java developers are increasingly adopting lambdas for replacing anonymous classes, replacing external with internal iterators, and for behavior parameterization. However, we found they sometimes do this inefficiently and are not leveraging the full power of the functional paradigm. We found that a core subset of developers are introducing most of the lambdas, mostly manually. These results call for improvements on the language design and more intelligent tool support, such as detecting inefficient use and suggesting appropriate code changes, and judiciously replacing inheritance with composition.

This formative study uncovers multiple directions for new research - 1. developing refactoring tools to eliminate the misuses of lambda expressions; remove auto-boxing from lambda expressions by using specialized primitive functional interfaces instead of the generic ones. 2. assist developers at exception handling when using lambda expressions 3. identify and eliminate unintended side-effects We hope that this chapter (and the entire study [89]) inspires a symbiotic ecosystem where researchers, language designers, and tool builders work together to increase the adoption of functional constructs.

Chapter 3: Mining Type Changes

Chapter 2 showed that developers frequently misuse lambda expressions by using generic functional interfaces instead of the specialized primitive functional interfaces. To eliminate such misuses of lambda expressions would require to perform a *type change* from the misused generic interface to the appropriate specialized alternative. However, if type change techniques could be configured to eliminate these misusages it could enhance the quality (and performance) of the program, without any manual burden. Application developers would just have to invoke the tool. However, automating type changes is very challenging. Therefore, we first perform an empirical study to understand the type changes performed by Java developers in practice.

This chapter describes common terminology used in the remaining chapters. We are assuming that the reader is familiar with the object-oriented terminology (e.g. classes, primitives, class inheritance, generics, composition or instance methods). First, the chapter introduces the problem of type changes, the challenges associated to mining them from source code history and the techniques we developed to overcome these challenges. Next, the chapter answers six research questions about the practice of type changes based on the fine-grained changes collected from a large corpus of 130 large open source projects. Finally, the chapter presents an *actionable, empirically justified set of implications* of our findings from the perspective of four audiences: Researchers, Tool Builders, Language Designers, and Developers.

3.1 Introduction

A *type change* is a common program transformation that developers perform for several reasons: *library migration* [4, 69, 134] (e.g., `org.apache.commons.logging.Log`→`org.slf4j.Logger`), *API updates* [24, 31] (e.g., Listing 3.1), *performance* [40, 44, 45] (e.g., `String`→`StringBuilder`), *abstraction* [138] (e.g., `ArrayList`→`List`), *collection properties* [41, 42] (e.g., `LinkedList`→`Deque`), *concurrency* [36] (e.g., `HashMap`→`ConcurrentHashMap`), *secu-*

rity [47] (e.g., `Random`→`SecureRandom`), and *maintainability* [27] (e.g., `String`→`Path`). To perform a type change, developers change the declared type of a program element (local variable, parameter, field, method return type) and adapt the code referring to this element (within its lexical scope) to the API of the new type. Due to assignments, argument passing, or public field accesses, a developer might perform a series of type changes to propagate the type constraints for the new type.

Listing 3.1: Type Change example

```
− SimpleDateFormat formatter= new SimpleDateFormat("yyyy");
+ DateTimeFormatter formatter= DateTimeFormatter.ofPattern("yyyy");
− Date d = formatter.parse(dateAsString);
+ LocalDate d = LocalDate.parse(dateAsString,formatter);
```

In contrast to refactorings that are heavily automated by all popular IDEs [9, 10, 50, 68], developers perform the vast majority of *type changes* manually. Ideally, type changes should be automated in a similar way as a developer renames a program element in an IDE, although we recognize that it is a far more challenging problem. The first step to advance the science and tooling for automating type change is to thoroughly understand its practice.

Most of the prior work studied type changes in the context of other evolution tasks such as API updates [24, 31, 32] and library migration [4, 69, 134]. However, there is a gap in understanding type changes in the general context. This gap in knowledge negatively impacts four audiences:

1. **Researchers** do not have a deep understanding of type changes and the role they play in software evolution. Thus, they might not fully understand and support higher level tasks, such as automated program repair [16, 96] that are composed from type changes.

2. **Tool builders** do not have an insight into the practice of type changes. Thus, they (i) are not aware if the type changes they automated [50, 67, 68, 74, 138] are representative of the ones commonly applied in practice, (ii) fail to identify new opportunities for developing automation techniques.

3. **Language and Library Designers** continuously evolve the types their clients use. However, designers are not aware of *what* types are commonly used and *how* the clients

adapt to new types. Without such knowledge they cannot make informed decisions on how to improve or introduce new types.

4. **Developers** miss educational opportunities about common practices applied when changing types in other projects, which could benefit their own projects.

To fill this gap, in this chapter we present the first longitudinal, large-scale, and most fine-grained empirical study on type changes performed in practice. We analyze the commit histories of 129 large, mature, and diverse Java projects hosted on GitHub. To do this, we developed novel tools, which efficiently and accurately mined 416,652 commits and detected 297,543 instances of type changes. We thoroughly evaluated our tools and they have 99.7% precision and 94.8% recall in detecting type changes. To advance the science and tools for type change automation, we use this rich and reliable dataset to answer six research questions:

RQ1 *How common are type changes?* We found 35% more instances of type changes than renames. Given that type changes are so common, it is worth to investigate how they can be automated.

RQ2 *What are the characteristics of the program elements whose type changed?* We found that 41.6% of the type changes are performed upon *public* program elements that could break the code. In addition, developers frequently change between *Internal* types. The current tool support for such changes is non-existent.

RQ3 *What are the edit patterns performed to adapt the references?* Developers often adapt to the primary type change by performing a secondary type change. For example, in Listing 3.1 type change `SimpleDateFormat`→`DateTimeFormatter` triggers a secondary type change `Date`→`LocalDate`. However, current techniques cannot infer mappings for such cascading type changes.

RQ4 *What is the relationship between the source and target types?* Among others, we found that in 73% of type changes the types are not related by inheritance. In contrast, most of the current IDEs automate type changes for types related by inheritance (e.g., Replace Supertype where Possible [68, 74, 138]). This reveals another important blind spot in the current tooling.

RQ5 *Are type changes applied globally or locally?* In 62% of cases developers perform type changes locally. In contrast, the current tools [58, 67, 74, 103, 138] perform a *global migration* in the entire project. This shows that the tool builders do not invest resources where automation is most needed.

RQ6 *What are the most commonly applied type changes?* From our entire corpus, we filter type changes that developers perform in at least two projects. We found that these 1,452 type changes represent 2% of all type change patterns, yet they are responsible for 43% of all type change instances. Tool builders should prioritize automating these popular type changes. Developers and educators can learn from these common practices.

Our findings have actionable implications for several audiences. Among others, they (i) advance our understanding of type changes which helps our community improve the science and tools for software evolution in general and specifically type change automation, (ii) help tool designers comprehend the struggles developers face when performing type changes, (iii) provide feedback to language and API designers when introducing new types, (iv) identify common practices for developers to perform type changes effectively, and (v) assist educators in teaching software evolution.

This chapter makes the following contributions:

Questions: To the best of our knowledge, this is the first large-scale and most fine-grained (at commit level) empirical study of type changes in Java. We answer six research questions using a corpus of 297,543 type changes. This makes our findings representative.

Tools: We developed novel tools to efficiently detect type changes from a corpus of 416,652 commits. We also manually validated our tools and show they have high precision (99.7%) and recall (94.8%). To help our community advance the science and practice of type changes, we make the tools and the dataset available at [72, 73, 75].

Implications: We present an *actionable, empirically justified set of implications* of our findings from the perspective of four audiences: Researchers, Tool Builders, Language Designers, and Developers.

3.2 Research Methodology

In the rest of the chapter we refer to the tuple `(SourceType,TargetType)` as a *Type Change Pattern* (TCP). A *Type Change Instance* (TCI) is applying a TCP on a program

element (i.e., variable, parameter, field, method declaration) in a commit and adapting its references.

3.2.1 Subject Systems

Our corpus consists of 416,652 commits from 129 large, mature and diverse Java projects, used by other researchers [89] to understand language constructs in Java. This corpus [89] is shown to be very diverse, from the perspective of LOC, age, commits, and contributors. This ensures our study is representative. It is also large enough to comprehensively answer our research questions. The complete list of projects is available online[1].

In this study, we consider all commits in the epoch January 1, 2015 – June 26, 2019, because researchers observed an increasing trend in the adoption of Java 8 features after 2015. Java 8 introduced new APIs like `FunctionalInterface`, `Stream`, `Optional` and enhanced the `Time`, `Collection`, `Concurrency`, `IO` and `Math` APIs. Thus, we use these particular projects and their commits in this particular epoch, because it allows us to collect and study type changes involving the new ($\geq$`Java 8`) built-in Java types. We excluded all merge commits, as done in other studies [126], to avoid having duplicate result.

3.2.2 Static Analysis of Source Code History

3.2.2.1 Challenges:

Most refactoring detection tools [34, 115, 154] take as input two fully built versions of a software system that contain binding information for all named code entities, linked across all library dependencies. However, a recent study [143] shows that only 38% of the change history of software systems can be successfully compiled. This is a serious limitation for performing our longitudinal type change study in the commit history of projects. It poses a threat to the external validity of our empirical study, since a small number of project versions can be compiled successfully for extracting type changes. Since the majority of versions cannot be compiled, we would not be able to retrieve fine-grained details of the types, thus making it challenging to understand the characteristics

[1] http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/projects.html

of the types involved in a type change. If we built 38% of the commits in the project history, it would be time and resource consuming. This would prevent our study to scale beyond a few thousand commits.

We overcome these challenges for performing our fine-grained and large-scale study in two ways. First, we extended the *state-of-the-art* refactoring detection tool, RefactoringMiner [141, 142], to accurately and efficiently detect type changes at commit-level. We built upon RefactoringMiner, because it has been shown to have a superior accuracy and faster execution time [141, 142] than competitive tools also operating at commit-level, such as RefDiff 1.0 [125] and RefDiff 2.0 [127]. Second, we created a novel tool, TypeFactMiner, which accurately and efficiently retrieves detailed information about the types involved in the type change, without requiring to build the software system.

3.2.2.2 Detecting TCIs:

Detecting accurately a TCI is not straightforward, as these changes can get obfuscated by the other changes (i.e., overlapping refactorings) in a commit, where methods and classes containing the TCI get moved, renamed or removed. For this purpose, we extend **RefactoringMiner** [141, 142] to detect 4 kinds of type changes, namely (i) *Change Variable Type*, (ii) *Change Parameter Type*, (iii) *Change Return Type*, and (iv) *Change Field Type*. RefactoringMiner uses a state-of-the-art *code matching algorithm* to match classes, methods and statements inside method bodies, and accurately detect refactorings at commit level. It also records AST node *replacements* when matching two statements, which we utilize to infer the aforementioned type change kinds.

To evaluate the precision and recall of RefactoringMiner, we extended the oracle used in [141], which contains true refactoring instances found in 536 commits from 185 open-source GitHub projects, with instances of the four type change kinds. To compute precision, the first two authors manually validated 1843 TCIs reported by RefactoringMiner. Most of the cases were straightforward, and thus were validated individually, but some challenging cases were inspected by both authors to reach an agreement. To compute recall, we need to find all true instances for the 4 type change kinds. We followed the same approach as in [142] by executing a second tool, namely GumTree [51], and considering as the ground truth the union of the true positives reported by Refac-

toringMiner and GumTree. GumTree takes as input two abstract syntax trees (e.g., Java compilation units) and produces the shortest possible edit script to convert one tree to another. We used all *Update* edit operations on types to extract TCIs and report them in the same format used by RefactoringMiner. Table 3.1 shows the number of true positives (TP), false positives (FP), and false negatives (FN) detected/missed by RefactoringMiner. Based on these results, we conclude that our extension of RefactoringMiner has an almost perfect precision and a very high recall (94% - 96.5%) in the detection of TCIs. Thus, our results in Section 3 are reliable.

Table 3.1: Precision and recall of RefactoringMiner

Refactoring Type	TP	FP	FN	Precision	Recall
Change Parameter Type	597	1	35	99.8%	94.5%
Change Return Type	386	2	14	99.5%	96.5%
Change Variable Type	649	2	42	99.7%	93.9%
Change Field Type	212	1	10	99.5%	95.5%
Average	1843	6	101	99.7%	94.8%

3.2.2.3 Detecting the Adaptations of References:

RefactoringMiner analyzes the matched statements referring to a certain variable to increase the precision in the detection of variable-related refactorings, such as variable renames. We use these references, to understand how developers adapt the statements referring to the variable/parameter/field whose type changed. For *Change Local Variable Type*, *Change Parameter Type*, and *Change Field Type*, RefactoringMiner reports all matched statements within the variable's scope referring to the variable/parameter/field on which the TCI was performed. While for *Change Return Type*, it returns all matched `return` statements inside the corresponding method's body.

If these matched statements are not identical, RefactoringMiner reports a set of AST node *replacements*, which if applied upon the statement in the parent commit would make it identical to the matched statement in the child commit. Using these AST node replacements, we extract the 11 most common *Edit Patterns (RQ3)* performed to adapt the statements referencing a variable whose type changed. RefactoringMiner reported 532,366 matched statements for 297,543 mined TCIs, creating a large data set of real world edit actions performed to adapt the references in a type change.

3.2.2.4 Qualifying Type Changes:

Analyzing only the syntactic difference of AST Type nodes is not enough to correctly detect a type change. For instance, when a TCI qualifies the declared type (e.g., `Optional<String>` → `java.util.Optional<String>`) there is no actual type change. In such cases, it is important to know the qualified name of the type before and after the TCI is applied. If `Optional` was bound to `java.util.Optional`, there is no type change. If `Optional` was bound to `com.google.common.base.Optional`, then there is one.

Moreover, to record accurately the type changes that are more commonly performed *(RQ6)* and their characteristics *(RQ2)*, we need to know the fully-qualified types that changed when a TCI was performed. For example, if the type change is `List`→`Optional`, depending upon the context (i.e., import declarations) where the types are used, `List` could correspond to `java.util.List` or `io.vavr.List` and `Optional` could correspond to `java.util.Optional` or `com.google.common.base.Optional`. Finally, knowing further details about these types, such as the fields and methods they declare, or their super types, would allow us to do an in-depth investigation of the relations between the changed types.

However, there are certain challenges in extracting the qualified types, without building the commit: (1) we have an incomplete source code of the project, because we analyze only the modified/added/deleted java files in a commit, (2) we do not have the source code of the types declared in external libraries. To mitigate these challenges we developed a novel tool **TypeFactMiner**, which efficiently and accurately infers the fully qualified name of the type to which a variable declaration type is bound.

Collecting Type Bindings from Commit History: At the core of TypeFactMiner are heuristics, which reason about the import statements, package structure, and the types declared in the entire project, similar to the ones discussed by Dagenais et al. [25]. To represent the types declarations, TypeFactMiner uses *Type Fact Graphs* (TFG), recently proposed by Ketkar et al. [74]. These are abstract semantic graphs that capture each declared class/interface/enum, the qualified signature of methods/fields it declares, and the local variables/parameters declared inside methods. For the oldest commit, which contains at least one Java file, we map all existing type declarations to a

TFG. For the subsequent commits, we incrementally update this TFG by analyzing only the added, removed, moved, renamed and modified files. This optimization allows us to scale our fine-grained study to hundreds of thousands of commits. The TFG representation allows us to infer transitive inheritance or composition relationships between types (RQ4).

Collecting Type Bindings from External Libraries: A type change often involves types that are declared in the standard Java library (e.g., `java.lang.String`) or third party libraries (e.g., `org.slf4j.Logger`). For this purpose, TypeFactMiner analyzes the bytecode of the project's library dependencies to extract information for the publicly visible type declarations (classes/interfaces/enums). For the types declared in JDK, TypeFactMiner analyses the jars contained in the `openjdk-8` release. For external dependencies, TypeFactMiner fetches the corresponding jar files for the dependencies required by the project at each commit. Since a project can contain multiple `pom.xml` files (for each module) with dependencies amongst them, TypeFactMiner generates an `effective-pom` [8] at each commit and parses this file to identify the external dependencies. It then connects all external type bindings to the nodes in the TFG of the analyzed project according to their usage.

The validity of the answers to our research questions relies upon how accurately TypeFactMiner infers the fully qualified names of the types. To compute the precision of TypeFactMiner, we create a golden standard based on the qualified names returned by the Eclipse JDT compiler. However, to obtain the qualified names the commits have to be built, which is time consuming because each commit might require a different build tool version, Java version, or build command. This prevented us from randomly sampling commits from our dataset. So, we selected 4 projects, namely `guava`, `javaparser`, `error-prone` and `CoreNLP` and automated the process to build each commit (and its parent) that contained a TCI. We successfully built 467 commits that contained 4715 TCIs. For the program elements involved in the TCIs, we get the qualified types from Eclipse JDT compiler, which we use as the golden standard. We found that TypeFactMiner correctly inferred the qualified names for the types involved in 4652 TCIs (i.e., 98.7% precision).

Out of 428,270 TCIs found in 40,865 commits, TypeFactMiner filtered out 130,727 TCIs, where (i) the corresponding types were renamed or moved to another package

within the examined project (i.e., an internal Rename/Move Type refactoring triggered the type change) (ii) no actual type change happened (i.e., a non-qualified type changed to qualified and vice-versa), leading to a total of 297,543 *true* TCIs. The **efficient** and **accurate** tools we created and validated allow us to collect *extensive* and *reliable* results, to empirically justify our implications.

3.3 Results

3.3.1 How common are type changes?

To provide some insight about how commonly developers perform type changes, we compare this practice with another commonly applied source code transformation, namely the renaming of identifiers (i.e., Rename refactoring) [11, 100]. Such a comparison is feasible, because both type change and rename can be performed on the same kind of program elements, i.e., local variables, parameters, fields, and method declarations. All these program elements have a name and a type (return type in the case of method declarations). Thus, it is possible to make a direct comparison between the number of type changes and renames for the same kind of program elements to understand which practice is more common. Morever, RefactoringMiner detects rename refactorings with an average precision of 99% and recall of 91% [142], which are very close to the average precision/recall values reported in Table 3.1 for type changes, allowing for a fair comparison of the two practices.

Table 3.2: Mined Source Code Transformations

	Variable	**Parameter**	**Field**	**Method**	**Total**
Type Change	83,393	93,229	48,279	72,642	297,543
Rename	53,416	63,612	30,852	71,476	219,356
Δ Percentage	+56.1%	+46.6%	+56.5%	+1.6%	+35.6%

Table 3.2 shows the number of type changes and renames on variables, parameters, fields, and methods that RefactoringMiner detected in 95,576 commits. As we can see in Table 3.2, type changes are around 50% more populous than renames on variables, parameters and fields, while return type changes are slightly more populous than method renames. Moreover, we observed that 297,543 type changes occur in 40,865 commits, while 219,356 renames occur in 46,699 commits, i.e., the density of type changes is higher

(7.3 per commit) than that of renames (4.7 per commit).

> **RQ1 Conclusion:** Type changes are more commonly and frequently performed than renames. In comparison to renaming, there is negligible tool support and research for type changes.

3.3.2 What are the characteristics of the elements whose type changed?

To answer this question, we studied various characteristics of the program elements involved in TCIs. We explore characteristics, such as (i) *kind:* field, method or variable, (ii) *visibility:* public, private, protected or package, (iii) *AST Type node:* simple, primitive, array, or parameterized and (iv) *namespace:* internal, standard Java library, or third-party library.

Assume project p has n commits, and $TCI(e, i)$ is a type change instance on program element e in commit i, and x is a value for characteristic y of program elements, we define:

$$\textbf{\textit{proportion(p, x)}} = \frac{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has } x \text{ value for characteristic } y\}|}{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has any value for characteristic } y\}|}$$

Further, assume that elements(x, y, i) is the set of all program elements having value x for characteristic y in the modified files of commit i, regardless of whether their type changed or not, we define:

$$\textbf{\textit{coverage(p, x)}} = \frac{\sum_{i=1}^{n} |\{TCI(e,i) \mid e \text{ has } x \text{ value for characteristic } y\}|}{\sum_{i=1}^{n} |\text{elements}(x,y,i)|}$$

Only studying the proportions of the different values a characteristic can take (e.g., the *visibility* characteristic takes values `public`, `private`, `protected`), may result in misleading findings, because it does not take into account the underlying population distribution. For example, by studying proportions we could find that most TCIs are performed on `public` elements, just because there are more `public` elements in the source code of the examined projects. Therefore we study the *coverage* of the different values a characteristic can take, with respect to all program elements having the same value for that characteristic (whose type did or did not change). The *project-level* distributions for the proportion and coverage of the different values of a characteristic are shown as Violin plots. To assess if there is a statistical difference among these

distributions, we perform the Kruskal-Wallis test (the result of the test is shown on top of each Violin plot). A p-value ≤ 0.05 rejects the null hypothesis that the medians of the distributions are equal. To compare two distributions with visibly close medians, we report p-values obtained from the pair-wise post-hoc Dunn's test.

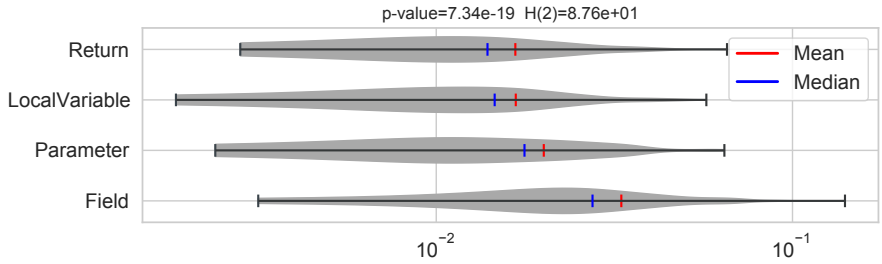


Figure 3.1: Project-level distribution of type change coverage per program element kind

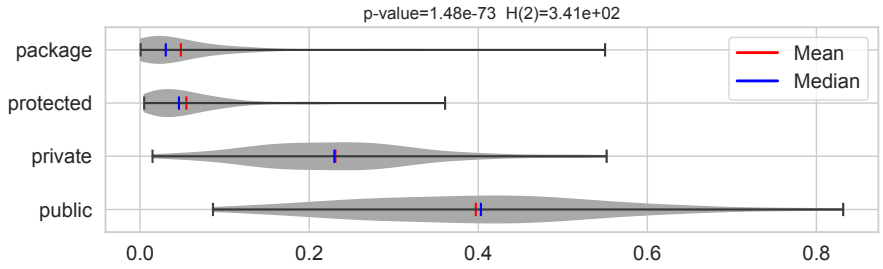
3.3.2.1 Program Element Kind:

The scope of the element on which a type change is applied determines the impact the change has upon the program. Transforming a *field*, *method return type* or *method parameters* affects the API of the program, while transforming *local variables* affects the body of a single method only [100]. Table 3.2 shows that the largest proportion of type change affects method parameters, followed by local variables. However, Figure 3.1 shows that the median coverage of performing *Change Field Type* is the largest. Our results are in congruence with the results obtained by Negara et. al [101] who surveyed 420 developers, and ranked *Change Field Type* as the most relevant and applicable transformation that they perform. They also report that *Change Field Type* is the most highly desired feature in IDEs.

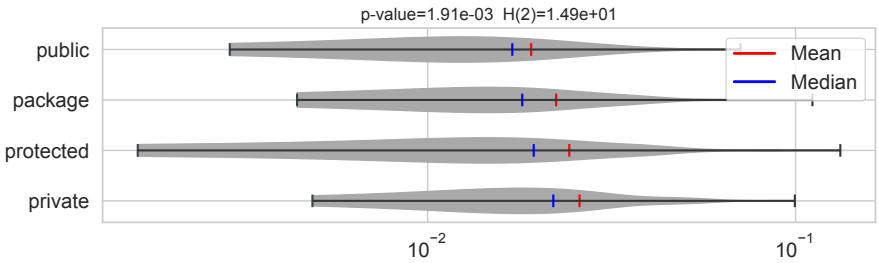
3.3.2.2 Program Element Visibility:

If a type change affects the signature of a *package* visible method, a developer should update the call sites of this method within the same package. However, if this method is *public* visible, a developer should update the call sites of this method in the entire program, but more importantly this type change could introduce backward incompatibility

for the clients of the library. Figure 3.2 shows the proportion and coverage for the access levels - *public*, *private*, *protected* and *package*.



(a) Proportion of type change instances

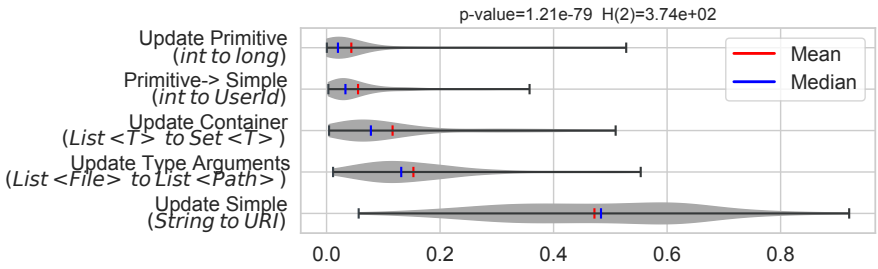


(b) Coverage of type change instances

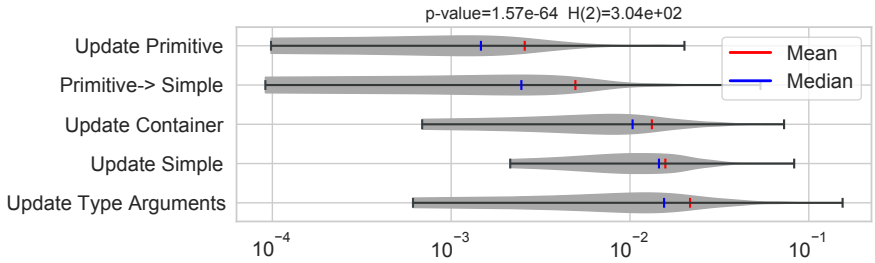
Figure 3.2: Project-level distribution per visibility kind

Figure 3.2a shows that type changes are most commonly applied on `public` program elements. However, in Figure 3.2b the coverage of type changes on `public` elements is lower than `private` elements (p-value=0.0013). This shows that, although the raw number of type changes on `private` elements is less than `public` elements, developers tend to change more often the types of `private` elements compared to `public` ones. This indicates that developers are more cautious when performing type changes on `public` elements, possibly taking into account backward incompatibility issues.

Researchers [24, 31] have thoroughly studied the impact of different kind of changes on software evolution. Cossette et al. [24] categorize type change as a hard to automate breaking change. Dietrich et al. [31] categorize a change based on binary and source code incompatibility. We analyze the type changes that are performed on `public` elements and observe that 14.2% introduce binary but no source incompatibility, while the remaining introduce both. Below we report the occurrences of type changes applied on `public`



(a) Proportion of type change instances



(b) Coverage of type change instances

Figure 3.3: Project-level distribution per AST Type node kind

elements based on the proposed categories in [31]:

1. **Binary and Source Incompatible:** We found 106,329 TCIs (35.8%) that can potentially introduce breaking changes.

2. **Binary Incompatible but Source Compatible:** This interesting phenomenon appears in Java programs when the code compiles, but results in a runtime failure, due to mismatch of rules between compiler and JVM. We found the following instances in our corpus: *method return type replaced by subtype* (4,249), *method parameter type replaced by supertype* (6,437), *primitive narrowing of return type* (1,373), *wrapping and unwrapping of primitive parameter and return types* (1,663), and *primitive widening of method parameters* (3,258).

3.3.2.3 Program Element AST Type Node:

Java developers have to explicitly define the type for all declared methods and variables. Java 8 allows nine kinds of syntactic structures to express the declared type of elements [66]. For example, Simple (`String`), Parameterized (`List<Integer>`), Primitive (`int`), Array (`int[]`).

Listing 3.2: Updating the argument of a parameterized type

```
– IgniteBiTuple<String, AtomicLong> m = getTuple();
+ IgniteBiTuple<String, LongAdder> m = getTuple();
– AtomicLong l = m.getValue();
+ LongAdder l = m.getValue();
```

According to Figure 3.3a, *Simple Types* are more commonly changed than other AST Type nodes. However, in Figure 3.3b, we can observe that the coverage of changing the *Type Arguments* of parameterized types is the most (p-value=4.09×10^{-32}). Changing the *Type Arguments* of parameterized types is a more complex task than changing *Simple Types*, because there are additional type changes that propagate through the parameterized container. For example, in Listing 3.2 to perform the change `IgniteBiTuple<String,AtomicLong>`→`IgniteBiTuple<String,LongAdder>` one would have to propagate the type changes to the call sites of method `Map.Entry.getValue()`, because `IgniteBiTuple` implements the `Map.Entry` interface. Propagating such changes requires inter-procedural points-to and escape

analysis, which is not supported by any current tool automating type changes.

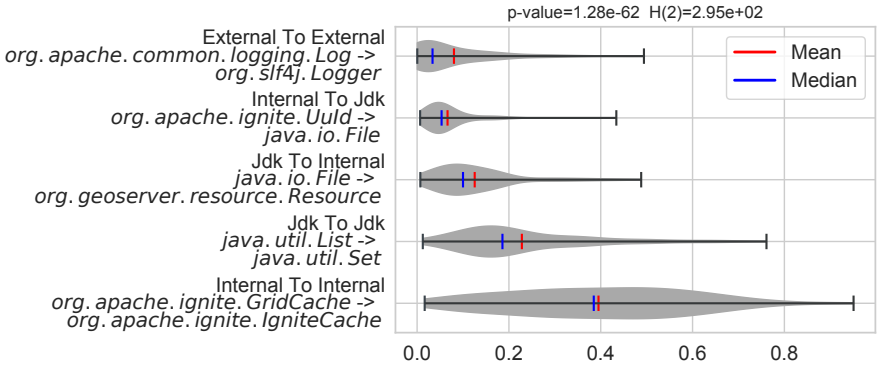
3.3.2.4 Program Element Namespace:

We categorize program elements based on the **relative location** of the source and target types with respect to the project under analysis. We find the fully qualified name of each type using TypeFactMiner, and label it as: (i) ***Internal*** (type declared in the project), (ii) ***Jdk*** (type declared in the standard Java library), or (iii) ***External*** (type declared in a third party library). We assume that developers can perform more easily type changes involving *Internal* than *External* types, as they are more familiar with the types defined internally in the project, or can ask co-developers in the project who have more expertise on these internal types. On the other hand, type changes involving *External* types are more difficult to perform, as developers need to study external documentation, which might be outdated or unavailable, or refer to Q&A forums for more information.

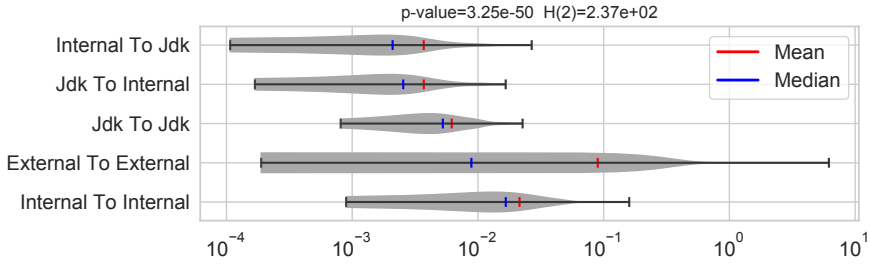
For *Simple Type* changes, we qualify the types before and after the change. For *Type Argument* changes to parameterized types, we qualify the changed type arguments (e.g., `List<File>`→`List<Path>`, the source type is `java.io.File` and the target type is `java.nio.file.Path`). For composite type changes (e.g., `List<Integer>`→`Set<Long>`), we qualify the base type changes (e.g., `java.util.List`→`java.util.Set` and `java.lang.Integer`→`java.lang.Long`).

Figure 3.4a shows that developers most commonly change `Internal`→`Internal` and `Jdk`→`Jdk` types. Type changes between `External` types are rarely performed (5.14%), of which only 27.8% have source and target types defined in different external libraries. This confirms the findings of Teyton et al. [134], who conclude that third-party library migration is not a common activity.

However, in Figure 3.4b, we can observe that the median of `External`→`External` type change coverage is greater than that of `Jdk`→`Jdk` (p-value=5.3×10^{-35}). In addition, the mean of `External`→`External` type change coverage is the largest. To further understand this distribution, we identify outliers using the Q3 + 1.5 × IQR rule. We investigate the TCIs performed in 21 outlier projects and find that developers perform hundreds of such `External`→`External` TCIs. For example, we found 1902 `org.apache.common.Log`→`org.slf4j.Logger` TCIs in 8 projects for library migration and 1254 TCIs in 7 projects to update from `google.protobuf-2` to `google.protobuf-3`.



(a) Proportion of type change instances



(b) Coverage of type change instances

Figure 3.4: Project-level distribution per namespace kind

Table 3.3: Mined edit patterns

Description	%TCI	Example
Rename variable	54.85%	`String filepath→ File file`
Rename Method call	7.09%	`applyAsLong→applyAsDouble`
Modify arguments	2.70%	`apply(id)→apply(usr.getId())`
Modify Method call	25.69%	`f.exists()→Files.exist(p)` `s.length()→s.get().length()`
Replace with Method call	0.82%	`new Long(5)→Long.valueOf(5)`
(Un)Wrap argument	0.99%	`read(p)→read(Paths.get(p))`
Update Literal	0.51%	3→3L or "1"→"1.0"
(Un)Cast	0.13%	5/7 →(double)5/7
Cascade same types	12.06%	See Listing 3.2
Cascade different types	5.81%	See Listing 3.1
Assignment↔Call	0.26%	`b = true↔b.set(true)`

The results also show the importance of inferring the type-mappings to perform a library migration or update. Migration mapping mining techniques [3, 26, 120, 151] have focused on mining method-level mappings and have missed the type-mappings across the libraries.

RQ2 Conclusion: (i) 41.6% of type changes affect *public* elements, introducing binary and/or source incompatibilities. (ii) Updating *Type Arguments* of parameterized types has the largest type change coverage; however, the current state-of-the-art tools do not support the changes that need to be propagated.

3.3.3 What are the edit patterns performed to adapt the references?

From the 297,543 TCIs mined, we found that developers applied some edit pattern to adapt the references in 130,331 TCIs (43.8%). To further shed light on these cases, in Table 3.3 we report the percentage of TCIs for different edit patterns. We observe that for 54.85% of TCIs with edited references, developers rename the program element whose type changed (e.g., `String filepath` → `File file`). This makes sense as developers try to use variable names that are intention-revealing. This makes it easy to understand and maintain the program because the names reflect the intention of the new type. Arnaoudova et al. [11] were the first to observe this qualitatively, but we are the first to measure this quantitatively.

The second most applied pattern is to adapt method calls by updating the name, modifying the call chain structure, or modifying the receiver or the arguments. This requires inferring method-mapping between the source and target types and the type-mapping between the return type and arguments of the methods. This result motivates the work that infer API mappings [120].

The third most commonly applied pattern to adapt references is *cascade* type changes, which involves additional type changes to other places in the code. For example, in Listing 3.1 the type change `SimpleDateFormat`→`DateTimeFormatter` applied to variable `formatter` triggers another type change `Date`→`LocalDate` (i.e., cascade type change) applied to variable `d`. We found that in 12.06% of TCIs developers perform a *cascade* type change, which is similar to the original type change, while in 5.81% of TCIs the *cascade* type change is different from the original. To perform such cascade type changes, the replacement rules must cover all potential type changes between the source and target type. This requirement was initially discovered by Li et al. [85], but our study is the first one to empirically show that this happens often in practice.

No edit patterns was applied to adapt the code to the type change (in 56.2% TCIs). We found that (i) the variables whose type changed are passed as arguments to method calls before getting actually consumed (i.e., some API method is called), (ii) the type changes involved wildcard types (`T` → `? extends T`), hierarchically related types, primitive widening/narrowing, or (un)boxing.

RQ3 Conclusion: In 54.85% type changes, developers rename the variables to reflect the changed type. In 17.87% of type changes developers perform a *cascade* type change involving the same or different types.

3.3.4 What is the relationship between the source and target types?

To answer this question, we check if the source and target types are related by (i) inheritance i.e., they share subtype or supertype relationship, or share a common super type (other than `java.lang.Object`), or (ii) composition i.e., one type is composed of the other.

We found that in 7.5% of the TCIs the types are composition-related. In 27.08% of the TCIs the types have an inheritance relationship. The tools [50, 67, 138] for performing type changes have exclusively focused on parent-child relationships (e.g., Use

Supertype Refactoring). However, we found that 44.56% of the inheritance-related type changes actually have a sibling relationship (e.g., `List`→`Set`). This highlights a blind spot in the current tooling for 85.1% type changes, where the source and target types are siblings, composition-related, or have no relationship.

Composition and Inheritance are two ways of designing types based on *what they are* or *what they do*. The seminal work on Design Patterns by the Gang of Four [56] often advocates composition over inheritance. We are interested to find the effect of this design choice when performing a type change. Thus, we define:

Adapted Statement Ratio $= \frac{|\text{Adapted Statements}|}{|\text{Referring Statements}|}$

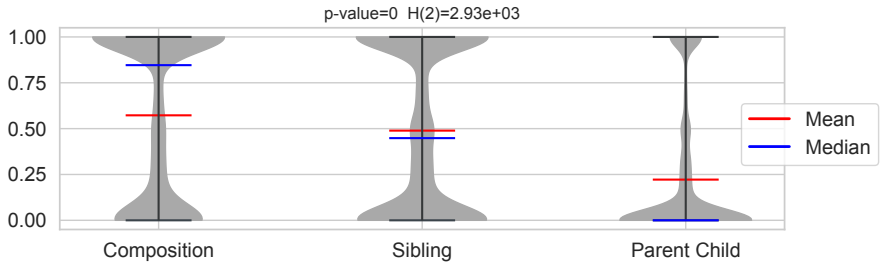


Figure 3.5: Distribution of adapted statement ratio

Referring Statements is the set of matched AST statement pairs within the scope of the variable on which a type change was applied, that reference this variable. These statements can be considered as the statements belonging in the def-use chain [109] of the variable whose type changed. *Adapted Statements* is the subset of *Referring Statements*, where an edit was performed to adapt to the type change. This ratio is (i) non-negative and normalized within the interval [0, 1], (ii) has true null value of *0*, when no edits are performed (e.g., `ArrayList`→`List`, where both types share a similar API), (iii) has a maximum value of *1* when all *Referring Statements* are edited (e.g., `java.io.File`→`org.neo4j.io.DatabaseLayout`, where the two types share no common API).

Figure 3.5 shows the distribution of *adapted statement ratio* corresponding to TCIs when the source and target types have a composition, sibling, and parent-child relationship. The plots show that the median *adapted statement ratio* is higher when the source and target types have a composition relationship than when they have sibling or parent-

child relationship. Moreover, the median *adapted statement ratio* is higher when the source and target types have a sibling relationship than when they have a parent-child relationship.

To gain further insight into this, we analyze the edit patterns applied w.r.t. the relationship of the source and target types. Table 3.4 shows that developers rename identifiers more often when the source and target types have a composition relationship than a hierarchical relationship. Since, identifier names represent defined concepts [118], one possible explanation is that developers assign names to program elements based on *what they represent* and not based on *what they do*. Performing type change between hierarchically related types does not change what the element represents (e.g., `ArrayList` is a `List`, whereas `List` and `Set` are both `Collections`), while this is not always true when types are related by composition (e.g., `File` and `DatabaseLayout` represent different concepts).

Furthermore, developers modify method calls more often when the source and target types are related by composition. For example, when `neo4j` developers performed the type change `File` → `DatabaseLayout`[2], they consistently replaced the references to variables representing directories with getter calls `layout.getDirectory()`.

Sibling types often provide different methods (e.g., `List` provides methods to add and remove an element in a specific index through methods `add(int index, E element)` and `remove(int index)`, while `Set` does not offer such functionality). They also provide similar methods through their common supertype. Table 3.4 shows that, modifying or renaming a method invocation is a common edit pattern, when adapting to a type change between *sibling* types. In fact, previous researchers who proposed techniques to perform such type changes, identify one-to-one and one-to-many method mappings to modify or rename method invocations. For example, Dig et al. [36] replace `ConcurrentHashMap`

[2]http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/neo4j/tci_project3859.html

Table 3.4: Edit patterns to adapt TCIs grouped by relationship

Edit pattern	**Composition**	**Sibling**	**Parent-Child**
Rename Identifier	77.36%	45.03%	40.9%
Rename Method Call	3.91%	9.9%	6.4%
Modify Method Call	34.26%	30.54%	24.8%
Cascade Type Change	10.28%	19.56%	8.93%

with `HashMap`, Tip et al. [138] replace `Vector` with `ArrayList`, Li et al. [85, 111] replace `HashTable` with `HashMap`, and Ketkar et al. [74] replace `Function` with `UnaryOperator`.

> **RQ4 Conclusion:** In 65.42% of type changes, the source and the target types have no hierarchical or composition relationship. Despite the advantages of using composition over inheritance, when it comes to changing types, composition requires more adaptations than inheritance.

3.3.5 How common are type migrations?

Are type changes applied globally in the form of a type migration, or selectively on specific parts of the code? How often do developers perform type migration? What are the most common migrations? Answering such questions is important for researchers and tool builders to better support the common practices.

To compute the percentage of migration for a given `SourceType` in a project, we need to count the instances where `SourceType` has been changed to any `TargetType`, and the instances where `SourceType` has not been changed in the commit history of the project. We decided to study the migration phenomenon on a `SourceType` level, instead of a type change level (`SourceType`→`TargetType`), because we found that in many cases developers change a given `SourceType` to multiple `TargetTypes` depending on the context. For example, in project `google/closure-compiler` Guava type `ImmutableEntry` has been changed in some places to `BiMapEntry` and in other places to Java's `Map.Entry` depending on desired property.

Assume project p has n commits ($1 \leq n \leq |\text{all commits in } p|$), where each of these commits contains at least one occurrence of a TCI involving `SourceType` t. Further, $TCI(e, i)$ is a TCI involving `SourceType` t on program element e in commit i, and elements(t, i) is the set of all program elements having type t in all Java files of commit i, we define:

$$\textit{coverage(p, t)} = \frac{\sum_{i=0}^{n-1} |\{TCI(e,i)\}| \quad + \quad |\{TCI(e,n) \mid e \text{ has SourceType } t\}|}{\sum_{i=0}^{n-1} |\{TCI(e,i)\}| \quad + \quad |\text{elements}(t,n)|}$$

In the formula above, elements(t, n) represents the program elements having type t in commit n of project p. If all these elements are involved in a TCI in the last commit where `SourceType` t has been changed, then `SourceType` t is *migrated* (i.e., *coverage(p, t)* $= 100\%$).

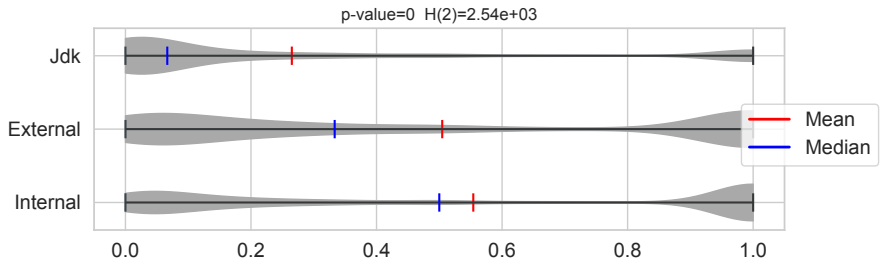


Figure 3.6: Project-level distribution of type change coverage

Figure 3.6 shows the distribution of type change coverage for three categories of `SourceType`, namely *Internal*, *External* and *Jdk* types. We can clearly observe that *Jdk* types are more selectively changed, while *Internal* and *External* types tend to be more globally changed (median = 0.51 and 0.34 respectively). In addition, *Internal* types are more globally changed than *External* types (p-value = 3.6×10^{-32}) with relatively more migrations, i.e., developers migrate *Internal* types (45.2%) more than *External* (38.3%) and *Jdk* (16.1%) types. This highlights a major blind spot in previous research on type migration [74, 85, 138] that focuses mainly upon migration between *Jdk* types (e.g. `Vector` → `ArrayList` or `HashTable` → `HashMap`).

We found that 16 projects migrated `FinalizerThread` from Jdk to `FutureTask` from Jdk, `TrustedFuture` and `InterruptibleTask` from Guava, or `LeaderSwitcher` from neo4j. The complete list of migrations[3] can be found at our website [75].

RQ5 Conclusion: In 61.71% of cases developers perform type changes in a *selective* rather than a *migration* fashion. *Type Migration* is most commonly performed on *Internal* types.

3.3.6 What are the most commonly applied type changes?

We group all 297,543 TCIs by the tuple <`SourceType`, `TargetType`>, expressing a TCP. For instance, in Listing 3.1 there are two type changes, namely <`SimpleDateFormat`, `DateTimeFormatter`> and <`Date`, `LocalDate`>, and in Listing 3.2 there is one type change <`AtomicLong`, `LongAdder`>. We found a total of 50,640 distinct TCPs. To find the most popular TCPs from our dataset, we select those that were

[3]http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/Migrations.html

Table 3.5: A few popular type change patterns

Type Change Pattern	#projects, #commits	#n	Known Reasons
int → long	(75,623)	4,600	Widening Primitive
int → double	(22, 47)	115	Conversion [49]
java.util.List → java.util.Set	(68, 281)	742	Different properties
java.util.LinkedList → java.util.DeQue	(8,8)	9	[41, 42, 43, 46, 48]
java.util.ArrayList → java.util.List	(52, 152)	560	Use Supertype
java.util.HashMap → java.util.Map	(38,112)	348	Where Possible [138]
java.util.Map → java.util.concurrent.ConcurrentMap	(37, 93)	193	Concurrency [36]
int → java.util.concurrent.atomic.AtomicInteger	(32,65)	86	
java.util.concurrent.atomic.AtomicLong → java.util.concurrent.atomic.LongAdder	(11,16)	227	Performance
java.lang.StringBuffer → java.lang.StringBuilder	(38,109)	280	[40, 44, 45]
java.util.List → com.google.common.collect.ImmutableList	(18, 66)	145	Immutability
java.util.Set → com.google.collect.ImmutableSet	(9,50)	95	[39, 46]
java.lang.String → java.nio.file.Path	(23,56)	502	Conceptual
long → java.util.Date	(2, 2)	9	Types [27]
org.apache.commons.logging.Log → org.slf4j.Logger	(8,123)	1,902	Third Party library migration [3, 69, 134]
com.mongodb.BasicDBObject → org.bson.Document	(3, 5)	45	
java.util.Random → java.security.SecureRandom	(6,6)	8	Security [47]
java.lang.String → java.security.Key	(2,2)	2	
org.joda.time.DateTime → java.time.ZonedDateTime	(5,6)	126	Deprecation [116]

performed in at least 2 projects. This results in 1,452 TCPs[4] that collectively account for 64,310 TCIs.

We found that 70.2% of the popular TCPs involve `Jdk` types. The **10 most** popular TCPs involve (i) *primitive types*: `int`, `long`, `void`, `boolean`, and (ii) sibling types with a common supertype: `java.util.List`, `java.util.Set`, `java.util.Map`. Other popular TCPs involve types declared in `java.util`, `java.lang`, `java.time` and `java.io`. None of these TCPs are supported by the current tools.

We further analyse the popular TCPs and find that 40% of these involve `Internal` types, while the rest involve `External` types. This result is surprising, as we did not expect to find any TCPs involving `Internal` types, because they would get filtered out by the "at least two projects" predicate. On further investigation, we found that TCPs involving `Internal` types for a given project, affect dependent projects that need to adapt to the `External` type change. For example, the developers of *apache/hbase* changed the return type of 22 public methods from `HRegion` to `Region`. When the projects *apache/hadoop*, *phoenixframework/phoenix* bumped their *hbase* dependency to version `1.1.3`, they adapted the invocations of these methods by performing the same

[4]http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/A/popular.html

type change (`HRegion`→`Region`). Such scenarios occur when library developers introduce a breaking change and all the clients adapt to that change when they update their dependencies. The results highlight a blind spot in the current research that has primarily focused on library migration and update [3, 24, 31, 32, 69, 134] and ignored intra-project type changes.

To provide an overview of the reasons for performing these popular TCPs, we studied the related research literature and developer documentation. In Table 3.5, we report 10 common reasons for performing type changes, along with some representative type changes extracted from our corpus, for each reason.

RQ6 Conclusion: 2.27% of the most popular type change patterns shared across projects account for 43% of type change instances. None of the top-10 most popular type change patterns are automated by current tools.

3.4 Implications

We present actionable, empirically-justified implications for four audiences: (i) researchers, (ii) tool builders and IDE designers, (iii) language and library designers, and (iv) developers and educators.

3.4.1 Researchers

R1. Foundations for Software Evolution (RQ1, 2 & 6) We found that type changes are more frequently applied than renames, but they are less studied. Previous studies on program transformations focused on refactorings like renames, moves and extractions [99, 100, 126]. We also observe that 41.6% of the type changes can potentially introduce breaking changes. Moreover, we find empirical evidence showing that when a library developer introduces a breaking change by performing a type change, the clients adapt to it by performing similar type changes. Our dataset [75] contains fine-grained information, including links to the exact lines of code in GitHub commits, where developers performed type changes and adapted the references to the program elements whose type changed. Such detailed information can help researchers to better design longitudinal studies to understand software evolution [18, 19], to perform more accurate API updates [24, 31, 32, 52],

library migrations [3, 4, 69, 134], and automated program repairs [96, 122] that involve type changes.

R2. Naming Conventions (RQ3) In 54.85% of TCIs, the program element gets renamed too (e.g., `File file` → `Path filePath`). Our results also show that developers tend to rename elements more often when the source and target types have a *composition* or *sibling* relationship. Previous researchers [11] who studied renaming in depth, missed the opportunity to explore the impact of type changes on the renaming of program elements. The tools we developed can be used by researchers to further explore this relationship. Researchers [2, 15, 146] have developed techniques, which recommend an element's name based on its usage context. These techniques could be applied whenever developers perform type changes.

R3. Infer Type Mappings (RQ2 & 3) Our findings show that *cascade* type change is a frequently performed edit action, when developers adapt the references of variables whose type changed. This edit action applies a secondary type change, often involving different types than the primary type change. Moreover we observe that 41.6% type changes are applied on public elements, introducing binary and source code incompatibilities. Thus, in order to perform safe API updates or migrations, it is imperative for the current techniques to infer type mapping for performing the cascade type changes, in addition to inferring method mappings.

R4. Support Parameterized Types (RQ2) The performed type changes frequently update the argument of a parameterized type. Current techniques [74, 85, 138] can modify the Parameterized type container (`Vector<String>` → `List<String>`) or replace Parameterized types with Simple types (`Function<Integer, Integer>` → `IntUnaryOperator`). However, they cannot adapt the program correctly when the type argument changes (e.g., `Map<String,Integer>` → `Map<String, Long>`). Performing this type change correctly requires inter-procedural points-to and escape analysis, which is not supported by any of the current techniques.

R5. Generalize Techniques for Sibling Types (RQ1 & 5) The most popular type changes are performed between sibling types that share a common super type (e.g., `java.util.List` → `java.util.Set`). These types represent similar concepts with some differences in their properties. Previously, researchers have solved

specific instances of these type changes, such as `HashMap`→`ConcurrentHashMap` [36], `Vector` → `ArrayList` [138], replace `HashTable`→`HashMap` [85, 111]. However, these techniques hardcode the semantic differences between the sibling types. Our data highlights a need for more general techniques to encode the differences between the properties of the two types.

R6. Crowdsource Type Changes (RQ1 & 6) We found that type changes are highly repetitive, within individual commits (7.3 TCIs per commit) but also across multiple commits from distinct projects. This confirms the findings of others on the repetitiveness of code changes [101, 104, 108, 121, 122], and calls for new research on crowdsourcing type change mappings from previously applied type changes. Our dataset [75] could be used as a starting point.

3.4.2 Tool Builders and IDE Designers

T1. Automate Reference Adaptation (RQ1) Type change is a very commonly applied transformation. This highlights that IDEs should provide support for advanced composite refactorings, which perform a type change and adapt the code referring to the variable whose type changed. While current IDEs support refactorings like *Change Method Signature* [50] or type migration [67], they only update the declaration of the method or variables, but do not adapt the references. The *adaptation process* requires identifying (i) the method mappings between the types to update the method call sites, and (ii) replacement rules for cascading type changes (Listing 3.1).

However, better tool support for type changes is desperately needed. A survey of 420 developers [101] ranked type change as the most highly desired feature (among commonly applied transformations) for IDE automation. Moreover, Nishizono et al. [110] found that among other source code maintenance tasks, *Change Variable Type* requires the longest comprehension time.

T2. Support Selective Type Changes (RQ5) All existing tools that perform type changes [67, 74, 85, 138] follow a migration approach, where the type change patterns are exhaustively applied within a particular scope. However, we observed that in 61.71% of type changes, developers apply them *selectively* on an element,

based on the context of code. The existing techniques should give the user more fine-grained control (other than specifying the scope) on *where* a type change should be applied. For example, developers perform the type change `String`→`URL` [27] judiciously, rather than eradicating all usages of type `String` in the project.

T3. Support Internal Project Type Changes (RQ2) Developers most often perform custom type changes between types which are declared in the project itself (i.e., *Internal*). The most appropriate techniques for performing such custom transformations are through DSLs [13, 111], however researchers [17] found that text-based DSLs are awkward to use. More research is needed to make it easier to express custom type changes.

3.4.3 Language and Library Designers

L1. Understand Library Usage (All RQs) Language and library designers continuously evolve types. They enhance existing types, deprecate old types (e.g., `Vector`), introduce new types for new features (e.g., `java.util.Optional` for handling `null` values), or provide alternate types with more features (e.g., `java.util.Random`→`java.security.SecureRandom`). Our findings, the accompanying dataset [75], and the tools we developed, can help language and library designers to understand *what* types are most commonly used, misused, and underused, and *how* the clients adapt to new types. Thus, they can make informed and empirically-justified decisions to improve or introduce features.

L2. Adopt Value Types (RQ6) We found 3,747 type changes which box or unbox primitive types (e.g., `int` to `java.lang.Integer`). This practice is widespread in 101 projects from our corpus. The proposed *value types* feature in Project Valhalla (JEP 169 [123]) could eliminate these changes, by enabling developers to abstract over primitive types without the boxing overhead.

3.4.4 Software Developers and Educators

D1. Rich Educational Resources (RQ1, 2 & 6) Developers learn and educators teach new programming constructs through examples. Robillard and DeLine [119]

study what makes large APIs hard to learn and conclude that one of the important factors is the lack of usage examples. We provide 50,640 real-world examples of type changes in 129 large and mature Java projects. Because developers might need to inspect the entire commit, we provide link to the exact line of code in the GitHub commits (e.g., see [22]).

3.5 Summary

This chapter presents a fine-grained and large-scale empirical study to understand the type changes performed in 129 open source Java projects. To perform this study, we developed and validated tools to statically mine type changes and their subsequent code adaptations from the commit history of Java projects. We employed these tools to create an extensive and reliable data set of 297,543 type changes to answer six questions about the practice of type change. Some of our key surprising findings are:

1. Type changes are *more common* than renaming.
2. To adapt the code, developers often perform secondary *cascade* type changes, which are different than the primary type change.
3. Developers often rename elements, when changing their type.
4. Type changes between types having a *composition* relation need more adaptation effort than those with *inheritance* relation.
5. Developers more often perform type changes on `public` program elements rather than `private`, `package-private`, and `protected` elements, introducing potential breaking changes.
6. Although the raw number of type changes on `private` elements is less than the number of type changes on `public` elements, developers tend to change more often the types of `private` elements compared to `public` ones, indicating possible considerations for preserving backward compatibility.
7. Developers most often perform *selective* type changes, rather than *migrating* types in the entire project.

8. Type *migrations* are most commonly performed on *internal* project types.

The results presented in this study call for more intelligent tool support and further research to assist the developers by automating the task of type changes. We hope that this chapter motivates the community to advance the science and tooling for type change automation.

Chapter 4: Type Migration In Ultra-Large -scale codebases

The empirical study in Chapter 2 highlighted a common misuse of lambda expressions, where a generic functional interface is used over its efficient and specialized counterpart. Eliminating such misuses would require a scalable and configurable technique to automate the task of performing type changes in large open source projects. Chapter 3 explores the practice of performing type changes in the Java open source community, revealing that type changes are performed as frequently as the most common refactoring - *renaming*. It discusses various motivations for performing type changes, the technical challenges faced when performing type changes, and highlights opportunities for researchers and tool builders to improve the current automation.

This chapter describes T2R, a configurable framework for type migration (i.e. performing type changes on each element of source type in the program, exhaustively) in ultra-large-scale code bases. The first section motivates the problem and highlights the challenges for automating type migration . The second section thoroughly discusses the MapReduce amenable three-step process to collect, analyze and transform the program. The third section evaluates T2R for its *accuracy*, *scalability* and *usefulness* at eliminating misuses of functional interfaces, that were observed in Chapter 2.

4.1 Introduction

As programs evolve, an existing type `T` may need to be replaced by another type `R`, because `T` has been deprecated, or `R` is more efficient. For example, a programmer might need to replace usages of `HashMap` with `ArrayMap` to improve runtime performance [124]. Such a refactoring activity for going from `T` to `R` is known as *type migration*. Type migration modifies the declared types of variables or methods in a program and propagates the necessary changes throughout the program, preserving the type correctness.

Manually performing type migration can be quite tedious. First, programmers have to find all the instances of the type to be migrated. Second, they need to find all the dependencies in the source code. This is further complicated by type propagation

over assignment operations, parameters of the methods, overridden methods, and class hierarchies. Third, they need to make sure that every callsite has a counterpart in the new type. Finally, they must perform the transformation. These tasks can easily overwhelm developers. For example, when CoreNLP developers replaced generic with specialized Java 8's Functional Interfaces, the type migration involved 34 files containing 75 declarations, 74 allocation sites, 35 call sites, 3 subclasses and 39 lambda expressions. With the size of the projects the complexity of type migration also increases.

Type migration is a foundational step in class/library migration [133], fixing API-breaking changes in clients (e.g., updating a method's signature [30]), or correcting inefficient uses of an API [89]. These refactorings are generally hard to automate [23]. Existing automated approaches for type migration fall short when dealing with ultra-large codebases. Modern IDEs provide little support for type migration, often leaving out crucial analysis required for safe type migration, such as the mappings between the new and old types' methods. Moreover, state-of-the-art type migration techniques [13, 63, 77, 85, 111, 136, 137, 138] require in-depth whole program analysis. For example, researchers have proposed techniques for class library migration—a frequently-applied type migration—using *type constraints* analysis [13, 55, 137, 138]. However, type constraints analysis is resource intensive [138] and not scalable to ultra-large-scale codebases, since it has to extract constraints for *all* program types. While this was a great breakthrough for the program analysis community in the previous decade, it is less suitable for today's codebases, e.g., for large open source projects of hundreds of thousands LOC which we used in our formative study, or Google's codebase of 300M LOC. Another limitation of the current techniques is their extensive dependence on IDEs. These approaches perform resource-intensive whole-program analysis locally, inside of the IDE, and are unable to take advantage of the modern workflow of continuous integration on dedicated servers, thus hindering developers' productivity.

In this chapter, we propose a scalable and IDE-independent technique for type migration that integrates with most build systems (e.g., Ant, Maven, Gradle), and scales to ultra-large Java codebases. Our technique is composed of three consecutive steps, each amenable to MapReduce [28] parallel processing. To reduce the analysis space, in the first step, our approach passes over the entire codebase in search of language constructs (e.g., method signatures, method calls, variables) that match the *to-be-migrated* types. We then serialize each matched language construct to the filesystem. In the

```java
import java.util.function.Function;
interface LinearSearcher {
  double minimize(Function<Double, Double> f);
}

class GoldenSectionLineSearcher implements LinearSearcher {
  @Override double minimize(Function<Double, Double> f) {
    double val = getValue();
    return f.apply(val);
  }
}

class SVMLightFactory {
  LinearSearcher minimizer = new GoldenSectionLineSearcher()
  public double heldOutC() {
     Function<Double, Double> sq = x -> x*x;
     return minimizer.minimize(sq);
   }
}
```

(a) Before Type Migration

```java
import java.util.function.DoubleUnaryOperator;
interface LinearSearcher {
  double minimize(DoubleUnaryOperator f);
}

class GoldenSectionLineSearcher implements LinearSearcher {
  @Override double minimize(DoubleUnaryOperator f) {
    double val = getValue();
    return f.applyAsDouble(val);
  }
}

class SVMLightFactory {
  LinearSearcher minimizer = new GoldenSectionLineSearcher();
  public double heldOutC() {
     DoubleUnaryOperator sq = x -> x*x;
     return minimizer.minimize(sq);
   }
}
```

(b) After Type Migration

Figure 4.1: Motivating Example

second step, we construct a graph, representing the language constructs collected in the first phase as nodes and the relationships between them (e.g., a method declaration and its invocations) as edges. Using a set of migration-specific constraints, we analyze the graph to yield a list of refactorable candidates. Finally, in the third step, our technique passes again over the codebase in search of matches with the refactorable candidates, and applies the corresponding textual transformations in-place.

We implemented this approach in a tool called T2R (`T` $\rightarrow$ `R`). Our approach is generalizable to any type migration. However, to help the reader understand the complexities of type migration, in this chapter we use T2R to *specialize* the usage of Functional Interfaces across Java codebases, e.g., replacing `Function<Integer, Integer>` with `IntUnaryOperator`, or `BiFunction<U,V,Boolean>` with `BiPredicate<U,V>`. We were inspired by our previous work [89] where we studied 100,000 lambda expressions used in open source Java projects, and observed that 20% of the generic Functional Interfaces could be replaced with their specialized alternatives. Using generic Functional Interfaces causes *Autoboxing* and *Unboxing* between primitive and object types (e.g., `int` and `Integer`), which severely degrades performance [112]. Specializing Functional Interfaces effectively avoids the imposed overhead.

To evaluate our approach, we run T2R on Google's codebase with 300M lines of Java code. We also run T2R on seven performance-critical open-source projects. They are the best-in-class in domains such as databases, code quality analysis, and NLP, and are highly-optimized, totaling 2.6M LOC. T2R generated 130 patches in total, of which 126

compile and pass tests successfully. The original developers accepted 98% (114/126) of these patches.

This chapter makes the following contributions:

- A framework for type migration in ultra-large codebases, which employs a three-step process to collect, analyze, and transform types. Each step is amenable to MapReduce processing, thus making the approach scalable.
- A graph modelling the type structure of the program, facilitating analysis for safe type migration.
- An instantiation of the framework, T2R, which migrates the uses of generic Java 8 Functional Interface types to their specialized forms.
- An evaluation of the technique on seven open-source projects and Google's Java codebase, which shows our refactoring is scalable, safe, and useful.

4.2 Motivating Example

We show the intricacies associated with type migration through a real-world example. Figure 4.1 shows a simplified view of a T2R-generated patch that we sent to the open-source project CoreNLP [130]. The original patch comprises seven Java files, involving nine variables, three subclasses, and nine call sites, which we do not show because of space constraints.

`Function<X,Y>` is a generic Functional Interface introduced in Java 8, which accepts a value of type `X` and returns a value of type `Y`. Suppose that the developer considers migrating the type `Function<Double, Double>` (source) to its specialized counterpart, `DoubleUnaryOperator` (target) from `java.util.function`. Essentially, these two are semantically identical. The difference is that the single abstract method `apply()` declared in `Function<Double, Double>` accepts and returns values of the boxed `Double` type, while `DoubleUnaryOperator` deals directly with primitive `double`s. Unboxing is the automatic conversion by the Java compiler between the object wrapper classes and their corresponding primitive types (e.g., `Double` to `double`). Consequently, using `DoubleUnaryOperator` improves performance as it avoids unnecessary unboxing operations. A recent study [89] shows that developers often use the more expensive generic

Functional Interfaces instead of the specialized alternatives. This is perhaps due to the fact that there are 35 specialized functional interfaces available in Java and developers are not fully aware of their existence.

Going back to the example of Figure 4.1a, the interface `LinearSearcher` (line 2) declares the method `minimize()` with a parameter of type `Function<Double, Double>`. The class `GoldenSectionLineSearcher` implements `LinearSearcher` (line 5), and thus its `minimize()` method (line 6). The class `SVMLightFactory` (line 11) invokes `minimize()` on an instance of `GoldenSectionLineSearcher` (line 15).

Assume a developer starts refactoring the code by changing the parameter `Function<Double,Double> f` of `LinearSearcher.minimize()` (line 3) to the target type, `DoubleUnaryOperator`, as shown in Figure 4.1b. To make the code compile, the developer has to propagate this change to all the types which implement `LinearSearcher`, e.g., the type of parameter `f` in `GoldenLineSectionSearcher.minimize()` (line 6). The developer then has to migrate the type of the arguments of `minimize()` at all the callsites across the codebase.

Note that `minimize()` is invoked in `SVMLightFactory` (line 15) but also at other locations in the code (not shown in Figure 4.1). To migrate the invocation in `SVMLightFactory`, the developer must change the type of `sq` (line 14). In Java, a Functional Interface can be instantiated using a *lambda expression*, which is an anonymous function that can be created and used without belonging to any class. The initializer of `sq` is a lambda expression, in which its parameter of type `Double` is used for a multiplication. The developer would also need to check for potential uses of the methods invoked on the parameter within the lambda expression's body, if there existed any (e.g., `x -> x.doubleValue()`).

Figure 4.1b illustrates the migrated code. Note that, to apply this refactoring manually, the developer would need to perform an in-depth analysis on the entire source code to find all the callsites, and perform several nontrivial and time-consuming tasks, including: 1) checking the places wherein object references appear (the original patch in Figure 4.1 contains nine variables and their respective callsites and assignments), 2) checking the inheritance hierarchy in the migrated types (in the original patch, two methods are hierarchically related), 3) checking the subtyping relations (the original patch contains three subclasses of `Function<Double,Double>`), 4) computing the transitive closure of the change and ensuring it does not reach into external libraries (those cannot be changed),

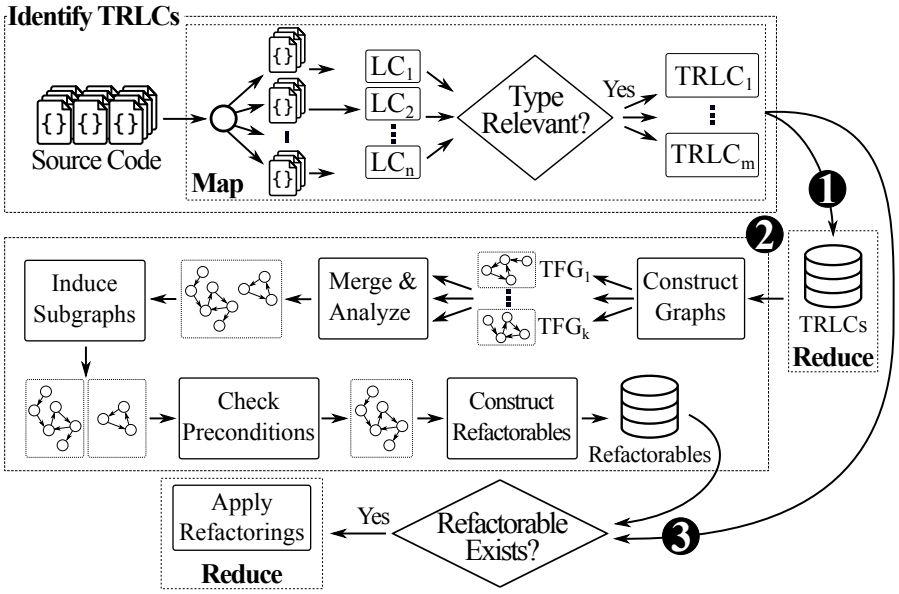


Figure 4.2: Approach Overview

and 5) verifying that the code does not invoke methods from type `T` that are absent in `R` (e.g., `Function.andThen()` has no corresponding method in `DoubleFunction`). Part of the required steps for a safe type migration might be facilitated using the navigational support of the IDE (e.g., finding usages of variables or subclasses), or by relying on the compile errors raised after each modification, but this technique does not scale as previously discussed. Next, we discuss our proposed approach to scalable type migration in large codebases.

4.3 Approach

Type migration impacts variable types and method signatures. Reasoning about how to propagate type changes safely across the whole codebase requires a whole-program type-dependency analysis. However, this is challenging to make scalable. One solution is to make the analysis distributed; but existing type migration techniques cannot adapt to distributed processing frameworks as they access single source code files one at a time in an arbitrary order, during which no global state is maintained. To overcome this, we designed a three-step algorithm (Figure 4.2) that takes as input

the source code and the TRANSFORMATION SPECIFICATIONS between types `T` and `R`. The TRANSFORMATION SPECIFICATIONS define the source and target types (`T` → `R`) and their corresponding equivalent methods (`Function<Double,Double>.apply()` → `DoubleUnaryOperator.applyAsDouble()`).

The three phases include (i) collecting relevant type and syntactic information from each compilation unit, (ii) analyzing the collected information and merging it across compilation units to get a whole program view, and (iii) transforming the code. Dividing the type migration technique into three separate phases enables scalability in large codebases where the entire code spans hundreds of millions of LOC. Each of the steps becomes amenable to distributed execution internally through, for example, MAPREDUCE [28]. The output of each step is fed as input to the next step.

At a high level, first the **collection** phase identifies all the type-related source code constructs by traversing over each compilation unit in a distributed manner and stores them in a language construct representation. Second, the **analysis** phase transforms the language constructs into graphs and merges them to identify sites where the type migration propagates across the whole codebase. Third, the **refactoring** phase applies the code changes by passing over the entire codebase in a distributed fashion.

In the following subsections, we describe the three phases.

4.3.1 Collecting Type-Relevant Language Constructs

We reduce the problem of analyzing *all* the types in the entire codebase to the search for language constructs that match with the migration source type `T` (e.g., `java.util.function.Function` for the example in Figure 4.1). This reduction allows us to scale to large codebases.

As shown in Figure 4.2, the input to the first phase is the source code and the type `T` to be migrated. The goal of this phase is to extract only relevant information from the source code that is necessary for the type migration analysis at hand. The output of this step is a collection of Type-Relevant Language Constructs (TRLCs).

Definition 1 (Language Construct)**.** A Language Construct (LC) is a syntactic part of the program formed by one or more lexical tokens in accordance to the rules defined by the programming language.

Examples of language constructs in Java include method declarations, method invocations, or variables. Table 4.1 presents the language constructs our work targets.

Definition 2 (Type-Relevant Language Construct)**.** The Type-Relevant Language Construct (TRLC) of a LC of type τ captures the IDENTIFICATION of LC as well as the IDENTIFICATION of all type-dependent expressions of LC.

Definition 3 (IDENTIFICATION)**.** The IDENTIFICATION of a language construct LC is a 4-tuple $(name, kind, type, owner)$, where:

- *name* is the name of the AST node corresponding to LC. For example, for the method declaration `foo()`, name is "foo". For anonymous constructs (e.g., lambda expressions, anonymous classes), the name is `NULL`,
- *kind* is the kind of the AST node of LC (e.g., `MD` for method declarations and constructors, and `VAR` for local variables, fields and parameters),
- *type* is the explicit declared type of LC,
- *owner* is the IDENTIFICATION of LC's enclosing language construct (e.g., in Figure 4.2, the owner of `Function<Double,Double> f` is the IDENTIFICATION of `minimize()`, whose owner is the IDENTIFICATION of `GoldenSectionLineSearcher`).

T2R visits the LCs of each compilation unit in the codebase in parallel—possibly in an arbitrary order, as imposed by the runtime distributed infrastructure. It parses the source code into Abstract Syntax Trees (ASTs), and collects binding information for types and symbols. To infer type and hierarchy information, the code needs to be compiled. To compile the code in a scalable manner, our approach focuses on the compilation unit being visited, and retrieves the compilation unit and its dependencies from the database of an indexer. We schedule the indexer to traverse the entire codebase periodically (e.g., every night) in a MAPREDUCE style to populate a database that contains all compilation units and their inputs.

For each visited LC, our approach also analyzes its type-dependent expressions (see Table 4.1, third column). If a type-dependent expression is a (sub)type of `T`, our approach constructs a TRLC for that LC using the available type and syntactic information from the AST node being visited. Each TRLC captures IDENTIFICATION

Table 4.1: LCs and TRLCs

Language Constructs (LCs)	**Kind Label of LC**	**Type-dependent Expressions**	**TRLC Example**
Method Declaration Constructor	`MD`	Parameters, `return` statement	`double minimize(Function<Double,Double) f`
Method Invocation Method Reference Class Instantiation	`MI`	Receiver, Arguments	`f.apply(val)`, `minimizer.minimize(sq)`
Method Parameter Local/Instance Variable	`VAR`	Type Declaration, Initializer	`double minimize(Function<Double,Double) f`
Assignment	`ASGN`	LHS and RHS of an assignment	`sq = x -> x*x`
Lambda Expression	`LMBD`	Lambda Expression	`x -> x*x`
Explicit/Anonymous Class Declaration	`CLS`	`implements` clause Overridden Method Declaration	

objects depending on the kind of LCs; e.g., for the variable declaration statement `Function<Integer,Integer> sq = x -> x*x`, the TRLC captures two IDENTIFICATIONs, namely for the variable `sq` and the lambda expression `x -> x*x`.

Figure 4.3 shows a sample IDENTIFICATION constructed for the lambda expression `x -> x*x` declared in `SVMLightFactory.heldOutC()` (Figure 4.1 line 14). Observe that the enclosing assignment expression has made `sq` (i.e., the initialized variable) the *owner* for the lambda expression. The *owner* hierarchy continues from `sq` to the `heldOutC()` method, the `SVMLightFactory` class, and eventually the package in which `SVMLightFactory` is defined. This representation essentially allows uniquely identifying all language constructs throughout the codebase that might be affected by the type migration.

Furthermore, whenever our approach finds relevant inheritance hierarchy information, it constructs IDENTIFICATIONs for `super` methods when the class wherein `super` is referenced is of type (or extends) `T`, and also for subclasses of `T`. Since our approach constructs these IDENTIFICATIONs based on local syntactical information (i.e., another compilation unit contains the sub/superclass's declaration), we call them *inferred* IDENTIFICATIONs. Our approach adds these inferred IDENTIFICATIONs to the corresponding TRLCs as well.

Our collection step emits these TRLC instances as it passes over the codebase in

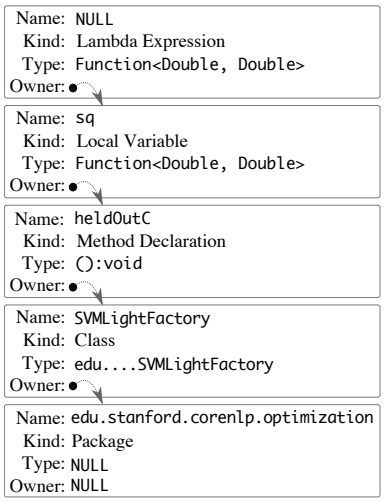


Figure 4.3: IDENTIFICATION for `x -> x*x` in `Function<Double,Double> sq = x -> x*x`

parallel, and then subsequently serializes them to the filesystem for further processing in the next step.

4.3.2 Detecting Refactorable Language Constructs

The result of the previous phase of our approach is a set of TRLCs, collected using local information about their type-dependent LCs and in an arbitrary order of the analyzed compilation units. For a safe type migration, we would need to consider the relationships among all TRLCs collected for the entire source code. For example, the TRLC constructed for the method declaration `GoldenSectionLineSearcher.minimize()` (Figure 4.1a, line 6) has information that it is dependent on the type of its parameter `Function<Double,Double> f`, but does not have the information of the arguments passed to its invocations. To safely migrate its parameter's type, we need to propagate the changes to the arguments in the method's callsites too. This normally requires using an inter-procedural analysis, which is infeasible in ultra-large-scale codebases.

The goal of the second phase of our approach is to establish and analyze such relationships between the LCs to achieve the effect of an inter-procedural analysis, in order to decide which LC should (and can) be refactored safely in the codebase. The input to

this phase is the set of collected TRLCs, as well as the TRANSFORMATION SPECIFICATIONS. The output is a set of refactoring instructions for modifying language constructs of type `T` that can be safely migrated to the new type `R`.

4.3.2.1 Graph Analysis

The core of our analysis phase revolves around the notion of a graph representation of the LCs, i.e., the Type-Fact Graph (TFG). A TFG captures the type-dependency relationship between different LCs of an entire program. TFG is inspired by the formalization of refactorings with graph transformations [95], and type constraints [138].

Definition 4 (Type-Fact Graph)**.** A Type-Fact Graph (TFG) is a directed, labeled graph $G = (V, E)$, where:

- V is the set of nodes in TFG; each node represents an IDENTIFICATION.
- $E \subseteq V \times L \times V$ is a set of directed edges in TFG, where L is a finite set of edge labels. The edges correspond to the semantic relationships between two nodes in TFG, and the labels determine the type of the relationship.

Our approach establishes a directed, labeled edge between two TFG nodes v_1 and v_2 when there is a type relationship between the two source code elements that v_1 and v_2 represent. These relationships are similar to type constraints [13]. The edge label $l \in L$ denotes the type of the relationship. For example, there is a type relationship between a method declaration in an interface and its overriding method declaration in an implementing class, because changing the overridden method's signature should be propagated to the overriding methods. In this example, we establish a directed edge from the TFG node that corresponds to the overriding method, which has the type label `MD`, to the TFG node that corresponds to the overridden method with the same type label, and we label the edge as `AFFECTED_BY_HIERARCHY`. Table 4.2 illustrates a list of such relationships and the corresponding edge labels.

4.3.2.2 Constructing the Graph

In this step, we form a TFG of all the TRLCs. This unified TFG captures all the information needed for inferring the final refactoring sites for a safe type migration. The

Table 4.2: TFG Edge Labels

Node type / Edge Directions	Edge Label	Description
MD→MD	AFFECTED_BY_HIERARCHY	Relation due to hierarchy
MD→VAR, VAR→MD/CLS/LMBD	PARAMETER, OWNER	Relation between method and its parameters
MI→VAR/MI/CLS/LMBD, VAR/MI/CLS/LMBD→MI	ARGUMENT, ARGUMENT_OF	Relation between method invocation and its argument
VAR→VAR/MI/CLS/LMBD, VAR/MI/CLS/LMBD→VAR	ASSIGNED_AS, ASSIGNED_TO	Relation owing to assignment
VAR/MI→MI, MI→VAR/MI	METHOD_INVOKED, REFERENCE	Relation between method invocation and its receiver
MD→VAR/MI/CLS/LMBD, VAR/MI/CLS/LMBD→MD	RETURNS, RETURNED_BY	Relation between method declaration and its return expression
CLS→MD	OVERRIDES	Relation between class and overridden method
MI→MD, MD→MI	DECLARATION, INVOCATION	Relation between method invocation and its declaration
VAR→CLS	OF_TYPE	Relation between variable when its type is subtype of the source type

key steps in this phase are depicted in Figure 4.2:

1. Constructing individual TFGs from TRLCs,
2. Merging individual TFGs into a unified, enriched TFG.

Constructing individual TFGs from TRLCs. First, our approach deserializes the physical representation of each TRLC and transforms it into an individual TFG. As an example, Figure 4.4 shows the TFGs constructed for two TRLCs in the class `SVMLightFactory` of the motivating example.

The grey boxes depict the items that are in the graph; we also show the owners for these TFG nodes for clarity. Also note that there are repetitive IDENTIFICATIONs across the two TFGs. This is due to the fact that the first phase of our approach does not have a holistic view of the entire codebase, and only has local information, as it visits a single compilation unit at a time in a distributed MAPREDUCE process.

Constructing the enriched TFG. Our approach then incrementally merges the individual TFGs among themselves. The *Merge & Analyze* block depicted in Figure 4.2 takes as input two TFGs and merges them by taking the union of their nodes (i.e., IDENTIFICATIONs) and edges. Subsequently, our approach performs two analysis and enrichment operations upon this merged graph, to 1) find method declarations for method invocations, and 2) replacing nodes associated with *inferred* IDENTIFICATIONs with non-

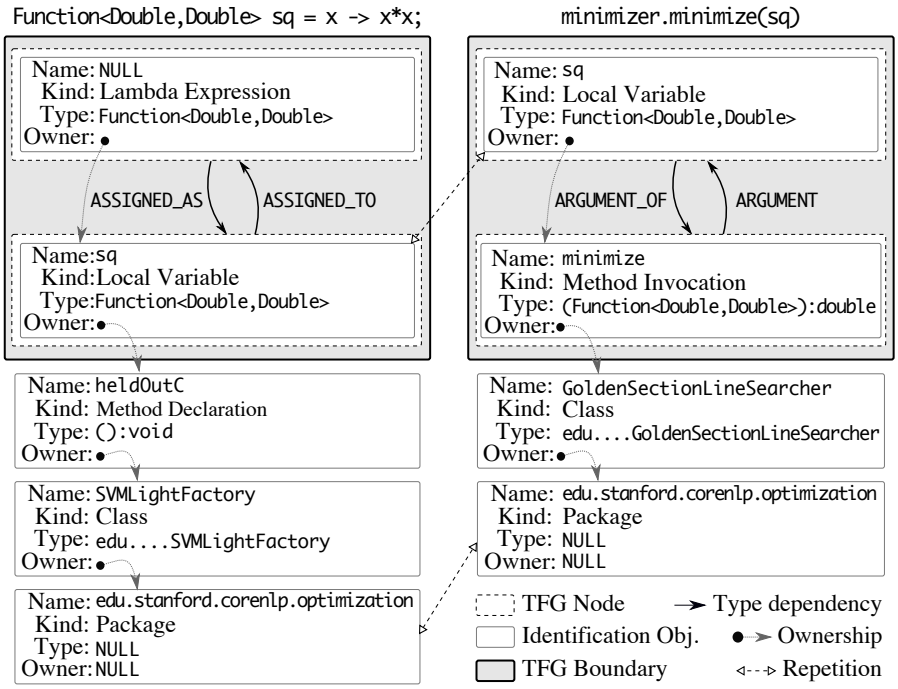


Figure 4.4: Individual TFGs for `SVMLightFactory`

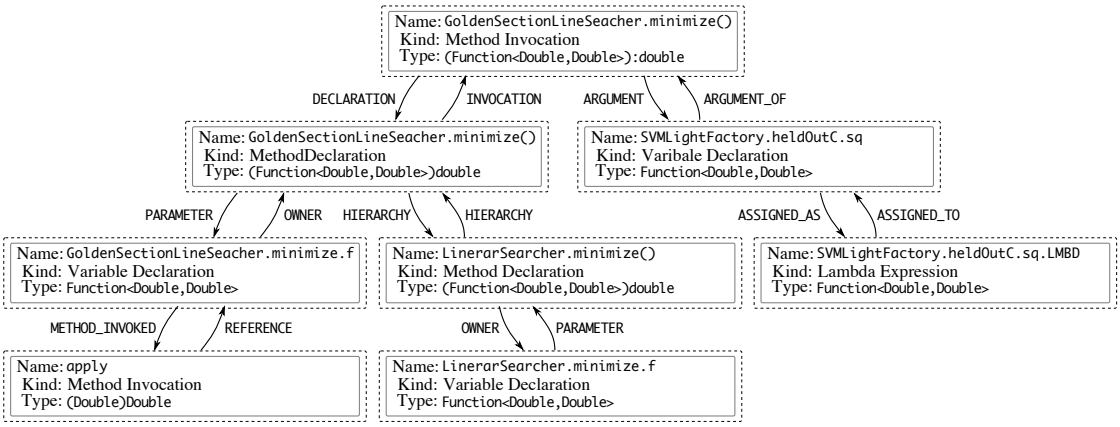


Figure 4.5: Unified TFG

inferred ones. Figure 4.5 illustrates the example of applying this operation on the two independent TFGs shown in Figure 4.4.

Finding declarations of method invocations. For each node representing a method invocation, our approach searches the entire TFG to find a node representing its declaration. The approach adds new edges between the two nodes. The edges between the nodes corresponding to the method declaration and method invocation for `GoldenSectionLineSearcher.minimize()` in Figure 4.5 illustrate the result of this analysis and enrichment. Observe that the IDENTIFICATIONs of the declaration and invocation nodes only differ by their *kind*. In practice, our approach uses this property of IDENTIFICATIONs to find declarations corresponding to method invocations.

Replacing inferred identification nodes in the TFG with non-inferred ones. Recall that, in the first phase, our algorithm adds *inferred* IDENTIFICATIONs for LCs that are not syntactically present in a compilation unit, but they must exist somewhere else for the source code to be compilable. For example, when collecting the type-relevant language constructs, the algorithm could infer that there must be a declaration for a specific method in a supertype, using the available semantic information. This is because the collection phase only has a local view of TRLCs, thus, the corresponding TFG node for this method declaration might be missing. In the analysis step, however, the algorithm has a holistic view of all the TRLCs; thus, the algorithm in this step attempts to discover these missing nodes. If the search is successful, the algorithm replaces the inferred nodes by the identified nodes, else the nodes remain inferred and

Table 4.3: TRANSFORMATION SPECIFICATIONS from `Function<Double,Double>` to `DoubleUnaryOperator`

LC kind	Predicate over LC expressions	Refactoring Instruction
VAR	Type Declaration = `Function<Double,Double>`	`Change Type To DoubleUnaryOperator`
MD	Return Type = `Function<Double,Double>`	`Change Type To DoubleUnaryOperator`

CLS	Type $\preceq$ `Function<Double,Double>`†	`Change Type Super To DoubleUnaryOperator`
		`Change Name OveridnMthd To applyAsDouble`
		`Change Type OveridenMthd Param 0 To double`
		`Change Type OveridenMthd Return To double`

LMBD	Type $\preceq$ `Function<Double,Double>`	`Change Type To DoubleUnaryOperator`
MI	Receiver.type $\preceq$ `Function<Double,Double>` $\wedge$ name = `"apply"`	`Change Name To applyAsDouble` `Change Type Receiver To DoubleUnaryOperator`

† `A`$\preceq$`B` means A is (sub)type of B.

graphs containing these nodes are filtered out by the preconditions.

In Figure 4.5, observe also that the algorithm has *merged* the variable declaration `sq` in `SVMLightFactory` with the argument `sq` passed to the method `minimize`. Since our approach preserves the edges in this operation, it has not omitted the initializer of the variable `sq`.

4.3.3 Generating Refactorables

After the algorithm merges all the graphs into a single TFG, it identifies its disconnected subgraphs. Each disconnected subgraph signifies a set of language constructs that have type dependencies among themselves in the program. Figure 4.5 shows one such disconnected subgraph obtained from the motivating example (Figure 4.1). This graph essentially depicts all the sites which are type dependent on the signature of `minimize()` in `GoldenSectionLineSearcher`. T2R first filters these TFGs through a set of preconditions; and then reduces each filtered TFG into a set of REFACTORABLEs which captures each node of the TFG (i.e. IDENTIFICATION) and a corresponding REFACTORING INSTRUCTION for the language construct it represents.

4.3.3.1 Refactoring Preconditions

In order for type migration transformations to be type correct (i.e., the resulting code is compilable), we define preconditions that need to be satisfied for each TFG subgraph. These preconditions are as follows:

Precondition 1: The TRANSFORMATION SPECIFICATIONS should include refactoring instructions for each type of node in each individual TFG subgraph. For example, when the method `andThen()` is invoked on a variable `v` of type `Function<Double,String>`, the type of `v` cannot be migrated to `DoubleFunction<String>`, because `DoubleFunction<String>` does not have a corresponding method for `andThen()` (i.e., no refactoring instruction can exist for this migration). This precondition guarantees that all the methods of the source type `T` will have a corresponding method in `R`. When this precondition is violated, there will be methods in `T` that are called on references to `R` after refactoring, which results into compiler errors.

Precondition 2: In each individual TFG subgraph, there must exist no node where the kind of the node's associated IDENTIFICATION has remained *inferred* at the end of the analysis phase. This happens, for example, to language constructs for which the declaration is in an external library. In our motivating example (Figure 4.1), if `sq` is passed as an argument to a method of a third party library, we would not refactor it. Violating this precondition essentially means changing types in parts of the source code while not being able to change other relevant places, e.g., an external dependency, which leads to breaking the type correctness of the program.

Precondition 3: The type migration refactoringshould not affect the elements declared using generic types with type variables. This precondition can, for example, prevent incorrect type changes applied on generic types appearing in class hierarchies. As an example, consider the interface `I<U,V>` declaring the method `void m(Function<U,V> p)`. A class `C1` which implements `I<Integer,Integer>` must declare a method `void m(Function<Integer,Integer> p)`. Changing the type of the parameter `p` in `C1.m()` to `IntUnaryOperator` warrants changing the type of the same parameter in `I.m()`. However, there might be other classes implementing `I` with different explicit type parameters (e.g., `C2` which implements `I<Boolean, Integer>`); therefore, we are not allowed to apply this change in the first place or else it will break the type correctness of the program.

Precondition 4: (Optional) The TFG representation is flexible to express additional constraints for the type migration. For instance, a user can define a custom precondition for not propagating changes across public methods or packages.

If any of the preconditions are not satisfied by a TFG subgraph, that particular subgraph becomes unsuitable for refactoring. We only migrate the elements represented by subgraphs which satisfy *all* our preconditions.

4.3.3.2 Generating Refactorables

For the TFG subgraphs that satisfy the preconditions, we produce a Refactorable item for each node in the filtered TFGs based on the Transformation Specifications provided by the user. A Refactorable encapsulates an Identification and the corresponding Refactoring Instruction (illustrated in Table 4.3) for the language construct that the Identification represents. These Refactoring Instructions are provided by the user in the Transformation Specifications. The Refactoring Instruction expresses the refactoring action to be performed, the parameters of this action, and the sub-constructs of the language construct on which the action should be applied. For example, in Table 4.3, Refactoring Instruction for `CLS` (i.e., an explicit/anonymous class declaration) expresses three actions: 1) the super type (declared using the `implements`/`extends` clause) has to be changed from `Function<Double,Double>` to `DoubleUnaryOperator`, 2) the name of the overridden method `apply()` has to be changed to `applyAsDouble`, and 3) the return type and the zeroth parameter of `apply()` has to be changed from `Double` to `double`. The expressiveness of Refactoring Instruction helps our approach to handle various language constructs.

Our approach then serializes these refactorables to the filesystem to use them in the final step of our migration.

4.3.4 Applying Refactorings

The final phase is responsible for applying the detected refactorings to the source code. As illustrated in Figure 4.2, similar to the collect phase, we visit AST nodes of the source code in search of matching TRLCs in a MapReduce parallel process. We then check each TRLC against the set of refactorables that were generated at the end of the

previous phase. If we find a match, we use the refactoring instruction of the refactorable upon the visited language construct by performing an in-place AST node rewriting. For example, the algorithm translates the instruction `ChangeType:DoubleUnaryOperator` for parameter `f` of `minimize` in `LinearSearch`, to a rewriting action, which replaces the type of the variable declaration with `DoubleUnaryOperator`. We also import the new types into the enclosing class, if they are not already present.

4.4 Evaluation

To evaluate the efficacy and real-world relevance of our approach, we address the following research questions:

RQ1 (Accuracy): How accurate are the type migrations performed by our approach?

RQ2 (Usefulness): Do developers find our type migrations useful in practice?

RQ3 (Scalability): How scalable is our type migration?

4.4.1 Software Corpora

We evaluate T2R using two software corpora: (1) seven open-source projects, and (2) the proprietary codebase of Google. Table 4.4 presents our corpora in terms of their source code size.

4.4.1.1 Open-source Corpus

We use seven open-source, performance-critical projects which are mature and popular (thousands of stars on GitHub) and are widely used in the industry, and extensively use Functional Interfaces.

These projects are: Cassandra, Presto, Neo4j (best-in class, scalable and highly-performant databases [7, 102, 114]), SonarQube [128], CoreNLP [130], java-design-patterns [65], and Speedment [129]

Finding any missed opportunity to further specialize functional interfaces in such projects, is an important contribution.

For this corpus, we provide T2R with Transformation Specifications to migrate seven generic Functional Interfaces to 35 specialized alternatives. We use this corpus to assess the accuracy and usefulness of our approach (i.e., RQ1 and RQ2). Although we designed T2R to scale on ultra-large codebases using MapReduce on dedicated hardware, we show that even open-source developers that use commodity hardware (e.g., a laptop) can still use T2R effectively.

4.4.1.2 Proprietary Corpus

Our proprietary corpus contains over 300M lines of Java code at Google. For this corpus, we provide T2R with Transformation Specifications for migrating the type `java.util.function.Function` to 13 specializations. These were requested by Google as a proof of concept for implementing an ultra-large scale type migration. In addition to the first two research questions, we use this corpus to assess the scalability of our approach (i.e., RQ3); thus, our evaluation employs MapReduce parallel computing in the first and third steps of the migration process.

4.4.2 Methodology

Since T2R operates as a greedy algorithm to migrate as many usages of type `T` to `R` in the codebase, we compute accuracy (RQ1) using standard metrics from information retrieval. To measure the accuracy, we apply the patches generated by T2R to all the open-source projects in the corpora.

We compute *precision* as the fraction of generated patches that passed successfully (no compilation or test failures), and thus were correct refactorings. To compute *recall*, we first need to determine the maximal set of refactorings in the corpus, and then compute how many of these T2R found. From the initial candidates, i.e., all the programming constructs that are defined of type `T`, we manually computed how many candidates are suitable for refactoring (i.e., they pass preconditions). We then report how many of these T2R found, and the reason why it rejected the remaining candidates.

To evaluate the *usefulness* of the type migrations in practice (i.e., RQ2), we submit the patches produced by T2R to the original developers and report the number of accepted patches. We also measure and report the size of the submitted patches.

Table 4.4: Results

			Accuracy		Usefulness				Scalability
Corpus		Candidate TRLCs	Patches Generated	Patches Passed	Patches Accepted	Files Changed	Lines Added	Lines Removed	SLOC
Proprietary	Google	161,255	59	57	56	70	289	265	300M
OSS	Neo4j	609	17	16	16	46	243	214	787K
	CoreNLP	412	18	18	18	34	157	144	576K
	SonarQube	82	2	2	2	6	12	11	551K
	Presto	564	11	11	0 (10)†	36	91	87	543K
	Cassandra	478	16	15	15	23	82	71	355K
	Speedment	251	5	5	5	14	89	84	127K
	java-design-patterns	14	2	2	2	2	8	4	27K
Total		**163,665**	**130**	**126**	**114**	**231**	**973**	**880**	**302.6M**

† The value shown in parentheses is the number of patches which are currently under review by the developers.

To evaluate the *scalability* of T2R (i.e., RQ3), we measure the number of lines of code T2R can handle.

4.5 Results

Table 4.4 summarizes the results we obtained by running T2R over the open-source and proprietary corpora.

4.5.1 Accuracy

Table 4.4 reports the number of initial TRLC candidates for the refactoring (which the developer would need to investigate and possibly modify for a safe type migration), how many subgraphs satisfy all the preconditions and T2R generated patches for, and how many of the applied patches preserve the syntax and semantics of the program.

T2R generated 130 patches, out of which 126 compiled and passed the tests successfully with an overall precision of 97%.

For the open-source corpus, T2R generated 71 patches, out of which one introduced a compilation error and one failed a single test case from the thousands of tests available for each project. The compilation error occurred in Cassandra due to T2R's implementation limitation in handling static method calls as method references. Also, a test case in the integration test suite of Neo4j failed since the applied changes affected a public Java method called by Scala.

For the proprietary corpus, T2R generated 59 patches, out of which two introduced

integration test case failures, due to the usage of Java reflection and dependency injection in the code, which T2R does not handle currently. This is a well-known limitation [135] of all practical refactoring tools.

The recall rate of T2R is 100% for the open-source projects. Considering the size of the codebases used in our evaluation, the number of the generated patches might seem small. Note that we are using mature projects where performance is at the forefront. For example, CASSANDRA is a highly-optimized database used predominantly in industry, and its developers had already manually refactored code to use specialized Functional Interfaces. Nevertheless, T2R was still able to find 15 patches that affected its core modules.

A lead developer of CASSANDRA mentioned that one of these patches in particular *"was deep inside the database and actually is very important for performance".*

Moreover, not all candidate TFG subgraphs result in applicable patches. In our OSS corpus, 70% of these subgraphs do not pass precondition #2: we cannot proliferate changes to external libraries. Particularly, Functional Interfaces are usually used with Java's `Stream` or `Optional` APIs and the refactoring would need to change these APIs. Similarly, we noticed that custom Functional Interfaces provided by external libraries (e.g., GUAVA Collections) are extensively used in the proprietary corpus. The open-source experiments also reveal that 28% of TFG candidate subgraphs do not pass precondition #3, and 2% of patches failed preconditions #1 and #4.

4.5.2 Usefulness

We sent the 126 passing patches to the original developers as pull requests. At the time of writing this paper, 10 patches for the OSS corpus are still under review. Of the remaining patches, 114 were accepted, with an acceptance rate of 98%.

Table 4.4 shows the number of patches accepted by the developers as well as details about the impact of the changes, i.e., the number of Java files affected and lines added/deleted.

In total, we sent 69 patches affecting 161 files to the developers of the open-source corpus, out of which 58 were accepted, and 10 are currently under review. One patch was rejected since it affected method signatures of the project's public APIs, which the developers did not want to change. We could define a precondition to prevent T2R

from touching public APIs, if we had known this constraint a priori. On average, T2R produced patches affecting 2.03 files, adding 9.17 lines and deleting 8.12 lines per patch. Our companion website [71] contains these patches and the developers' responses.

For the proprietary corpus, we sent 57 patches affecting 70 files to the Google developers, out of which 56 were accepted. One patch was rejected since the developer found `Predicate<String>` to be less readable than `Function<String, Boolean>` in a test case. On average, T2R produced patches affecting 1.19 files per patch, adding 4.88 lines and deleting 4.64 lines per patch.

4.5.3 Scalability

We focus on the proprietary corpus for evaluating the scalability due to its sheer size. In the first phase, T2R ran over 300M lines of Java code and collected 161,255 type-relevant language constructs (TRLCs) in around 15 minutes. T2R then analyzed the collected TRLCs in the second phase to detect 292 refactorables in around three minutes. Finally, T2R ran again over the entire codebase to find matches with the refactorables and generate patches, in around 15 minutes.

T2R executed these three phases on Google's cloud infrastructure. The first and third phases used MapReduce parallel processing. In total, the whole type migration process took 33 minutes for 300M LOC, highlighting the scalability of T2R.

4.6 Discussion

4.6.1 Threats to Validity

External Validity. Do our results generalize? We chose a diverse set of software corpora, including multiple highly-rated open-source systems used by other researchers [89], in addition to an ultra-large-scale Google codebase consisting of 300M lines of code in total. The corpora covers a wide range of software domains, sizes, and development practices.

Does T2R generalize to other type migrations? In this paper we illustrated T2R's features by using a complex refactoring, Specializing Functional Interfaces (see Section 4.2). Since T2R handles a large variety of constructs, we believe T2R can be

extended to other type migrations.

Internal Validity. Does T2R produce valid results? Our evaluation shows that T2R can handle real-world migration of Functional Interfaces safely. We also wrote extensive test suites to ensure covering a wide range of language constructs.

Reliability. Can others replicate our results? We have made the open-source version of T2R and the results available [71].

4.7 Summary

Type migration is crucial to ensure a codebase evolves and does not incur technical debt. As the size of a codebase increases, both the need for and the complexity of such migrations increase. In this chapter, we present an automated technique for scalable type migration in large codebases. We implemented the approach in a framework called T2R, and we use it to specialize Functional Interfaces in Java. Our results show that the type migrations performed by T2R on seven open-source and Google's codebases is scalable, safe, and useful. Even in highly optimized projects, T2R found 114 opportunities for improving performance.

Our work shows that by using a graph modelling the type dependencies in the source code, we can mimic an inter-procedural program analysis, which is amenable to the MapReduce parallel and distributed computing, and therefore, is scalable to ultra-large-scale code bases.

In the future, we aim to make T2R more applicable to more instances of type migration by developing a DSL which can express mappings between `T` and `R`. We also plan to implement the graph analysis phase of T2R as MapReduce to further enhance its scalability.

Chapter 5: Inferring Type Changes

Chapter 2 and Chapter 3 describe two empirical studies that highlight the applicability of type change techniques in the real world, reveal their shortcomings and uncover new opportunities for improving them. Chapter 4 describes T2R, a type change(migration) technique for performing type changes in large to ultra-large scale code bases, based on specifications provided by the user. At the heart of this technique is a graph modelling that captures the type dependencies between program elements across the code base. We implement this technique in an IDE-independent tool, and evaluate its effectiveness at eliminating misuses of functional interfaces from large high quality open source projects and Google's Java codebase containing 300MLOC.

T2R and other type change techniques require the user to manually encode the syntactic transformations required to perform the desired type changes, introducing a barrier in the adoption of these techniques. To overcome this challenge, in this chapter we introduce TC-Infer that infers the adaptations required to perform a type change as rewrite rules in Comby language. These rewrite rules can be provided as specifications to T2R and other type change techniques. The first part of the chapter walks the reader through the nuances associated to inferring such rewrite rule, the next part describes the TC-Infer and the final section evaluates the *applicability*, *effectiveness* and *usefulness* of the rules produced by TC-Infer.

5.1 Introduction

As programs evolve, the types of program elements are changed for several reasons, such as improving *performance* [40, 44, 45] (e.g., `String`→`StringBuilder`), *maintainability* [27] (e.g., `String`→`Path`), introducing *concurrency* [36] (e.g., `HashMap`→`ConcurrentHashMap`), handling *deprecation* or performing *library migration* [4, 69, 134] (e.g., `org.apache.commons.logging.Log`→`org.slf4j.Logger`). Such a program transformation where the type of a program element (i.e., variable, field, or method) is updated, and then the type constraints of the new type are propagated to

the code base by adapting the code referring to this element, is called a *type change*.

Despite that developers perform type changes more frequently [76] than popular refactorings such as *rename*, the tool support for type changes is negligible compared to refactoring automation. Developers predominantly perform type changes by hand [76]. This can be tedious, error-prone and it can easily overwhelm the developers. Researchers [13, 60, 74, 84, 94, 131, 147, 149] and tool builders [9, 50, 67, 97] have proposed techniques that assist developers in performing these type changes.

The Achilles heel of these techniques is that the user has to manually encode the syntactic transformations required to perform the desired type changes. While these techniques allow the transformations to be expressed as rewrite rules over templates of Java expressions, they are still manual and labour intensive because it requires the developers to encode the transformations. When the developers are unfamiliar with the types, they would have to ask a co-developer or look up the documentation (which could be outdated or unavailable), release notes, or Q&A forums to understand how to correctly adapt the code to perform the type change. Even when the developers are familiar with the types involved in the type change, using such program transformation systems is not straightforward (their learning is measured in weeks or months [17, 79]). This introduces a barrier to the adoption of these techniques.

Given that many software evolution tasks are repetitive by nature[61, 105, 107], our key insight is that developers from multiple open-source projects apply similar type changes in their projects. In a corpus of 400,000 type changes performed in 130 open source projects, researchers [76] observed that 68% of them were performed in more than one commit. If we could harness this rich resource of type change examples, we could infer the adaptations and reduce the burden on the developers. This will improve the applicability and utility of the current type change techniques.

In this chapter we introduce a technique, TC-Infer, that learns the task of performing type changes by analyzing several examples of how other open source developers have performed the same type change previously. First, TC-Infer mines the commit history of projects and identifies the type changes and other refactorings performed. Then TC-Infer analyzes them to deduce rewrite rules that capture the required adaptations to perform the type change. The rules produced by our technique can be readily used by existing state-of-the-practice type migration tools like IntelliJ Platform's Type Migra-

tion[1], or state-of-the-art tools that use type constraints [13] or type-fact graphs [74]. We leverage two state-of-the-art techniques : (i) RefactoringMiner [76, 142] to identify refactorings and (ii) Comby [145] to represent and perform light-weight syntax transformations as rewrite rules over templates of Java expressions. Particularly, our technique TC-Infer accepts the type changes reported by RefactoringMiner as input and returns rewrite rules for these type changes as Comby templates.

To evaluate the *applicability* of TC-Infer, we applied it to infer rewrite rules for the most popular type changes applied in our corpus of 400K commits from 130 projects. We found that TC-Infer reported 4931 rewrite rules for 522 popular type changes from our corpus. These type changes are diverse in nature - they comprised 1. varied type kinds (e.g., primitives, paramterized types), 2. varied namespaces (e.g. JDK, project specific types or external third-party library types) 3. interoperable types (e.g., `StringBuffer`→`StringBuilder`) and non-interoperable types (e.g., `String`→`List<String>`). Further, to demonstrate the *effectiveness* of TC-Infer in the real world, we evaluate its accuracy on a dataset of 245 commits containing 3060 instances of 60 diverse type change patterns. We manually validated the changes, and our results show that rules produced by TC-Infer have precision of 99.2% and recall ranging from 60% upto 100%.

We also demonstrate the utility of TC-Infer by developing a plugin for the IntelliJ IDEA that provides assistance to developers to perform type changes. To evaluate the utility of IntelliT2R, we run IntelliT2R on four performance-critical open-source projects. IntelliT2R generated 50 type changes which compile and pass tests successfully. At the time of writing, the original developers have already accepted 15 of them.

In summary, this chapter makes the following contributions:

1. TC-Infer: A technique that analyses the type changes performed in a commit and then deduces the performed adaptations as rewrite rules in the Comby language

2. We empirically evaluated our TC-Infer to demonstrate its *applicability*, *effectiveness* and *utility*

3. IntelliT2R: An IDE plugin that assists developers at performing type changes by

[1]https://www.jetbrains.com/help/idea/type-migration.html

Table 5.1: Motivating Examples

	Element Before & After	**Usage before & After**	`RewriteRule`
1.	int x; → **long** x;	x = 0; → x = 0**L**	:[n∼\d+]→:[n]L
2.	File x; →**Path** x;	x.exists() → **Files.exists**(x)	:[r].exists()→Files.exists(:[r])
3.		new FileOutputStream(new File(x, f)) →Files.**newOutputStream**(x.**resolve**(f))	new FileOutputStream(new File(:[a],:[b])) → Files.newOutputStream(:[a].resolve(:[b]))
4.	boolean x; →**AtomicBoolean** x;	x=true; →x.**set**(true)	:[l]=:[r∼true]→:[l].set(:[r∼true])
5.	:[t] x; →**Optional<**:[t]**>** x;	x.substring(1,5); → x**.get()**.substring(1,5);	:[r]→:[r].get()
6.		x = null; → x = **Optional.empty()**	null→Optional.empty()
7.		Optional.of(Utils.trx(x)) → x.**map**(Utils**::**trx)	Optional.of(:[r].:[m](:[a])) →:[a].map(:[r]**::**[m])
8.	Optional**<Integer>** x; → **OptionalInt** x;	x = Optional.empty() →x = **OptionalInt**.empty()	Optional.empty()→OptionalInt.empty()
9.	AtomicLong x; → **LongAdder** x;	x.get() → x.**sum**()	:[r].get()→:[r].sum()
10.		x.set(0)→x.**reset**()	:[r].set(0)→:[r].reset()
11.	List**<**:[t]**>** x; → **Set<**:[t]**>** x;	x = new ArrayList**<>**(items) → x = new **HashSet<>**(items)	new ArrayList**<>**(:[a])→new HashSet**<>**(:[a])
12.		ls.get(0) → ls.**iterator().next()**	:[r].get(0) → :[r].iterator().next()

surfacing the rules produced by TC-Infer.

5.2 Motivating Examples

Table 5.1 showcases a few scenarios that highlight the intricacies associated to inferring the rewrite rules. The first two columns (*Elements Before/After*) show the element upon which the type change is performed, the next two columns (*Usage Before/After*) present the adapted usage of the element, and the last column presents the *rewrite rules* encoding the adapted usages. For instance in row 9, the type change from `AtomicLong` to `LongAdder` involves renaming the callsite from `get` to `sum`. This adaptation is represented by the rewrite rule `:[r].get()→:[r].sum()`. The left side of the rule is an arbitrary Java expressions with a template variable (`:[r]` binds the source code to template variable `r`), which is matched to a program AST. The right side of the expression is also a Java expression with holes, where each template variable denotes a substitution

with an appropriate fragment of the program AST, as matched on the left side.

A plethora of edits are applied to adapt the usages of the element: Adding the `L` suffix (Table 5.1 Row 1), replacing an instance method with a static method invocation (Table 5.1 Row 2), updating a static method invocation (Table 5.1 Row 8) or updating a class instance creation (Table 5.1 Row 11). Often these edits adapt the a commonly used idiom of a type. For instance, in (Table 5.1 Row 12) when the type change from `List` to `Set` is performed, the idiom `ls.get(0)` is replaced with the idiom `ls.iterator().next()`. Similarly in (Table 5.1 Row 7) when the variable is wrapped with the `Optional` data type, the idiom that involves invoking a static method `Utils.trx(x)` gets converted to using the `map` method with a member reference to the static method `Utils::trx`. The adaptations can also involve a composition of two edits. For instance in (Table 5.1 Row 3) the type change from `File` to `Path` requires the nested call to two constructors `new FileOutputStream(...)` and `new File(...)` to be converted to a static method invocation `Files.newOutputStream()` and an instance method invocation `resolve()`. It can readily be seen that, constructing these rules by hand can be cumbersome. However, all the current type migration techniques require the user to do so.

While some type changes are performed between inter-operable types (e.g. `File`→`Path` or `StringBuffer`→`StringBuilder`), others can be semantic altering (e.g. `List<String>`→`Set<String`). Each type change could have its own set of preconditions, apart from the general ones described by Balaban et al.[13]. For instance, Dig et al[36] proposed special preconditions for introducing concurrency (`Map`→`ConcurrentMap`) or Ketkar et al.[74] proposed special preconditions for eliminating *boxing*. One can imagine that, type changes like `List`→`Set` or `LinkedList`→`Deque` or `String`→`List<String>` will have their own set of specialized preconditions. Therefore, proposing a general technique that can completely automate the application of any type change is extremely challenging. However, given that a developer wants to perform a particular type change (semantic altering or not), it can be extremely useful if a tool can suggest(and apply) the transformations needed to adapt common idioms. For instance, when performing the type change `List`→`Set`, developers usually adapt the idiom `new ArrayList<>()` to `new HashSet<>()` and adapt `ls.get(0)` to `ls.iterator().next()`. The goal of our technique is to infer rewrite rules for the adaptations applied to common syntactic idioms in the previously performed type changes, and the suggest these rules to the user in the future when performing the type change.

5.3 Technique

Algorithm 1 Overview of TC-INFER

```
1: function TcInfer(typeChangePattern,commits)
2:     for commit in commits do
3:         tcInstances, rfctrs = RefactoringMiner(commit)
4:         if tcInstance.tcPattern == typeChangePattern then
5:             for tcInstance in tcInstances do
6:                 for adpt in tcInstance.adaptations do
7:                     adpt ← PreProcess(adpt, rfctrs)
8:                     for rule in InferRules(adpt) do
9:                         if relevant(rule, typeChangePattern) then
10:                            yield rule
```

TC-INFER is a technique that produces the rewrite rules applied for adapting the source code to particular type change patterns (e.g., `String`→`Path`) performed in the input commits. Algorithm 1 describes the overall approach of TC-INFER. First, we collect all type change instances and other refactoring instances identified by REFACTORINGMINER in each input commit. Next, we pre-process the adapted statements reported for each type change instance where the input type change pattern was performed, and then infer the rewrite rules capturing each adaptation. We report all the rewrite rules that are relevant to the input type change pattern. These rewrite rules express the syntactic transformations required to adapt the source code elements to a particular type change pattern. At the heart of TC-INFER is the AST differencing algorithm INFERRULES (introduced in Algorithm 3) which involves two main steps - (i) establishing the mapping between most similar nodes in the AST, and (ii) deducing rewrite rules that if performed on the former AST produces the later one.

5.3.1 Basic Concepts

We will now describe some basic concepts.

Definition 5.3.1 (AST)**.** Let T be an *AST*. The tree T has one *root* node. Each node $t \in T$, has a *parent* $p \in T$ (except for the root). Each node $t \in T$, has a list of *children*. Each node $t \in T$, has an associated *label* (i.e., AST node kind) and a *value*, which is a string.

Definition 5.3.2 (TEMPLATE)**.** A lightweight way of matching syntactic structures of a program's parse tree, like expressions and function blocks. It is basically an arbitrary Java expression with template variables (or holes), that is matched to program AST.

Recently researchers van Tonder and Le Goues [145] proposed COMBY, a multi-language syntax transformation technique for declaratively rewriting syntax with templates. We use the Java instantation of COMBY as our templating engine. A brief overview of the template syntax and matching behavior can be found on its website[2].

Definition 5.3.3 (TEMPLATEVARIABLE)**.** According to COMBY's syntax, `:[n]` binds the source code to a template variable `n`. A template variable can match all characters (including whitespace) lazily up to its suffix (analogous to .*? in regex), but only within its level of balanced delimiters.

Our technique matches the code snippet to a general template variable `:[n]` if the code snippet does not match the following specific kinds of template variables:

- `:[[a]]` - matches identifiers, analogous to \w+ in regex
- `:[n~[+-]?(\d*\.)?\d+\$]` and `:[n~\d+]` - matches numbers
- `:[h~0[xX][0-9a-fA-F]+]` - matches hexadecimals
- `:[[exc~([A-Z][a-z0-9]+)+]]` - matches class names
- `:[[exc~\``(.*)\``]]` - matches string literals
- `:[c~^[A-Z]+(?:_[A-Z]+)*]` - matches constants

Considering such specific kinds of template variables allows us capture richer context when inferring the rewrite rules. It also minimizes the spurious matches when applying the rewrite rules.

Definition 5.3.4 (REWRITERULE- $L \rightarrow R$)**.** The left side of REWRITERULE is a TEMPLATE that is matched to a program AST, while the TEMPLATE on the right side contains TEMPLATEVARIABLE that denote the substitution with an appropriate fragment of the program AST, as matched on the left side. For instance, the rule

[2]https://comby.dev/docs/syntax-reference

`:[v].exists()`→`Files.exists(:[v])` will match concrete instances like `f.exists()` and `mngr.getResource().exists()`, and will rewrite them to `Files.exists(f)` and `Files.exists(mngr.getResource())`, respectively.

Definition 5.3.5 (GETTEMPLATEFOR)**.** Given a code snippet c, this operation returns a template that captures the structure of the entire code snippet. To generate such a template, the source code snippet is parsed as AST, and each child of the root of the AST is replaced with a template variable, iff the child is not a token(s) that has a special meaning in Java's grammar (e.g., keywords like `new` or `return`, or special characters like `,` or `;`). - See Example 5.3.1

In other words, it is a template which only consists of holes and special Java tokens. The idea of inferring such structural templates for code snippets is inspired from recent work by Luan et al. [87].

Definition 5.3.6 (MATCH)**.** Given a template T and a code snippet c, this operation returns a mapping between the TEMPLATEVARIABLEs in T and sub-expressions of c; iff the T matches the entire snippet c and each template variable in T is bound to a valid Java syntax tree. - See Example 5.3.1

Definition 5.3.7 (SUBSTITUTE)**.** Given a TEMPLATE T and mappings from TEMPLATEVARIABLEs in T to syntactically valid Java expressions, this operation returns the template T' where the TEMPLATEVARIABLEs in T are replaced with the corresponding expressions.- See Example 5.3.1

Definition 5.3.8 (REWRITE)**.** Given a rewrite rule $L \rightarrow R$ or a list of rules $L_1 \rightarrow R_1$, $\ldots$ $L_n \rightarrow R_n$ and a code snippet c, this operation applies (sequentially) the input rewrite rule on c.

Definition 5.3.9 (INTERSECT($\cap$))**.** Given two matches $m1$ and $m2$ (i.e., the output of the MATCH operation), this operation returns a mapping between TEMPLATEVARIABLEs across $m1$ and $m2$ that bind to the same value. In other words, it is a set intersection over the values the TEMPLATEVARIABLEs are bound to. - See Example 5.3.1

Definition 5.3.10 (INTERSECT-ISSUBTREE($\cap^s$))**.** Given two matches $m1$ and $m2$ this operation returns a mapping between TEMPLATEVARIABLEs such that the the value bound to the TEMPLATEVARIABLEs of $m2$ is a subtree of the values bound to TEMPLATEVARIABLEs of $m1$.- See Example 5.3.1

Definition 5.3.11 (DIFFERENCE(−))**.** Given two matches $m1$ and $m2$, the operation $m1 - m2$ would return TEMPLATEVARIABLEs from $m1$ that are bound to a value that no variable in $m2$ binds to. In other words, it is a set difference operation over the value that the TEMPLATEVARIABLEs are bound to. This operation returns a list of TEMPLATEVARIABLEs sorted by size of its value. - See Example 5.3.1

Example 5.3.1. Examples demonstrating the basic operations:
c1 = `x.substring(1)`
c2 = `x.get().substring(1)`
t1 : GETTEMPLATEFOR(c1) = `:[r].:[m](:[a~\d+])`
t2 : GETTEMPLATEFOR(c2) = `:[r'].:[m'](:[a'~\d+])`
m1 = MATCH(c1, t1) # {`r:x,m:substring,a:1`}
m2 = MATCH(c2, t2) # {`r':x.get(),m':substring,a':1`}
s1 = SUBSTITUTE(t1, {`r:foo()`}) # `foo().[m](:[a])`
m1 ∩ m2 = {`m:m', a1':a1'`}
m2 $\cap^s$ m1 = {`r':r`}
m1 − m2 = `[r]`
RENAMETEMPLATEVARIABLE(t1, {r : x}) = `:[x].:[m](:[a])`

5.3.2 Input

```
1
2  - File fldr;
3  + Path fldr;
4
5  - readfldr(fldr, mode, extensions)
6  + readfldr(fldr.toFile(), extensions)
7
8  - new ResourceHandler(new Handler(fldr.getAbsolutePath()))
9  + new ResourceHandler().set(new Handler(fldr.toAbsolutePath()
10 +                                    .toString())
11
12 - new FileOutputStream(new File(fldr,"test.txt"))
13 + Files.newOutputStream(fldr.resolve("test.txt"))
```

Figure 5.1: Type Change Instance reported by REFACTORINGMINER for the type change pattern `File→Path`

We use RefactoringMiner to collect type changes and other refactorings performed. It can detect 80 kinds of refactorings applied in the history of a Java project. These refactorings include sub-method level refactorings like rename variable, type changes or extract/inline variable. Recently Tsantalis et al. [142] have shown that RefactoringMiner can detect type changes with 99.7% precision and 94.8% recall. In particular it reports four kinds of type changes - *Change Variable Type*, *Change Parameter Type*, *Change Return Type*, and *Change Field Type*, along with the *relevant* statements updated across the commits that refer to the element whose type has changed, i.e., statements in the def-use chain (Figure 5.1). In the case of *Change Return Type* it reports the matched `return statements` of the corresponding method whose return type changed. However, it should be noted that RefactoringMiner reports the pairs of adapted statements that it could match. These could be a subset of all pairs of statements that were actually adapted to perform the type change. Identifying all adapted statements would require additional type-binding information and call-graph analysis, but RefactoringMiner works purely on syntax. As input our technique accepts a set *type change instances* reported by RefactoringMiner.

5.3.2.1 Pre-processing

It has been observed by previous researchers [76] that type changes are often complemented with other refactorings like renaming and extract/inline variable. However, these refactorings are not mandatory to be performed when a type change is performed. Therefore, we normalize the collected adaptations by undoing the renaming and extract/inline variable refactoring in the snippets. These key insights reduce the delta between the statement mappings reported by RefactoringMiner, thus reducing the number of noisy rewrite rules being generated (i.e., *false positives*).

5.3.3 Output

For each type change pattern (i.e. `int`→`long`, or `String`→`Optional<String>`) performed in the input type change instances, our technique will produce a set of rewrite rules (`RewriteRules`) that could be applied to adapt the usages to the constraints of the

new type.

```
TransformationSpec ::= RewriteRule RewriteRules
      RewriteRules ::= RewriteRule Guards RewriteRules | ∅
            Guards ::= TemplateVariable Guard Guards | ∅
             Guard ::= Type Guard | regex Guard | ∅
```

As shown above, `TransformationSpec` contains a `RewriteRule` to capture the type change pattern (`int`→`long` or `List<:[t]>`→`Set<:[t]>`) and the `RewriteRules`. In `RewriteRules`, each `RewriteRule` is associated to `Guards`, where these guards constrain the code snippet that binds to the `TemplateVariable`s, either based on *regular expressions* and/or the return type of the code snippet. For instance, for the rule `:[r].exists()`→ `Files.exists(:[r])` from Table 3.3 (Row 2), we record that the return type of `r` is `File`. Similarly in the rule `:[n`∼`\d+]`→ `:[n]L`, we infer two guards - return type of `n` is `int` and that `:[n]` is a number literal. While the regex `Guard` is expressed using the Comby language itself, we separately record the `Type` guard. These `Guards` minimize the spurious matches when applying the rewrite rules. `TransformationSpec` is basically an adaptation of the Twining syntax proposed by Nita and Notkin [111] to the Comby language with additional *regex* based guards.

For each rewrite rule, we also report the real-world instances where the adaptation expressed by the rewrite rule is performed. The number of unique instances, commits, and projects where a rewrite rule was applied are used as a heuristic for its *confidence*, when suggesting these rules to developers in the IDE. The rewrite rules encoded in Comby syntax can be losslessly translated to IntelliJ's structural replacement templates or to the DSL proposed by Balaban et al. [13] and Ketkar et al [74], since all of these are closely related to the Twining language [111].

5.3.4 TC-Infer

5.3.4.1 Generating the `RewriteRules`

Given two versions of a code snippet, the goal of GenerateRewriteRule in Algorithm 2 is to deduce the rewrite rule applied across the two versions. The higher level

intuition is to first capture the structure of the before and after code snippets as templates (T_1 and T_2) and then to refine them into a rewrite rule by mapping the holes of T_1 to the holes of T_2, if possible. We will now highlight the intricacies related to inferring the rewrite templates.

Algorithm 2 Generate Rewrite Rules

```
1: function RefineRule(LHS, RHS)
2:     RHS ← renameTemplateVars(RHS, LHS ∩ RHS)
3:     if any(LHS ∩^S RHS) or any(RHS ∩^S LHS) then
4:         LHS, RHS ← Decompose(LHS, RHS)
5:         LHS, RHS ← ↻ RefineRule(LHS, RHS)
6:     LHS ← Substitute(LHS, LHS − RHS)
7:     RHS ← Substitute(RHS, RHS − LHS)
8:     return RHS, LHS
9: function generateRewriteRule(c1, c2)
10:    LHS, RHS ← Match(getTemplateFor(c1), c1), Match(getTemplateFor(c2), c2)
11:    return RefineRule(LHS, RHS)
```

Example 5.3.2. Lets consider a simple example Table 3.3 Row 4).

```
1 - x = true
2 + x.set(true)
```

As described in Algorithm 2, we first construct a structural template (Definition 5.3.5) that matches the two code snippets - `:[lh]=:[rh]` and `:[r].:[m](:[a])` (Line 10). The two structural templates and their respective matches - {`lh:x, rh:true`} and {`r:x, m:set, a:true`} are passed to RefineRule. Then the TemplateVariables that map across the two templates (LHS ∩ RHS - Definition 5.3.10), are consistently renamed (Line 2)- i.e. `lh` → `r` and `rh` → `a`. Finally, the TemplateVariables that do not map across the two templates (LHS − RHS/RHS − LHS) are substituted with their concrete values (Line 6 & Line 7) - resulting in the rewrite rule `:[lh]=:[rh]`→`:[lh].set(:[rh])`.

Example 5.3.3. Let's consider the adaptation (Table 3.3, Row 7) applied to perform the type change from :[t]→Optional¡:[t]¿.

```
1 - Utils.trx(s)
2 + s.map(Utils::trx)
```

The structural template capturing the structure of these snippets are `:[r].:[m](:[a])` and `:[r'].:[m'](:[a'])` respectively. Consequently, the matches produced are

LHS={`r:Utils, m:trx, a:s`} and RHS={`r':s, m':map, a':Utils::trx`}. Since $LHS \cap RHS$ = {`a:r'`} we update the RHS to `:[[r']].:[[m']](:[[a']])` in Line 2 (i.e., r' renamed to a). In Line 3 we check if any template variables need to be further decomposed ($LHS \cap^S RHS$ ={`r:a'`}). Next, the source code bound to the variables a' is decomposed into the template (`:[x]:::[y]`) and is substituted into the RHS `:[[a]].:[m'](:[[x]]:::[y])`. In the recursive call the common variables are consistently renamed i.e. $x \rightarrow r, y \rightarrow m$ and the unmatched template variables are substituted with their concrete values, resulting in the rewrite rule - `:[[r]].:[m](:[a])` $\rightarrow$ `:[a].map(:[[r]]:::[m])`.

5.3.4.2 Establishing mappings

Algorithm 3 The InferRules procedure

```
1: function getWeight(n1, n2):
2:     Rules ← InferRules(n1, n2)
3:     return Max(NumberOfTokensBoundToVars(Rules))
4: function getOptimalPairs(ns1,ns2)
5:     return HungarianMethod(ns1, ns2, getWeights)
6: function InferRules(n1, n2)
7:     if not isIsomorphic(n1, n2) then
8:         (LHS, RHS) ← GenerateRewriteRule(n1, n2)
9:         subRules ← []
10:        for c1, c2 in getOptimalPairs(n1.children, n2.children) do
11:            subRules.extend(InferRules(c1.value, c2.value))
12:        coarsestEdits = largestNonOverlapping(subRules)
13:        if Rewrite(coarsestEdits, n1) == n2 then
14:            if Rewrite((LHS, RHS), n1) == n2 then
15:                return subRules.append((LHS, RHS))
16:            return subRules
17:        else
18:            (LHS, RHS) ← Merge(subRules, (LHS, RHS))
19:            if Rewrite((LHS, RHS), n1) == n2 then
20:                return [(LHS, RHS)]
21:    return []
```

As described in Section 5.3.2, for each element whose type has changed, RefactoringMiner reports the *relevant* code snippets that are adapted, but it does not capture

the exact edits that are performed across the two snippets. In this section we will explain Algorithm 3, that looks for mappings between the two matched statements reported by RefactoringMiner. This algorithm (described in Algorithm 3) is based on how developers would naturally attempt to construct rewrite rules - search for unmodified pieces of code, then from the remaining figure out which containers of source code can be mapped to each other and then finally look for the precise mappings between the code snippets in the mapped containers. InferRules produces a flattened list of rewrite rules, that capture the atomic edits and composite edits applied across the two code snippets.

Example 5.3.4. Let's consider the following statement from Figure 5.1 adapted to perform the type change `File`→`Path`.

```
1 - new ResourceHandler(sourceDir,new Handler(new File(fldr)))
2 + new ResourceHandler().set(new Handler(Paths.get(fldr)),sourceDir)
```

For the given input nodes $n1$ and $n2$, TC-Infer first computes the rewrite template `new :[c](:[s],new Handler(new File(fldr)))` → `new :[c]().set(new Handler(Paths.get(fldr),:[s])` by invoking GenerateRewriteRule i.e. Algorithm 2. To deduce more fine-grained mappings, TC-Infer attempts to optimally pair the children of the nodes $n1$ and $n2$. Naively pairing the children in the order they appear is not a sound approach for two main reasons - (i) AST kind of $n1$ may not be same as $n2$ (in this example $n1$ is of the kind *class instance creation* and $n2$ is of the kind *method invocation*) (ii) children might be reordered, added or removed (in this example the method `set` accepts the arguments in the reverse order). Therefore, in our example TC-Infer will pair `new Handler(new File(fldr))` with `new Handler(Paths.get(fldr))` and `sourceDir` with `sourceDir` (Line 10). Consequently it will pair `new File(fldr)` with `Paths.get(fldr)`, and produce the rewrite rule `new File(:[a])`→`Paths.get(:[a])`.

To find optimal pairs we implemented and applied the *Hungarian method* [81], that tackles the *assignment problem* (Line 10). This problem consists of finding, in a weighted bipartite graph, a matching of a given size, in which the sum of weights of the edges is a minimum (or maximum). We treat the two lists of children as the partition and maximize the number of tokens bound to template variables in the rewrite rules inferred between the paired nodes. The optimal pairing not only allows us to continue finding more fine grained rules when the root nodes kinds don't match, but also accounts for reordering or

alteration of the children list. The method getWeights that provides weights to the HungarianMethod invokes the method Infer, moreover Infer is recursively invoked at Line 11. We prevent this redundant computation by tabulating the inferred templates against the offsets of the updated location.

5.3.4.3 Inferring composite rewrite rules

Example 5.3.5. Lets consider the adaptation from Table 3.3, row 3.

```
1 - new FileOutputStream(new File(fldr,"test.txt"))
2 + Files.newOutputStream(fldr.resolve("test.txt"))
```

While the operation generateRewriteTemplate can deduce the template variables for generalizing source code that is equal across the edit, it cannot deduce composite rewrite rules. In this example, first TC-Infer computes the rewrite rule **R1** = `new FileOutputStream(new File(fldr,"test.txt"))` → `Files.newOutputStream(fldr.resolve("test.txt"))`. At this step no template variables were inferred. Next, it deduces finer mappings from `new File(fldr,"test.txt")` to `fldr.resolve("test.txt")`. For this mapping, the template **R2**=`new File(:[a1],:[a2])` → `:[a1].resolve(:[a2])` is deduced. After it has collected the inferred rules for the optimal pairs of children nodes, it identifies the largest non overlapping rules Line 12. It then applies these edits to the input node $n1$ and checks if it yield node $n2$. It can be observed that, in our example applying the template **R2** upon `new FileOutputStream(new File(fldr,"test.txt"))` will not yield `Files.newOutputStream(fldr.resolve("test.txt"))`. Therefore, TC-Infer now attempts the merge the rewrite rules inferred for children **R2** into the rewrite rule learnt for the parent node **R1** to produce **R3** - `new FileOutputStream(new File(:[a1],:[a2]))` → `Files.newOutputStream(:[a1].resolve(:[a2]))` (Line 18). Infer will also report **R2** because it correctly captures the edit applied between `new File(fldr,"test.txt")` → `fldr.resolve("test.txt")`.

The function InferRules returns a flattened tree of edits, where the children edits are more fine-grained than the parent edit. Therefore, in Line 13 when we check if subRules transform node $n1$ to node $n2$, we consider the *coarsest* subrules (Line 12 largest non-overlapping edits) because these larger rules will be merged composite rewrite rules

of the fine-grained rules.

5.3.4.4 Enriching template variables with types

A template variable (like `:[a]`) in a rewrite rule can bind to any valid Java AST between closed parenthesis. This can lead to spurious matches. To avoid unintended matching, for each template variable we also record the return type of the bound expression. We obtain this information from the type inference provided in *Eclipse* JDT. This type information can be leveraged by the the type migration tools to minimize mismatches.

5.3.4.5 Identifying relevant edits

The updated statements reported by RefactoringMiner for each type change instance can also contain edits (some updated literals or expressions) that are not type dependent upon the root of type change. We consider an edit rewrite rule relevant to the type change from type S to type T, (i) if the return type of the concrete expression captured by the `LHS` of the rewrite rule is S (e.g., object creation or literals), (ii) if the rewrite rule contains template variables that match an expression (e.g. variable reference) of type S.

5.3.4.6 Eliminating *Unsafe* RewriteRules

The problem of expressing the change as a rewrite rule is that, any token (s) that does not appear in the before input code snippet(`n1`) but appears in the after code snippet(`n2`) will not be generalized as a hole. Therefore, if the adaptation involves usage of a new variable or a new string it cannot generalize it with respect to the larger context because it has access to the AST that matched the left side. Growing the size of the match to include the declaration of the variable might just make the rewrite rule more context specific. It is unclear how these scenarios could be expressed as rewrite rules therefore we mark these rules as *unsafe* and optionally eliminate them from the output.

5.4 Evaluation

To understand the effectiveness and the real-world relevance of our technique, we answer four research questions :

RQ1. How applicable is TC-Infer?Using TC-Infer is beneficial if rewrite rules inferred for a particular type change from one commit could be applied in another commit to perform the same type change. Are such scenarios common?

RQ2. Can we trust the existing practice for performing type changes?In this question we investigate if manually performing type changes could unknowingly introduce idioms for which there are better alternatives. This will highlight the importance of standardizing type changes with tools.

RQ3. How effective are the RewriteRules for performing type changes?In this question we compare the application of the rewrite rules inferred by TC-Infer to the changes performed by real-world developers. This will showcase the benefits and highlight the pitfalls of TC-Infer.

RQ4. Did developers find the RewriteRules useful? In this question we investigate if the rules produced by TC-Infer are useful to the developers to perform type changes in their IDEs.

5.4.1 Dataset

Our corpus consists of 416,652 commits from 129 large, mature and diverse Java projects, used by other researchers [89] to understand language constructs in Java. This corpus [89] is shown to be very diverse, from the perspective of LOC, age, commits, and contributors. This ensures our study is representative. It is also large enough to comprehensively answer our research questions. The complete list of projects is available online[3].

In this study, we consider all commits in the epoch January 1, 2015 – June 26, 2019, because researchers observed an increasing trend in the adoption of Java 8 features after 2015. Java 8 introduced new APIs like `FunctionalInterface`, `Stream`, `Optional` and enhanced the `Time`, `Collection`, `Concurrency`, `IO` and `Math` APIs. Thus, we use these

[3]http://changetype.s3-website.us-east-2.amazonaws.com/docs/P/projects.html

particular projects and their commits in this particular epoch, because it allows us to collect and study type changes involving the new (≥`Java 8`) built-in Java types. We excluded all merge commits, as done in other studies [126], to avoid having duplicate result.

We applied RefactoringMiner upon these 416,652 commits and identified 72,863 commits where at least one type change was performed. Next we group all the type change instances reported by RefactoringMiner based on the type change pattern - (sourceType, targetType) and identify 605 type change patterns as *popular type change patterns* because they are applied in at least 2 unique projects. We apply TC-Infer upon the instances reported by RefactoringMiner for these 605 *popular type change patterns*.

5.4.2 How applicable is TC-Infer?

In this question we explore the type changes that can benefit from TC-Infer, their various characteristics and how applicable was TC-Infer for these type change patterns. To find popular type changes, we first applied TC-Infer upon the all the instances corresponding to the 605 *popular* type change patterns and collected 4,931 *safe* rewrite rules for 522 type change patterns (86.28%). Further, we identified 274 (52.49%) type changes for which TC-Infer reported atleast one *popular rule*. We consider a rule *popular* if it is applied to adapt to the same type change in more than one commit. We identified 832 *popular* rules for the 274 type changes. On investigating 13.72% type changes with no reported rule, we found that:

1. the source and the target types were semantically so different (e.g. `String`→`Map<Integer, String>`) that no *safe* rewrite rule could be inferred

2. the source and the target type were so inter-operable that there was no need for any update (e.g. replacing with super type, primitive widening, or boxing). Our observation is in congruence with the findings of the previous researchers [76] had who also found that in type changes across types that are hierarchically related (and by widening) require the least adaptation compared to types that share a composition and sibling relationship.

Figure 5.2 plots the distribution of the number of popular rules reported

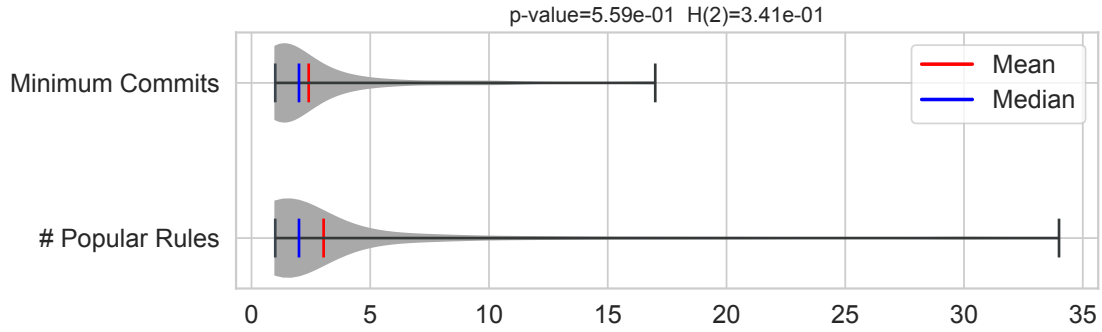


Figure 5.2: Distribution of the number of *popular* rules reported for each type changes and minimum number of commits required to infer the popular rules for each type change

for the 274 type changes. The mean *popular* rules inferred for each type change is 3.48. TC-INFER produces one *popular* rules for the type change `Function<X,Integer>`→`ToIntFunction<X>` , while for `long`→`int` it produced 34 *popular* rules.

Analyzing a single commit where a particular type change is performed will not surface all the *popular* rules because the updated code may not use all the popular idioms. The number of commits required to infer all the popular rules has a direct impact on the applicability of TC-INFER, because some type changes may not be performed in many commits. To evaluate this, we identify the smallest set of commits that contain all the *popular* rules for the type change in context. Computing this smallest set of commits can be viewed as the *Set Cover* problem. Given a set of elements $\{1, 2, \ldots, n\}$ (called the universe, in our case all the *popular* rewrite rules for each type change) and a collection m weighted sets (in our case these are commits) whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe, with minimum weight. While this problem is *NP-Complete*, its greedy approximation algorithm [158] suffices our purpose, since the cardinality of our universe is not very large (≤ 34). We applied this algorithm on the *popular* rewrite rules and the commits for each type change pattern, and identified the minimum set of commits that cover all the *popular* rewrite rules reported for the type change pattern. Figure 5.2 plots the distribution of the cardinality of this minimum set for the 274 *popular* type changes. On an average TC-INFER required approximately 2 commits to infer all the

popular rewrite rules for a type change, and required at most 18 commits to infer the 34 rules for the type change from `int`→`long`.

The number of *popular* rewrite rules depend on various factors like - source and target type, availability of examples, and ability of TC-Infer to infer rules from the previously applied type changes. It is not possible to determine if TC-Infer has inferred all possible rewrite rules for a particular type change. Intuitively all the possible rewrite rules for a type change are covered when the rules covers all the methods/constructors/fields in the source type. Rather previous researchers Li et al. [85] adopt this in their formalization of API migrations. However, this is not applicable to type changes since type changes have to consider *all* the possible usages of a particular type. For instance, updating wrapper methods (`Integer.toString(x)`→`Long.toString(x)`), where it is not possible to enumerate all the common static method invocations that could act as such wrappers. In API migrations such scenarios do not arise. Moreover, since type changes can be semantics altering sometimes no mapping are found for any member method/field. For instance, when the variable `x`'s type was changed from `String`→`Optional<String>` is applied, no reported rewrite rule affected the invoked method of `String`, rather all the references were simply updated to use `x.get()` (e.g., `x.trim()`→`x.get().trim()`).

Moreover, TC-Infer was able to infer rewrite rules for diverse type change patterns :

1. **Different AST Kind Nodes:**`PrimitiveType` → `PrimitiveType` (e.g., `int`→`long`), `PrimitiveType` → `SimpleType` (e.g.`int` → `OptionalInt`), `ArrayType` → `ParameterizedType`(e.g.,`byte[]` → `ByteBuffer`),...

2. **Differnt Namespaces:**`JDK Type` → `JDK Type` (e.g.,`Predicate` → `IntPredicate`) , `JDK Type` → `External Type` (e.g.,`java.util.List` → `com.google.ImmutableList`) , `Internal Type`→`JDK Type`, (e.g.,`FsPath` → `Path`) ...

3. **Interoperable**(e.g., `File` → `Path`) and **non-Interoperable** (e.g.,`List` →`Set`).

Table 5.2: Identified spurious RewriteRule introducing commonly disregarded idioms and the equivalent recommended RewriteRule

Type Change	Spurious Rule	n[1]/C[2]/P[3]	Recommended Rule	n/C/P
File→Path	:[v].getCanonicalPath()→ :[v].toString()	12/7/5	:[v].getCanonicalPath()→ :[v].toRealPath().toString()	15/8/3
	:[v].getCanonicalPath()→ :[v].toAbsolutePath().toString()	8/6/3		
	:[v].getAbsolutePath()→ :[v].toString()	60/8/6	:[v].getAbsolutePath() →:[v].toAbsolutePath().toString()	57/7/3
int→long	:[v]→(int):[v]	58/51/25	:[v]→Math.toExactInt(:[v])	8/4/4
:[t]→List<:[t]>	:[v]→Arrays.asList(:[v])	10/5/2	:[v]→Collections.singletonList(:[v])	9/5/2
:[t]→Optional<:[t]>	:[v] == null→ !:[v].isPresent()	2/2/1	:[v]→:[v].isEmpty()	3/2/1

1 **n**: frequency of the rewrite rule in our dataset
2 **C**: unique commits where the rewrite rule was performed
3 **P**: unique projects where the rewrite rule was performed

5.4.3 Can we trust the existing practice for performing type changes?

In this question we want to understand if the current practice of performing type changes are reliable enough to learn from. Do developers introduce bugs, inconsistencies or commonly disregarded code idioms when performing type changes? This will highlight the importance of standardizing type changes via tools.

We first identify *popular* type changes from our dataset, such that the authors of the paper can easily find documentation and discussions related to these types. For this purpose, we randomly sample 85 (approximately one-third) type change patterns from the 274 patterns for which a *popular* rule was reported. We then exlude all patterns involving *Internal* or project specific types, and identify 60 type change patterns for which documentation and discussions are publicly available. To answer this question, we manually investigate each of the 191 *popular* rewrite rules corresponding to the 60 type changes. The two authors that investigated these have five and two years of professional software development experience, respectively. We check whether 1. the rule is correct (i.e., similar to what a human would produce) based on the corresponding concrete examples from where the rule was inferred, 2. the rule preserves the semantics, and 3. the rule does not deteriorate a quality of the program by introducing a commonly disregarded code idiom. We obtained the list of commonly disregarded code idioms from

the IntelliJ's Java Code Inspections.[4]

We found that all 191 rules were correct, i.e., similar to what a human would produce from the concrete example. Further analysis of the 191 rules revealed *six* non-conforming rewrite rules, as shown in Table 5.2 for four type changes. The first two rewrite rules in the second column of Table 5.2 are semantically not the same, because `getCanonicalPath` resolves the path by accessing the local file system, while the methods `getAbsolutePath` and `toString` do not. Casting a `long` value to a `int` type is an unsafe practice because it does not handle the possible number overflow, instead Java 8's `Math.toIntExact` is recommended. We noticed that in the real world, developers sometimes apply some nonconforming rules that introduce unnecessary inconsistencies, performance or maintainability overheads. However, in the majority of cases the developers followed the best practices, thus we can learn from the wisdom of the crowd. This highlights the importance of standardizing the adaptation applied for performing type changes using rewrite rules that are verified by domain experts or by intelligent systems [1].

RQ2 Conclusion: We found that the 6 out of 191 *popular* rewrite rules that were non-conforming, were frequently applied; highlighting a need for automated technique that standardize type changes.

5.4.4 How effective are the RewriteRules for performing type changes?

To evaluatate the *effectiveness* of the rewrite rules inferred by TC-Infer, we replicate some type changes performed in our corpus and semi-automatically compare them to the changes applied by the original developer. For this purpose we developed IntelliT2R, that is built upon *IntelliJ's Type Migration* framework [67], and can be configured via the `TransformationSpec` produced by TC-Infer. We then compare the changes performed by IntelliT2R to those performed by the original developers.

[4]https://www.jetbrains.com/help/idea/list-of-java-inspections.html#probable-bugs

5.4.4.1 Identifying Test Scenarios

Choosing the commits for evaluating the *effectiveness* of our technique is not as straightforward as in the case of API Migration [83, 156], because randomly selected commits might not be use all corresponding APIs and operators. Therefore, for each type change pattern we identify the set of commits that *at least* contains all *popular* adaptations from our dataset,by using the minimum sets of commits identified in Section 5.4.2. We replicate the type changes applied in these 245 commits and manually compare them to the changes applied by the original developers.

5.4.4.2 Validating the Edits

For each statement s_p containing these type dependent idioms (e) in the parent commit(p), we find its matched statement s_c in the child commit(c). To obtain the *real mapping* (i.e., the adaptation applied by the original developer), we search RefactoringMiner's reported statement mappings to find a mapping containing statement s_p. If RefactoringMiner does not have a mapping for s_p, we get this information from the mapping store obtained by applying the *GumTree* algorithm [51] upon the files containing s_p in the parent and child commit, respectively. If any rewrite rule from our dataset transforms s_p into s_c, we consider it as a *True positive*. Otherwise, we run InferRules (Algorithm 3) upon the *real mapping* and collect the generated rewrite rules (R). We then apply R (if $R \neq \varnothing$) upon the identified idioms in commit p, and manually validate the edit.

1. *True Positive:* the rule(s) R applied on s_p correctly adapts to the type change. R could not transform s_p to s_n because the original developer had applied other unrelated overlapping changes.

2. *False Positive:* the rule(s) in R produces incorrect code because the rewrite rule mismatched when applied in context.

3. *Not Applicable:* $R = \varnothing$ and the performed adaptation involves usage of new additional functionality, or other unrelated changes.

4. *False Negative:* $R = \varnothing$ but the performed change is *Applicable*, implying Infer *could not capture the adaptation as a rewrite rule.*

Note that, running InferRules again on the *real mapping*, prevents us from counting a scenario *false negative* even when the correct rewrite rule was unavailable in our dataset. These scenarios occur because RefactoringMiner's statement matching algorithm fails to match and report these cases. We believe that RefactoringMiner can be further fine tuned to handle these scenarios. Our goal is to highlight the capabilities and expose the limitations of TC-Infer at deducing rewrite rules, for further improvement.

5.4.4.3 Results

In Table 5.4 we summarize the results of our experiment. It can be seen that in almost all the cases the precision is 100%. However, this is unsurprising since TC-Infer is very conservative when producing the rewrite rules (pre-processing the snippets, and identifying relevant and safe rules). Investigating the false positives revealed that other overlapping refactorings and semantic non-altering changes confused our technique (Algorithm 3). For instance, for the adaptation `(Long)Utilities.getRow()`→`(long)getRow()`, InferRules could produced the rule `(Long)Utilities.:[v]`→`(long):[v]` because our technique does not account for `Import as Static Method` refactoring.

We are more interested in the recall of the rules produced by our technique, i.e., the instances where our technique was not able to produce any rule for a particular adaptation. It can be seen that we have recall ranging from 67% for `Map<String,String>` → `Properties` to 100% for `AtomicLong`→`LongAdder`. We manually investigated each false negative and found three main reasons leading to them:

1. **Requires additional context**. The most common reason for *TC-Infer* to produce no rules across a given statement mapping ($s_p \rightarrow s_c$), is that the adaptation requires more information from the context than what was captured by the statement mappings. We observed that adaptations use elements (like variables) existing in the context, or require new elements to be created in the context. In the below example, the adaptation requires an instance variable of the type `Channel` from the context to replace the static method invocation with instance method invocation.

```
1 - final ChannelBuffer buffer=ChannelBuffers.buffer(6)
2 + final ByteBuf buffer=channel.alloc().buffer(6)
```

Capturing such edits will require comparing the changed data/control-flow across the

Table 5.3: Evaluated type changes

Type Change	n	#A	#UR	TP	NA	P	R
:[v0]→List¡:[v0]¿	95	43	15	27	9	1.00	0.79
:[v0]→Optional¡:[v0]¿	30	51	11	49	2	1.00	1.00
:[v0]→AtomicReference¡:[v0]¿	6	19	7	14	5	1.00	1.00
:[v0]→Supplier¡:[v0]¿	8	12	7	12	0	1.00	1.00
Entry¡:[v1],:[v0]¿→Entry¡:[v0],:[v1]¿	7	19	8	19	0	1.00	1.00
boolean→AtomicBoolean	4	11	5	10	1	1.00	1.00
byte[]→ByteBuffer	36	51	15	49	2	1.00	1.00
ImmutableList¡:[v0]¿→ImmutableSet¡:[v0]¿	2	5	1	5	0	1.00	1.00
Mongo→MongoClient	9	23	9	23	0	1.00	1.00
double→int	4	16	4	8	8	1.00	1.00
float→double	124	49	17	48	1	1.00	1.00
int→Duration	15	29	9	25	1	0.89	1.00
int→AtomicInteger	4	15	5	15	0	1.00	1.00
int→long	552	108	14	108	0	1.00	1.00
BufferedOutputStream→OutputStream	2	2	1	2	0	1.00	1.00
File→Path	18	33	16	33	0	1.00	1.00
File→hadoop.fs.Path	8	23	9	13	1	1.00	0.59
FileInputStream→InputStream	8	12	2	12	0	1.00	1.00
Boolean→boolean	9	19	10	19	0	1.00	1.00
Integer→int	190	48	39	45	1	1.00	0.96
Long→long	24	61	20	60	1	1.00	1.00
String→byte[]	38	10	4	7	3	1.00	1.00
String→int	6	7	7	7	0	1.00	1.00
String→File	26	33	8	31	2	1.00	1.00
String→InetSocketAddress	2	6	2	6	0	1.00	1.00
String→Path	11	18	5	16	2	1.00	1.00
String→UUID	5	4	2	4	0	1.00	1.00
String→regex.Pattern	18	12	7	12	0	1.00	1.00
StringBuffer→StringBuilder	517	105	4	103	2	1.00	1.00
Path→File	8	14	7	14	0	1.00	1.00
SimpleDateFormat→DateTimeFormatter	9	22	8	20	2	1.00	1.00
Date→Instant	15	25	7	21	3	1.00	0.95
Date→LocalDate	19	32	13	24	8	1.00	1.00
LinkedList¡:[v0]¿→Deque¡:[v0]¿	9	16	7	16	0	1.00	1.00
List¡:[v0]¿→ImmutableList¡:[v0]¿	15	12	4	11	0	0.92	1.00
List¡:[v0]¿→LinkedList¡:[v0]¿	9	24	4	22	1	1.00	0.96
List¡:[v0]¿→Set¡:[v0]¿	50	91	37	83	6	1.00	0.98
Map¡:[v1],:[v0]¿→ConcurrentMap¡:[v1],:[v0]¿	7	16	8	15	1	1.00	1.00
Map¡String,String¿→Properties	2	10	4	9	0	1.00	0.90
Optional¡Integer¿→OptionalInt	45	10	2	10	0	1.00	1.00
Queue¡:[v0]¿→Deque¡:[v0]¿	3	17	7	14	3	1.00	1.00
Queue¡:[v0]¿→BlockingQueue¡:[v0]¿	2	13	5	11	0	1.00	0.85
Random→SecureRandom	19	21	3	21	0	1.00	1.00
Stack¡:[v0]¿→Deque¡:[v0]¿	3	32	17	32	0	1.00	1.00
AtomicInteger→LongAdder	23	124	17	124	0	1.00	1.00
AtomicLong→AtomicInteger	2	11	3	6	5	1.00	1.00
AtomicLong→LongAdder	186	1026	22	1025	1	1.00	1.00
Function¡:[v0],Boolean¿→Predicate¡:[v0]¿	14	11	3	11	0	1.00	1.00
Function¡:[v0],Integer¿→ToIntFunction¡:[v0]¿	18	22	5	21	1	1.00	1.00
Supplier¡Integer¿→IntSupplier	8	15	2	15	0	1.00	1.00
long→BigInteger	17	4	2	4	0	1.00	1.00
long→Duration	10	15	4	14	1	1.00	1.00
long→Instant	7	13	13	13	0	1.00	1.00
long→AtomicLong	3	9	4	8	1	1.00	1.00
GetMethod→HttpGet	15	45	7	40	5	1.00	1.00
Log→Logger	424	300	6	295	5	1.00	1.00
ChannelBuffer→ByteBuf	39	59	12	32	15	1.00	0.93
DateTime→ZonedDateTime	283	256	25	249	3	1.00	0.98
TemporaryFolder→File	9	34	2	14	8	1.00	0.54
CompositeSubscription→CompositeDisposable	9	33	10	28	5	1.00	1.00

n: Number of type change instances **A**: Number of type dependent idioms
UR: Number of unique rewrite rules applied **TP**: True Positives
NA: Not Applicable **P**: Precision **R**: Recall

commit or reason about more source code surrounding the applied edit. Previous researchers [12, 83, 156] have developed techniques that can capture such context to perform library migrations and bug fixes. It is unclear how to declaratively express and apply them as rewrite rules. However, with latest developments in *language server protocols* this challenge is surmountable.

2. **Requires additional knowledge about the types**. We found that adapting statements for certain type changes requires deep understanding about the difference between the semantics of the before and after type. These adaptations involve identifying the mapping between the APIs, checking preconditions, and adapting the current program to leverage the properties offered by the new type. In this below example, the developer replaced the call to `add` with a custom logic that added a new functionality to leverage the constant time insertion that `LinkedList` offers via its `addFirst` and `addLast` method. However, inferring the addition of new functionality as a rewrite rule is currently out of our scope TC-Infer.

```
- List<String> ls
- ls.add(e);
+ LinkedList<String> ls
+ if (pred) ls.addFirst(e);
+ else ls.addLast(e);
```

Similarly, we observed that when developers change type from `List` to `Set`, they adapt the strategy that traverses the collection — from iterating over the collection with an index to using the `Iterator`.

3. **Requires additional inference**. In many cases, only reasoning about the syntactic transformations is not enough, because the adaptation also involves adapting the string literals. In the below example, the literal is updated from ‘‘`/status.txt`’’ to ‘‘`status.txt`’’, because the `resolve` method internally resolves the file separator.

```
- File f = new File(projectFldr + "/status.txt")
+ Path f = projectFldr.resolve("status.txt")
```

As an extreme case in this category we observed that, when the type change from `StringBuilder` to the new Java 8 type `StringJoiner` is performed, the adaptation may require data flow and control flow analysis to understand how the string is built, and then encoding this into the `StringJoiner` API.

Table 5.4: Evaluated type changes

Type Change	n	#A	#UR	TP	NA	P	R
:[v0]→List¡:[v0]¿	95	43	15	27	9	1.00	0.79
:[v0]→Optional¡:[v0]¿	30	51	11	49	2	1.00	1.00
:[v0]→AtomicReference¡:[v0]¿	6	19	7	14	5	1.00	1.00
:[v0]→Supplier¡:[v0]¿	8	12	7	12	0	1.00	1.00
Entry¡:[v1],:[v0]¿→Entry¡:[v0],:[v1]¿	7	19	8	19	0	1.00	1.00
boolean→AtomicBoolean	4	11	5	10	1	1.00	1.00
byte[]→ByteBuffer	36	51	15	49	2	1.00	1.00
ImmutableList¡:[v0]¿→ImmutableSet¡:[v0]¿	2	5	1	5	0	1.00	1.00
Mongo→MongoClient	9	23	9	23	0	1.00	1.00
double→int	4	16	4	8	8	1.00	1.00
float→double	124	49	17	48	1	1.00	1.00
int→Duration	15	29	9	25	1	0.89	1.00
int→AtomicInteger	4	15	5	15	0	1.00	1.00
int→long	552	108	14	108	0	1.00	1.00
BufferedOutputStream→OutputStream	2	2	1	2	0	1.00	1.00
File→Path	18	33	16	33	0	1.00	1.00
File→hadoop.fs.Path	8	23	9	13	1	1.00	0.59
FileInputStream→InputStream	8	12	2	12	0	1.00	1.00
Boolean→boolean	9	19	10	19	0	1.00	1.00
Integer→int	190	48	39	45	1	1.00	0.96
Long→long	24	61	20	60	1	1.00	1.00
String→byte[]	38	10	4	7	3	1.00	1.00
String→int	6	7	7	7	0	1.00	1.00
String→File	26	33	8	31	2	1.00	1.00
String→InetSocketAddress	2	6	2	6	0	1.00	1.00
String→Path	11	18	5	16	2	1.00	1.00
String→UUID	5	4	2	4	0	1.00	1.00
String→regex.Pattern	18	12	7	12	0	1.00	1.00
StringBuffer→StringBuilder	517	105	4	103	2	1.00	1.00
Path→File	8	14	7	14	0	1.00	1.00
SimpleDateFormat→DateTimeFormatter	9	22	8	20	2	1.00	1.00
Date→Instant	15	25	7	21	3	1.00	0.95
Date→LocalDate	19	32	13	24	8	1.00	1.00
LinkedList¡:[v0]¿→Deque¡:[v0]¿	9	16	7	16	0	1.00	1.00
List¡:[v0]¿→ImmutableList¡:[v0]¿	15	12	4	11	0	0.92	1.00
List¡:[v0]¿→LinkedList¡:[v0]¿	9	24	4	22	1	1.00	0.96
List¡:[v0]¿→Set¡:[v0]¿	50	91	37	83	6	1.00	0.98
Map¡:[v1],:[v0]¿→ConcurrentMap¡:[v1],:[v0]¿	7	16	8	15	1	1.00	1.00
Map¡String,String¿→Properties	2	10	4	9	0	1.00	0.90
Optional¡Integer¿→OptionalInt	45	10	2	10	0	1.00	1.00
Queue¡:[v0]¿→Deque¡:[v0]¿	3	17	7	14	3	1.00	1.00
Queue¡:[v0]¿→BlockingQueue¡:[v0]¿	2	13	5	11	0	1.00	0.85
Random→SecureRandom	19	21	3	21	0	1.00	1.00
Stack¡:[v0]¿→Deque¡:[v0]¿	3	32	17	32	0	1.00	1.00
AtomicInteger→LongAdder	23	124	17	124	0	1.00	1.00
AtomicLong→AtomicInteger	2	11	3	6	5	1.00	1.00
AtomicLong→LongAdder	186	1026	22	1025	1	1.00	1.00
Function¡:[v0],Boolean¿→Predicate¡:[v0]¿	14	11	3	11	0	1.00	1.00
Function¡:[v0],Integer¿→ToIntFunction¡:[v0]¿	18	22	5	21	1	1.00	1.00
Supplier¡Integer¿→IntSupplier	8	15	2	15	0	1.00	1.00
long→BigInteger	17	4	2	4	0	1.00	1.00
long→Duration	10	15	4	14	1	1.00	1.00
long→Instant	7	13	13	13	0	1.00	1.00
long→AtomicLong	3	9	4	8	1	1.00	1.00
GetMethod→HttpGet	15	45	7	40	5	1.00	1.00
Log→Logger	424	300	6	295	5	1.00	1.00
ChannelBuffer→ByteBuf	39	59	12	32	15	1.00	0.93
DateTime→ZonedDateTime	283	256	25	249	3	1.00	0.98
TemporaryFolder→File	9	34	2	14	8	1.00	0.54
CompositeSubscription→CompositeDisposable	9	33	10	28	5	1.00	1.00

n: Number of type change instances **A**: Number of type dependent idioms
UR: Number of unique rewrite rules applied **TP**: True Positives
NA: Not Applicable **P**: Precision **R**: Recall

5.4.5 Did developers find the RewriteRules useful?

To answer this question, we perform type changes recommended by *Effective-Java* using IntelliT2R, in four large open source projects - Apache/Flink, ElasticSearch, IntelliJ-Community and Cassandra. In particular we perform type changes that eliminate the misuse of Java 8's Functional Interface API (e.g., `Supplier<Long>`→`LongSupplier`) and Optional API(e.g., `Optional<Integer>`→`OptionalInt`). We obtain the required specifications for eliminating these misuses from the rewrite rules collected in RQ1 (Section 5.4.2). Finding any missed opportunity to specialize interfaces in such projects, is an important contribution because it eliminates boxing (un-boxing); improving the performance.

IntelliT2R performed 98 type changes to eliminate the misuses of the Java 8 interfaces, in the the four open source projects. These type changes that updated 46 source code files and affected 213 SLOC. After IntelliT2R applied the type changes in each project, we built it to ensure that the source code compiled successfully and all test cases passed. For two type changes, we had to manually perform edits to update the signature of overriding methods (our current implementation limitation). Next, we sent out these type changes as pull requests to the maintainers of the projects. At the time of the writing the dissertation, one out of the four pull requests was accepted, while others were under review.

5.4.5.1 IntelliT2R in action

Since we do not have any control over the tasks that our developers perform, it is cardinal for the plugin to be as user-friendly and intuitive as possible. Therefore, IntelliT2R assists the developer at performing the type changes in four modes:

1. **Inspection Mode:** In this mode IntelliT2R scans the user's code in the opened editor on the fly and looks for potential opportunities for performing type changes. Such *code inspections*[5] particularly looks for the possible misuses of the functional interface API (e.g., `Function<Integer, T>`→`IntFunction<T>`) and Optional API (e.g., `Optional<Integer>`→`OptionalInt`). Moreover, we also create heuristics that promote the adaptation of conceptual types [27] (e.g., `String`→`Path`).

[5]https://www.jetbrains.com/help/idea/code-inspection.html

2. **Classic Mode** The user can invoke the plugin from the context menu at the type of some code element (i.e., local variable, field, parameter or method) as an *intention action*[6] and choose one of the offered type changes (e.g., `String→Optional<String>)`. The current stock type migration refactoring tool in IntelliJ IDEA operates in the similar manner.

3. **Suggestion Mode:** Previous researchers [53, 57] observed that *discoverability* and *late awareness* led to the under use of refactoring tools. To counter this problem, we leverage IntelliJ's *Suggested Refactoring* interface[7]. It includes a *clickable icon* in the IDE's gutter panel which appears when the refactoring can be potentially applied and the corresponding *intention action* in the context menu. The plugin tracks when the developer starts and finishes updating the type of some code element (e.g., variable or field) and suggests automating the type change (if there exists the appropriate rule for this particular case).

4. **Quick Fix Mode:** To perform the type changes as carefully as possible, IntelliT2R would not automatically apply the rewrite rules that changes the return type of the considered expression (e.g. `:[v].mkdir()→Files.createDirectory(:[v])`, which changes the return type of the expression from `boolean` to `Path`). It can cause other types changes that should be considered by the developer in a cascade manner. We decided to surface such rewrite rules at the references as a *quick fix* that adapts it to the type change (if IntelliT2R has relevant rules). This mode is basically the extension of the *Suggestion Mode*.

It is also worth mentioning that IntelliT2R handles all the rewrite rules failed to update their corresponding code elements, showing them in an additional *tool window* after performing the refactoring. The developer can fix them manually or *undo* the whole refactoring as an atomic action in the IDE. We also provided the user with settings of a *Search Scope* of the type change refactoring (e.g., local, current file, or global). It constraints the scope for the reference search of the built-in IntelliJ's Type Migration framework.

[6]https://www.jetbrains.com/help/idea/intention-actions.html
[7]https://www.jetbrains.com/help/idea/rename-refactorings.html#inplace_rename

5.4.5.2 Usage statistics

We collect detailed usage of our tool while respecting the privacy of the user. For each type change performed using IntelliT2R, we record the changed types (e.g., `String`→`Path` or `:[t]`→`Optional<:[t]>`), a kind of the code element where the plugin was invoked (i.e., local variable, field, parameter or method), and the numbers of unique rewrite rules applied, references where a rewrite rule was successfully applied, and references where no rewrite rule have matched. We also record the mode (1)—(4) in which the the plugin was used and if the user performed an *undo* action to cancel the refactoring suggestion provided by IntelliT2R.

5.4.5.3 Results

When this draft was written, we did not have received permission from Jetbrains's due diligence team to collect any telemetry data. Getting this approval required more time than we earlier estimated. Our plugin has been developed, thoroughly tested and is available for use[8]. We expect to collect some telemetry data, until the final submission of the dissertation.

5.5 Discussion

5.5.1 Type Migration and API Migration

Recently many researchers [83, 156] have proposed advanced techniques that automate library updates and library migrations. At a higher level, all these techniques infer and perform adaptations required for possibly many-to-many type changes between types in the source library and types in the target library. They apply advanced tools like data-flow analysis or control flow analysis in the context of the usage and and leverage rich information like the library source code, jar or documentation to infer the adaptations and encode the transformation in complicated graph models. Since they require such rich information, their applicability is restricted to a few libraries. These adaptations very often comprise of performing simple one-to-one type changes, which the researchers[24, 83] categorize as *hard to automate*, and provide some naive automation by matching the

[8]https://github.com/JetBrains-Research/data-driven-type-migration

control-flow graphs.

On the other hand, the type migration techniques are more general and they perform a very large variety of one-to-one or one-to-many type changes based on input rewrite rules. However, rewrite rules are limiting because they can only use the information from the matched source code, and not from the context of the match (like ancestor or sibling AST nodes). Because of the lack of expressiveness of the rewrite rules, our technique could not infer any adaptations that required additional context (as discussed in Item 1) and additional knowledge about the types (Item 2).

Our results (Item 1 and Item 2) show that type migration techniques could clearly benefit by adopting some ideas from the the API migration techniques to capture additional context and knowledge. Moreover, *TC-Infer* can effectively infer rewrite rules from type change instances (i.e. examples of type changes), therefore the next generation of API migration techniques can add type migration technique to their tool kits.

5.5.2 Inferring the Preconditions

Balaban et al. [13] have extensively discussed about the basic preconditions that need to be satisfied to perform a type change safely. These preconditions ensure that the code after the transformation is type correct (i.e. the code compiles). Often just making changes for the sake of satisfying the type checker, may lead to unexpected behaviour of the program after the change. For instance, Dig et al. [36] proposed additional pre-conditions required to perform type changes that introduce concurrency (`HashMap`→`ConcurrentHashMap`). Unless the source and target type of the type change are completely isomorphic (e.g. `Optional<Integer>`→`OptionalInt` or `Vector`→`ArrayList`), each type change would require an additional custom preconditions to ensure that the type change is safe. For instance, to perform the type change `String` → `Path`, the string value assigned to the variable should be representing the concept of path (like `'/Users/foo/bar'`) or else it will throw a runtime exception. Similarly, the type change `List`→`Set` can be performed only if the program is independent of the order of the elements, else the transformation will not preserve the behavior of the program. While TC-Infer effectively infers the rewrite rules for adapting the common syntactic idioms, it cannot infer such preconditions for applying the rules. Moreover, these preconditions could also be used as a heuristic for offering the developer type

change suggestions proactively. We believe this is a very challenging problem that could be addressed by capturing more context (source code around the updated code, other changes performed in the commit) and performing deeper analysis (possibly using the dynamic traces).

5.5.3 Using InferRules for other purposes

TC-Infer is a technique that deduces the rewrite rules applied for performing the identified type changes in the commit. At the heart of it is InferRules, an algorithm that returns rewrite rules applied across the two input code snippets (valid ASTs). In TC-Infer we apply InferRules only upon the updated statements that refer to the element whose type is changed and then only keeps the rewrite rules that are relevant to the type change. However, InferRules is *general* and could also be used for deducing the rules from examples of code snippets for other purposes where rewrite rules are applicable. For instance, researchers [144] suggested an technique that modifies a program slightly to reduce the false positives produced by static analysis tools, where they manually crafted the rewrite rules as Comby templates. Similarly researchers Nita and Notkin [111] utilized rewrite rules for API migration and Rolim et al. [121] utilized rewrite rules for fixing bugs. Moreover, our representation of AST is language agnostic and so is our entire technique (note that Comby is also a multi-language syntax transformation technique). The only modules in the technique that is not language agnostic is, GetTemplateFor - that converts snippets to structural templates (Definition 5.3.5) and the special kinds of template variables that we consider (Definition 5.3.3). Developing such modules for another language is not difficult, rather previous researchers [87] have leveraged this idea to highlight that their tool that recommends code completion snippets is language agnostic. We believe that InferRules could be used more generally to deduce Comby templates capturing the edits applied across the input code snippet.

5.6 Summary

Type change is a crucial activity in evolving code bases. While performing type changes manually is extremely tedious, adopting the current state-of-the-art techniques for such changes is not easy. This chapter solves an important problem that introduces a barrier

to the adoption of type change techniques, by proposing TC-Infer to automatically infer the rewrite rules required to perform the type change, which otherwise have to handcrafted. We evaluate the TC-Infer's applicability for inferring rules for *popular* type changes, and show the effectiveness of these rules at performing 3060 instances of type changes previously performed by open-source developers. We further demonstrate the usefulness of the rules produced by TC-Infer by developing a plugin that recommends these rules to the user when performing a type change in the IDE.

Chapter 6: Putting it Together

6.1 Overview of the Dissertation

This chapter puts together the three main contributions of this dissertation - 1. TypeFactMiner - a mining technique that can identify type changes applied in the version commit histories of the given project (Chapter 3) 2. TC-Infer - a technique that the transformation specifications required to perform a type change, by analysing previously applied type changes (Chapter 5) 3. T2R - a three-step MapReduce amenable framework that can perform type changes in ultra-large scale codebases.(Chapter 4) The flowchart in Figure 6.1 showcases the interplay amongst these three main contributions of the dissertation.

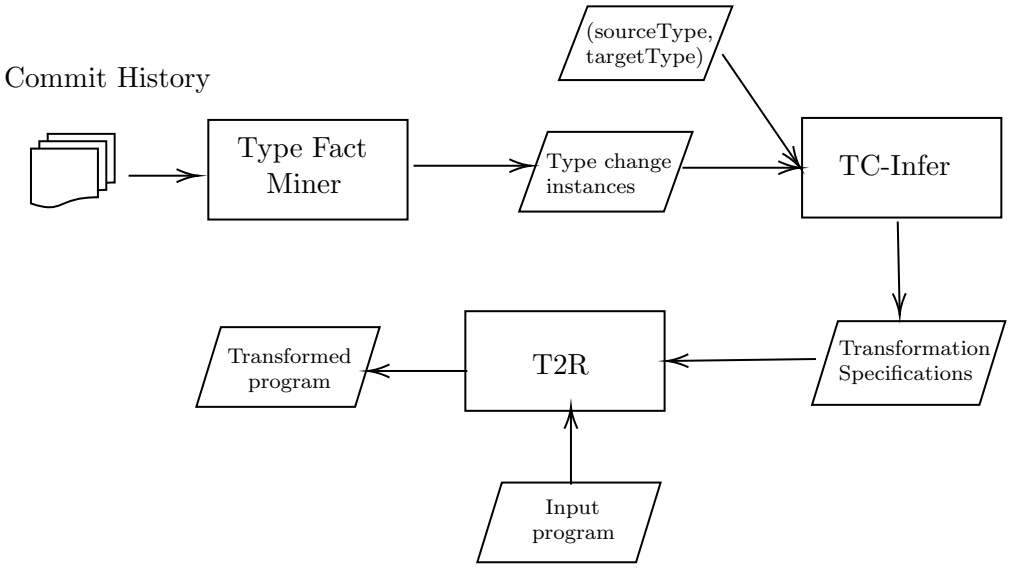


Figure 6.1: Overall architecture of the dissertation

In Chapter 3 we thoroughly describe our formative study that investigates the practice of performing type changes in open source Java projects through six novel questions. To answer these six research questions, we developed TypeFactMiner that efficiently

and effectively detect type changes from a corpus of 416,652 commits from 139 projects. We manually validated the tools and show that they have high precision (99.7%) and high recall (94%). For the input commit history, TypeFactMiner reports all the type change instances performed. Each type change instance is comprised of the updated element (i.e. variable, method or field) and all the matched statements within the variable's scope that refer to the variable/field/parameter in context.

The type change instances reported by TypeFactMiner, is the input to TC-Infer- the technique we propose in our next contribution as described in Chapter 5. The goal of TC-Infer is to infer rewrite rules for the adaptations applied to common syntactic idioms in previously performed type changes, and suggest these rules to the user when performing the same type change. These `TransformationSpec` (as described in Section 5.3.3) produced by TC-Infer captures the rewrite rules required to aadapt the source code to the type change. We utilize Comby's rewrite rules [145] to represent and perform the light-weight syntax transformations. In addition to the type change instance, TC-Infer also requires the user to provide the desired type change pattern (i.e. the source and target types) for which the transformation specifications should be produced. These rewrite rules can be losslessly translated to IntelliJ's structural replacement templates[1] or to the type migration DSLs proposed by Balaban et al. [13] and Ketkar et al. [74].

Our final contribution is T2R, a technique that can perform type changes in ultra-large code bases on the input transformation specifications. While T2R is the scalable IDE-independent ultra-scalable version of the technique, IntelliT2R is its interactive IntelliJ IDE adaptation. These specifications encode the edit patterns that need to be applied to adapt the source code the constraints of the new type. While users can manually encode these transformation specifications for the desired type changes, it is not straightforward, especially when the users are not knowledgeable about the involved types. To reduce this burden, users can use TC-Infer to infer `TransformationSpec` for the type change they desire from the version histories of relevant open source projects. T2R can accept these `TransformationSpec` as input and perform type changes in the input program based on these specifications.

[1] `https://www.jetbrains.com/help/idea/structural-search-and-replace.html`

6.2 Limitations

While we have developed an array of tools and techniques in this dissertation with an ultimate goal of automating type changes, these techniques have their own pitfalls and shortcomings. While some of these limitations are minor implementation problems that can be overcome with more engineering effort, others are fundamental in nature.

6.2.1 Limitations of TC-Infer

6.2.1.1 Misses Preconditions

Balaban et al. [13] laid out the basic preconditions for safely performing a type change involving inter-changeable types (e.g.`Vector`→`ArrayList`). However, they are not always enough; Dig et al. [36] proposed additional preconditions to safely update `HashMap` to `ConcurrentHashMap`. Similarly, the type change `List`→`Set` can be performed only if the program is independent of the order of the elements. While TC-Infer effectively infers the rewrite rules for adapting the common syntactic idioms, it does not infer preconditions for applying the rules. We believe this is a very challenging problem that could be addressed by capturing more context and analyzing dynamic traces.

6.2.1.2 Version Unaware

For safely suggesting and applying type changes in the real-world, the rewrite rules should be version specific, since types themselves evolve over time (API evoluton). This limitation can be easily overcome by analyzing the build system configuration files (like `pom.xml` and `build.gradle`) to identify the required version of Java and other third party libraries.

6.2.1.3 Language Specific

Currently TC-Infer is restricted to the Java language, however theoretically it is language agnostic (note that Comby is also a multi-language syntax transformation technique). Our representation of AST is generic and core algorithms (note that Comby is also a multi-language syntax transformation technique) operate upon these ASTs.

The only language dependent modules are- 1. RefactoringMiner, 2. GetTemplateFor(Definition 5.3.5).. While developing the later module for another language is straightforward; language-agnostic refactoring detection is very challenging, because each language has its own nuances to account for. For instance, recently researchers **?**] reimplemented RefactoringMiner in Python to support the Python language accounting for its dynamic nature. While Dilhara [37] proposed a technique that *Java-fies* Python program and enables JAva based AST analysis tools to work for python.

6.2.2 Limitations of T2R

6.2.2.1 Exception handling

T2R currently does not handle type migrations in which source and target types' methods throw different exception types. Such cases require complex changes to be propagated to the `throws` clauses, `throw` statements, and `try-catch` blocks.

6.2.2.2 Distributed graph analysis

While the collection and refactoring phases have been implemented with MapReduce processing, the analysis phase uses parallelism features of the Java Stream API. However, it could also be implemented in a MapReduce style using distributed graph libraries, e.g., Pregel [88] or Apache Giraph [6].

6.2.2.3 Java Reflection and Dependency Injection

Our current implementation does not handle reflection or dependency injection. This is a well-known limitation of mature, production-ready refactoring tools. A workaround would be to collect associated annotations such as `@inject`, `@provide`, or `@autowire`, and introduce a precondition for checking these annotations.

6.2.2.4 Declaring Transformation Specifications

Our experiments revealed that the Refactoring Instructions (as shown in Table 4.3) are expressive enough for a wide range of refactorings. However, one needs to manually

verify the completeness and correctness of the mappings.

6.2.3 Limitations of TypeFactMiner

6.2.3.1 Missing Context:

Currently TypeFactMiner reports type change instances that consist of the updated program element (i.e.e variable, field, method) and all the statements that refer to the element within the variable's scope (and the return statement in case of methods). However, these type change instances lack information from two aspects - 1. These matched statements could be a subset of all the statements that were actually adapted to perform the type change. 2. The matched statements reported could be incomplete. For instance, these reported matched statements do not contain the declaration of each identifier(variable or method name) that is used. Identifying all adapted statements would require additional type-binding information and call-graph analysis, but RefactoringMiner (or TypeFactMiner) works purely on syntax. While the former limitation can be overcome by providing RefactoringMiner with rich type-binding and def-use information from the compiler, this will make RefactoringMiner slow. One possible option, is to construct a bi-directional graph between these matched statements based on the *Type-Fact-Graph* relationships proposed in Chapter 4.

6.2.3.2 Language Specificity:

TypeFactMiner currently supports only Java programs. Extending the tool to support other languages has challenges that go beyond a simple engineering task. First, there are language-specific refactoring types that need to be accounted for, which are not applicable in other programming paradigms, or even languages from the same paradigm. For example, Go is an object-oriented language that uses struct embedding (i.e., type composition) instead of inheritance, making all inheritance-related refactorings inapplicable. Second, the type system used by a language affects the source code matching process. For example, in dynamically typed languages, function signatures consist only of the function name and number of arguments, thus making signature-based matching more ambiguous than statically typed languages. The matching of variable declaration state-

ments is also more ambiguous in dynamically typed languages, because we can rely only on variable identifiers and optional initializer expressions to match variable declarations, as variable types are not available in the source code.

Chapter 7: Related Work

We have successfully developed techniques for making the type changes automation techniques *scalable* and *usable*. This chapter situates our work in the larger context. Section 7.1 discusses about the empirical studies conducted by previous researchers about type changes but from the vantage point of higher level maintenance and evolution tasks such as API update and library migrations. Section 7.2 discusses about the approaches developed by previous researchers developed techniques for extracting change patterns from *examples* and assist developers at performing similar changes. Section 7.3 discusses about the techniques developed by previous researchers to perform type-related changes.

7.1 Empirical Studies on Type Changes

Previous work has studied type changes from the vantage point of higher-level maintenance and evolution tasks, such as API update and library migration, for decades [20].

Dig and Johnson [32] discovered that the changes that break the APIs are not random, but they tend to fall in particular categories. The largest share of these changes could be considered *refactoring* if looked from the point of view of the component itself. Dig later uses refactorings to formalizes component evolution. Later, Cossette and Walker [24] performed retroactive study into the presence and nature of the incompatibilities between several versions of a set of Java-based libraries. For each, they perform a detailed, manual analysis to determine what the correct adaptations are to migrate from the older to the newer version and investigate if any of the recommender techniques could correctly perform the adaptation. However, contrary to Dig they show that a small percent of API breaking changes can be addressed by then state-of-the-art-techniques. They categorize the adaptations as *automatable*, *partially-automatable* and *hard-to-automate* breaking changes. Unsurprizingly, they categorize type changes as *hard-to-automate*.

Previous researchers also investigated into API updates at the byte code level. Dig and Johnson [33] proposed *Reba* a refactoring-aware binary adaptation tool, that generates a compatibility layer for a library that allows both the old and the new interface to

be used in the same system. While Dietrich et al. [31] studied the risk of introducing binaries incompatibilities (the program is compiles but throws runtime failure) due to unsafe deployment of the evolved version of a library. This is particularly interesting because Java compiler and virtual machine use different rule sets to enforce contracts between the providers and consumers of an API.

Recently Al-Rubaye [1] performed a large scale empirical study over 58000 projects and 9 popular API migrations, and discuss the impact of the migrations on the overall quality of the code. Thier results show that the migrations significantly reduce coupling, increase cohesion and improve code readability. On the other hand, Kula et al. [82] performed a large scale study over 4600 projects and 2700 library dependencies to investigate if developers update their third party library dependencies. Surprisingly they show that although many of these systems rely heavily on dependencies, 81.5% of the studied systems still keep their outdated dependencies. Further their study reveals that 69% of the developers are unaware of the updates and less likely to prioritize the update in the future.

Teyton et al. [134] studied the practice of library migration in the Java ecosystem to understand how frequently, when, and why they are performed. Researchers have also studied API evolution and adoption for specific domains, for instance McDonnell et al. [91] study the Android ecosystem, Hora et al. [62] study Pharo ecosystems, [69] study logging library migration. Researchers have also studied the practice of API evolution for dynamically typed languages like Python. Dilhara et al. [38] performed a large-scale quantitative and qualitative empirical study to shed light on how developers use machine learning libraries, and how this evolution affects their code, and highlighted the need for specialized automation to handle the evolution of ML libraries.

While all these studies investigate type changes in the context evolution tasks like API updates/library migrations, our study in Chapter 3 investigates the practice of performing type changes as a whole (including *Internal* and *Jdk* type changes), by answering six broad research questions. Our longitudinal study of a large corpus helps us gain a deep understanding of the current gaps in research and tooling for type changes.

7.2 Extracting Edit Patterns

7.2.1 Library Evolution

Numerous approaches have been developed to infer properties of APIs, intended to guide their use by developers [120]. Earlier Chow and Notkin [21] proposed a technique that allowed the users to hand craft adaptations to update the client code. Later Dig and Johnson [33], Henkel and Diwan [59] and other researchers [132, 155] proposed tools like *CatchUp!* and *MahaldoRef*, that record the changes applied by library developers and recommend the same adaptations to the clients, when they update the library version. Dagenais and Robillard [26] proposed *SemDiff* that models API usage in terms of method calls and supports adaptation of these calls based on data-depenedencies.

Later, Nguyen et al. [106] proposed LibSync an advanced technique that guides developers in adapting API usage code by learning complex API usage adaptation patterns from other clients that already migrated to a new library version. It leverages the graph-based object model to represent the control and data dependencies of the program. Wu et al. [152] proposed *Aura* a novel hybrid approach that combines call dependency and text similarity and identify change rules for API updates.

Recently Fazzini et al. [52] proposed APIMigrator that automatically migrates Android API usages within 15 android apps by leveraging how developers of other apps migrated corresponding API usages and validates the migrations through differential testing. Similarly, Al-Rubaye [1] in his PhD dissertation creates techniques to mine third party library migrations applied in open source projects, infers the advantages of performing a library migrations and finally recommends it to other users in similar contexts, while guaranteeing safety.

Researchers Xu et al. [156], Lamothe et al. [83] proposed techniques that are focused at inferring complicated edits relating to updating certain APIs. Their techniques mine library-migration related edits in thousands of open source repositories, and analyze the documentation and binaries of these libraries of the concerned libraries, to infer large and complicated edits related to certain API updates, and then suggest it to the users.

All these apply advanced tools like data-flow analysis or control flow analysis in the context of the usage and and leverage rich information like the library source code, jar or documentation to infer the adaptations and encode the transformation in complicated

graph models. Thus they are also restricted to a pre-defined list of libraries. These adaptations very often comprise of performing simple one-to-one type changes, which the researchers[24, 83] categorize as *hard to automate*, and provide some naive automation by matching the control-flow graphs. On the other hand, the our techniques can infer rewrite rule and suggest rewrite rules for performing a any type changes, irrespective of which library it belongs to and its type kind (primitive, object type or array). Though rewrite rules are less expressive, they are human-comprehensible. Since *TC-Infer* can effectively infer rewrite rules from type change instances (i.e. examples of type changes), therefore the next generation of API migration techniques can add type migration technique to their tool kits.

7.2.2 Systematic Code Changes

Systematic code changes are high level transformations with related edits at code level. Kim et al. [80] propose an approach to automatically discover and represent systematic changes as logic rules. This technique starts with pre-defined set of *change rules* and infers high level rules involving logical opertors like *foreach* and *where* to capture systematic rules like "move the `print` method in *each* `Document` subclass to its super class". Similarly tools like *Repertoire* [117], *Sydit*[92] and *LASE*[93] infer a context-aware edit script for finding potential locations and transforming them. While type changes can be viewed as systematic edits, to apply type changes no context is required because the locations that require adaptation can be simply determined by type constraints ([13, 74]). Moreover as shown in Chapter 3, the syntactic transformations required for adapting to type changes is so large, that the techniques that require predefined set of change rules are not practical.

Recently Rolim et al. [121] developed *Refazer* that automatically generates program transformation from input ourpur examples. It leverages state-of-the-art programming-by-example methodology using the following key components: (a) a novel domain-specific language (DSL) for describing program transformations, (b) domain-specific deductive algorithms for efficiently synthesizing transformations in the DSL, and (c) functions for ranking the synthesized transformations. Also Long et al. [86] developed *Genesis* a system that infers code transforms for automatic patch generation systems . Given a set of successful human patches drawn from available revision histories, it generalizes

subsets of patches to infer transforms that together generate a productive search space of candidate patches. It then applies the inferred transforms to successfully patch bugs in previously unseen applications. Andersen and Lawall [5] propose a technique that generates generic patches from a set of files and their updated versions and applies these to other files. Yin et al. [157] propose a model that combines neural encoder with edit encoder, to express salient information of an edit and can be used to apply it. Similarly, other researchers [16, 96, 98] also apply program synthesis techniques infer program transformations. For instance, Miltner et al. [98] propose a modelss system BluePencil, that learns general purpose repetitive-edits on-the-fly in an IDE. It observes users as they make changes to the document, silently identifies repetitive changes, and automatically suggests transformations that can apply at other locations. Mesbah et al. [96] propose *DeepDelta* a technique that learns the edit patterns applied for repairing code that does not compile. They first encode the AST Diff in their domain specific language *Delta*, then they train a Neural Machine translation network with compiler diagnostics as input and the change Deltas as the desired output. The core of TC-Infer proposed in Chapter 5 has a similar goal as these techniques - inferring the program transformation from input-output examples. The difference is that, TC-Infer infers these edits as *rewrite rules* while these tools infer it in their own DSLs. While rewrite rules are not as expressive as the DSL programs these techniques produce, they are human-comprehensible making it easy to use, reason about and maintain.

7.3 Performing type-related changes

Previous work has proposed approaches for type-level refactorings, e.g., for facilitating class library migration by adaptors [14] or intermediate compatibility layers [35], generating new libraries for a constrained environment and migrating to them [148], migration based on manually-defined annotations [21] or capture-and-replay of changes [59], suggesting types in the migrated library based on existing examples and applying them [153], and changing references and method invocations after library migration using an Eclipse plugin [70].

Among others, a more promising approach for type migration is based on type constraints [13, 136, 137, 138], In this approach, a *migration specification* defines how legacy classes are mapped to use their replacement classes. Then they use an analysis based

on type constraints to determine where, for a given migration specification, it is possible to migrate uses of legacy classes without affecting the program's type-correctness or behavior. Moreover, they perform additional escape analysis to facilitate the replacement of synchronized legacy class with an unsynchronized new class and insert a synchronous wrapper around the code where such a migration is not feasible. Similarly, Khatchadourian [77] proposed a type checking technique to migrate legacy Java codebases to use enumerated types, designed to effectively handle primitive types since type constraints were excessive for handling primitives.

Despite the abundance, previous techniques have not been designed to scale to ultra-large-scale codebases. They usually depend on IDEs, which limits their applicability: the size and complexity of ultra-large-scale codebases do not allow applying sophisticated whole-program analysis required for type migration in any IDE running on machines with limited resources. In contrast, we have proposed a graph representation of type dependencies in the source code, essentially capturing *type constraints*, yet making the analysis highly-scalable through a distributed MapReduce approach. We showed that our approach can analyze a 300M LOC code base for type migration in 33 minutes, while existing approaches were evaluated on much smaller codebases, e.g., 272K LOC for a recent type-checking approach [77].

ClangMR [150] is a related MapReduce-based system for implementing and executing refactorings in large-scale codebases. ClangMR is limited to a single-step analysis of one compilation unit (e.g., a single source file) and is not able to propagate analysis results and code changes across multiple files. In contrast, our approach supports multi-step refactorings that require analyzing and propagating changes across the whole program; this is a primary contribution of this work. We have built the Google-variant of T2R on top of the Java equivalent of ClangMR at Google.

Chapter 8: Conclusion

8.1 Summary

The data types that are used in an application have a significant impact upon its quality. While one might argue that a developer should avoid performing type changes, that is never the case. In an actively-developer project, software engineers are always looking out for opportunities that improve performance and eliminate technical debt, or adapting to new requirements, often requiring to perform type changes.

Type changes is a very large and complex topic, and we believe no single tool or technique can handle all its aspects. For this dissertation we have conducted several investigations - first one explore the practice of type changes in the open source Java community and the next ones build and evaluate a tool-set for automating type changes.

We scrutinized the commit history of 130 open source Java projects to answer six research questions that explore various aspects of type changes like their frequency, characteristics, the relationship between the types and the edit patterns applied to adapt the source code to the type change. Among others, we found that type changes are actually more common than renamings, but the current research and tools for type changes are inadequate. Moreover, we found that 2% of the most popular type changes account for 43% of all the instances, highlighting an opportunity to learn to perform type changes from how other open source developers have performed it previously,

Performing type changes manually is very tedious and complex. As the size of a codebase increases, both the need for and the complexity of type changes increases. We improve over the state-of-the-art type change techniques, and present a T2R that uses a graph to model the *type constraints* and performs type changes *safely*, *accurately* and *efficiently* in code bases as large as Google's Java code base containing 300M LOC. The changes produced by T2R were accepted by Google developers and by open source developers maintaining state-of-the-art projects like *Cassandra* and *CoreNLP*.

The Achilles heel of these techniques is that the user has to manually encode the syntactic transformations required to perform the desired type changes. For this purpose,

we developed TC-Infer, that learns the task of performing type changes by analyzing several examples of how other open source developers have performed the same type change previously. We not only show the applicability and effectiveness of the rewrite rules produced by TC-Infer, but we also demonstrate the usefulness of these rewrite rules by extending IntelliJ's Type migration to use the transformation specification produced by TC-Infer.

8.2 Future Work

Although we have developed a technique to mine and infer the rewrite rules for type changes from commit histories and developed tools that automate type changes by applying these rewrite rules, we do not *recommend* type changes (i.e. automate the task of finding opportunities to perform type changes) beyond a few misuses like eliminating functional interfaces or the optional API. Previously, Dig et al. [36] developed a specialized type change technique that recommends new opportunity to use *concurrent* data structures like `ConcurrentMap`, and Khatchadourian [77] developed a specialized type change technique that recommends opportunities to use the *Enum* type kind. In Chapter 3 we identified hundreds of popular type changes performed in the open source projects and in Chapter 5 we infer rewrite rules for these. Below are the common trends of type changes that we believe could be recommended to the user:

1. **Optimal data structures** The most common trend of performing type changes involves changing the data structures (e.g. `List`→`Set`, or `ArrayList`→`LinkedList`). Understanding the motivation behind updating these data structures and developing a technique to appropriately suggest more optimal data structures based on the context, is a very promising direction to explore. Moreover, other than Java's own stock data structures, popular third party libraries like `guava` and `io.vavr` provide a plethora of useful data structures that could also be recommended. We believe just like we found un-optimized of functional interfaces API(Chapter 2), un-optimized use of the java collections API is just waiting to be found.

2. **Conceptual Types** As the language and libraries involve, the maintainers introduce newer conceptual types (like `Path`, `URL` or `Regex`) to improve the readability and maintainability of the code. In our study we found several such popular type changes

where developers conceptualize a raw data type like `String` or `long` to types like `Path`, `URL`, `Date`, `BigInteger`, `Security.Key` or `InetAdress`. Suggesting such type changes is very challenging and it would require analysing names of variables and methods, or analyzing the dynamic trace of the program.

3. **Monad-like Types** Over the last years, mainstream object-oriented languages like Java have adopted many features from functional programming languages like *lambda expressions*, *Optionals*, *Streams* or *CompletableFutures*. Adapting the existing programs to use these features manually is very complicated and tedious, because it requires the developer to reason about the control-flow and data-flow of a existing program and then substitute it appropriately with these new functional APIs. Recently Franklin et al. [54] proposed a technique that suggests basic refactorings from traditional `for-loops` to use the `Stream` API, Khatchadourian et al. [78] proposed an advanced technique to safe automated refactoring for parallelization of the Streams API and Tsantalis et al. [140] proposed a technique to use lambda expressions to eliminate code clones. While these techniques recommend use of functional programming features provided by Java, we believe this only scratches the surface. For instance, the `Optional` offers functions like `map`, `flatmap`, `filter`, `or` or `orElse`, that can often dissolve complex logic involving nested `if-else` and `assignments`, into a intuitive fluent API calls while providing *null safety*. Moreover, popular third party libraries like `io.vavr` provide advanced functional programming inspired features like `Try` (which is inspired from Scala's try monad), that could dissolve `try-catch` blocks by embedding error handling into normal program processing flow. Developing techniques to recommend such advanced and abstract types could be a promising avenue for future work, that could benefit both the users and maintainers of the language and other libraries.

Bibliography

[1] Hussein Ahmed Talib Al-Rubaye. *Towards the Automation of Migration and Safety of Third-Party Libraries*. Rochester Institute of Technology, 2020.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

[3] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, pages 60–71, Riverton, NJ, USA, 2018. IBM Corp. URL http://dl.acm.org/citation.cfm?id=3291291.3291299.

[4] Hussein Alrubaye, Deema AlShoaibi, Mohamed Wiem Mkaouer, and Ali Ouni. How Does API Migration Impact Software Quality and Comprehension? An Empirical Study. https://arxiv.org/abs/1907.07724, Jul 2019. URL https://arxiv.org/abs/1907.07724.

[5] J. Andersen and J. L. Lawall. Generic patch inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 337–346, USA, 2008. IEEE Computer Society. ISBN 9781424421879. doi: 10.1109/ASE.2008.44. URL https://doi.org/10.1109/ASE.2008.44.

[6] Apache. Apache giraph. URL {http://giraph.apache.org/}. Accessed: 2018-06-07.

[7] Apache. Cassandra, 2018. URL http://cassandra.apache.org/. Accessed: 23 August 2018.

[8] Apache. effective-pom, 2019. URL {https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html}.

[9] Apache. Netbeans refactoring, 2019. URL {http://wiki.netbeans.org/Refactoring}.

[10] Apache. Visual studio-refactor code, 2019. URL {https://docs.microsoft.com/en-us/visualstudio/ide/refactoring-in-visual-studio?view=vs-2019}.

[11] Venera Arnaoudova, Laleh M. Eshkevari, Massimiliano Di Penta, Rocco Oliveto, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Repent: Analyzing the nature of identifier renamings. *Software Engineering, IEEE Transactions on*, 40:502–532, 05 2014. doi: 10.1109/TSE.2014.2312942.

[12] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360585. URL https://doi.org/10.1145/3360585.

[13] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 265–279, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094832. URL http://doi.acm.org/10.1145/1094811.1094832.

[14] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5610429. URL https://doi.org/10.1109/ICSM.2010.5610429.

[15] Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. https://arxiv.org/abs/1809.05193, Aug 2018. URL https://arxiv.org/abs/1809.05193.

[16] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 613–624, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3338952. URL http://doi.acm.org/10.1145/3338906.3338952.

[17] Marat Boshernitsan, Susan L. Graham, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 567–576, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240715. URL http://doi.acm.org/10.1145/1240624.1240715.

[18] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 465–475, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106259. URL http://doi.acm.org/10.1145/3106237.3106259.

[19] Alexander Chávez, Isabella Ferreira, Eduardo Fernandes, Diego Cedrim, and Alessandro Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, SBES'17, pages 74–83, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5326-7. doi: 10.1145/3131151.3131171. URL http://doi.acm.org/10.1145/3131151.3131171.

[20] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance*, pages 359–368, Nov 1996. doi: 10.1109/ICSM.1996.565039.

[21] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*, pages 359–368, 1996. doi: 10.1109/ICSM.1996.565039. URL https://doi.org/10.1109/ICSM.1996.565039.

[22] Guacamole Client. commit with type change, 2011. URL https://tinyurl.com/yx2npj8g.

[23] Bradley Cossette and Robert J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT-/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, pages 55:1–55:11, 2012. doi: 10.1145/2393596.2393661. URL http://doi.acm.org/10.1145/2393596.2393661.

[24] Bradley E. Cossette and Robert J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 55:1–55:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393661. URL http://doi.acm.org/10.1145/2393596.2393661.

[25] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582153. doi: 10.1145/1449764.1449790. URL https://doi.org/10.1145/1449764.1449790.

[26] Barthélémy Dagenais and Martin P. Robillard. Semdiff: Analysis and recommendation support for api evolution. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 599–602, May 2009. doi: 10.1109/ICSE.2009.5070565.

[27] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. Refinym: Using names to refine types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 107–117, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5573-5. doi: 10.1145/3236024.3236042. URL http://doi.acm.org/10.1145/3236024.3236042.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce. *Communications of the ACM*, 51: 107–113, 2008. doi: 10.1145/1327452.1327492. URL https://doi.org/10.1145/1327452.1327492.

[29] Eclipse Java development tools. Jdt components, 2017. URL http://www.eclipse.org/jdt/.

[30] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73, Feb 2014. doi: 10.1109/CSMR-WCRE.2014.6747226.

[31] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, pages 64–73, Washington, DC, USA, Feb 2014. IEEE Computer Society. doi: 10.1109/CSMR-WCRE.2014.6747226.

[32] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, March 2006. ISSN 1532-060X. doi: 10.1002/smr.328. URL https://doi.org/10.1002/smr.328.

[33] Danny Dig and Ralph Johnson. Automated upgrading of component-based applications. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 675–676, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593491X. doi: 10.1145/1176617.1176668. URL https://doi.org/10.1145/1176617.1176668.

[34] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 404–428, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35726-2, 978-3-540-35726-1. doi: 10.1007/11785477_24. URL http://dx.doi.org/10.1007/11785477_24.

[35] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph E. Johnson. *ReBA*: *re*factoring-aware *b*inary *a*daptation of evolving libraries. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 441–450, 2008. doi: 10.1145/1368088.1368148. URL http://doi.acm.org/10.1145/1368088.1368148.

[36] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 397–407, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: 10.1109/ICSE.2009.5070539. URL http://dx.doi.org/10.1109/ICSE.2009.5070539.

[37] Malinda Dilhara. Discovering repetitive code changes in ml systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1683–1685, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3473493. URL https://doi.org/10.1145/3468264.3473493.

[38] Malinda Dilhara, Ameya Ketkar, and Danny Dig. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Trans. Softw. Eng. Methodol.*, 30(4), July 2021. ISSN 1049-331X. doi: 10.1145/3453478. URL https://doi.org/10.1145/3453478.

[39] Guava Documentation. Immutablelist, 2019. URL {https://guava.dev/releases/23.4-jre/api/docs/com/google/common/collect/ImmutableList.html}.

[40] Java Platform Documentation. Autoboxing and unboxing, 2019. URL {https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html}.

[41] Java Platform Documentation. Deque, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html}.

[42] Java Platform Documentation. Linkedlist, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html}.

[43] Java Platform Documentation. Map, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/util/Map.html}.

[44] Java Platform Documentation. Stringbuffer, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuffer.html}.

[45] Java Platform Documentation. Stringbuilder, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html}.

[46] Java Platform Documentation. List, 2019. URL {https://docs.oracle.com/javase/9/docs/api/java/util/List.html}.

[47] Java Platform Documentation. Securerandom, 2019. URL {https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html}.

[48] Java Platform Documentation. Set, 2019. URL {https://docs.oracle.com/javase/9/docs/api/java/util/Set.html}.

[49] Java Platform Documentation. Widening primitive conversion, 2019. URL {https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html\#jls-5.1.2}.

[50] Eclipse. Refactoring actions, 2019. URL {https://help.eclipse.org/kepler/index.jsp?topic=\%2Forg.eclipse.jdt.doc.user\%2Freference\%2Fref-menu-refactor.htm}.

[51] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3013-8. doi: 10.1145/2642937.2642982. URL http://doi.acm.org/10.1145/2642937.2642982.

[52] Mattia Fazzini, Qi Xin, and Alessandro Orso. Automated api-usage update for android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 204–215, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330571. URL http://doi.acm.org/10.1145/3293882.3330571.

[53] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 222–232, 2012. doi: 10.1109/ICSE.2012.6227191.

[54] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. LAMBDAFICATOR: From imperative to functional programming through automated refactoring. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1287–1290, 2013. ISBN 978-1-4673-3076-3.

[55] Robert M. Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, pages 71–96, 2005. doi: 10.1007/11531142_4. URL `https://doi.org/10.1007/11531142\_4`.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[57] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 211–221. IEEE Press, 2012. ISBN 9781467310673.

[58] Google. Error prone, 2011. URL `https://github.com/google/error-prone`.

[59] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*, pages 274–283, 2005. doi: 10.1145/1062455.1062512. URL `http://doi.acm.org/10.1145/1062455.1062512`.

[60] Mark Hills, Paul Klint, and Jurgen J. Vinju. Scripting a refactoring with rascal and eclipse. WRT '12, page 40–49, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315005. doi: 10.1145/2328876.2328882. URL `https://doi.org/10.1145/2328876.2328882`.

[61] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, April 2016. ISSN 0001-0782. doi: 10.1145/2902362. URL `https://doi.org/10.1145/2902362`.

[62] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to api evolution? the pharo

ecosystem case. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 251–260, Sep. 2015. doi: 10.1109/ICSM.2015.7332471.

[63] IntelliJ. Type migration refactoring. URL {https://www.jetbrains.com/help/idea/migrate.html}. Accessed: 2018-06-06.

[64] JavaParser. Java 9 parser and abstract syntax tree for java, 2017. URL http://javaparser.org/.

[65] JDP. Java-design-patterns, 2018. URL http://java-design-patterns.com/. Accessed: 23 August 2018.

[66] Eclipse JDT. Type, 2019. URL https://help.eclipse.org/luna/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Type.html.

[67] JetBrains. Type migration, 2019. URL {https://www.jetbrains.com/help/idea/type-migration.html}.

[68] JetBrains. Intellij - refactoring code, 2019. URL {https://www.jetbrains.com/help/idea/refactoring-source-code.html}.

[69] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. Logging library migrations: A case study for the apache software foundation projects. In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*, pages 154–164, May 2016. URL https://doi.ieeecomputersociety.org/10.1109/MSR.2016.025.

[70] Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring references for library migration. *ACM SIGPLAN Notices*, 45(10):726–738, 2010. doi: 10.1145/1932682.1869518. URL https://doi.org/10.1145/1932682.1869518.

[71] Ameya Ketkar. Companion website, 2019. URL https://ameyaketkar.github.io/T2RResults.html. Accessed: 14 February 2019.

[72] Ameya Ketkar. ameyaketkar/typechangeminer: Type change miner, June 2020. URL https://doi.org/10.5281/zenodo.3906493.

[73] Ameya Ketkar. Type change study data collected, June 2020. URL https://doi.org/10.5281/zenodo.3906503.

[74] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. Type migration in ultra-large-scale codebases. In *Proceedings of the*

41st International Conference on Software Engineering, ICSE '19, pages 1142–1153, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00117. URL https://doi.org/10.1109/ICSE.2019.00117.

[75] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Type facts companion website, 2019. URL {http://changetype.s3-website.us-east-2.amazonaws.com/docs/}.

[76] Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. *Understanding Type Changes in Java*, page 629–641. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450370431. URL https://doi.org/10.1145/3368089.3409725.

[77] Raffi Khatchadourian. Automated refactoring of legacy Java software to enumerated types. *Automated Software Engineering*, 24(4):757–787, December 2017. ISSN 0928-8910. doi: 10.1007/s10515-016-0208-8.

[78] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. Safe automated refactoring for intelligent parallelization of java 8 streams. *Science of Computer Programming*, 195:102476, 2020. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2020.102476. URL https://www.sciencedirect.com/science/article/pii/S0167642320300861.

[79] Jongwook Kim, Don Batory, and Danny Dig. Scripting parametric refactorings in java to retrofit design patterns. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 211–220, 2015. doi: 10.1109/ICSM.2015.7332467.

[80] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, Jan 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.16.

[81] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. doi: https://doi.org/10.1002/nav.3800020109. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[82] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Softw. Engg.*, 23(1):384–417, February 2018. ISSN 1382-3256. doi: 10.1007/s10664-017-9521-5. URL https://doi.org/10.1007/s10664-017-9521-5.

[83] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. A3: Assisting android api migrations using code examples. *IEEE Transactions on Software Engineering*, pages 1–1, 2020. doi: 10.1109/TSE.2020.2988396.

[84] Huiqing Li and Simon Thompson. A domain-specific language for scripting refactorings in erlang. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 501–515, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-28872-2.

[85] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. SWIN: Towards Type-Safe Java Program Adaptation Between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, PEPM '15, pages 91–102, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3297-2. doi: 10.1145/2678015.2682534. URL http://doi.acm.org/10.1145/2678015.2682534.

[86] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 727–739, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106253. URL http://doi.acm.org/10.1145/3106237.3106253.

[87] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360578. URL https://doi.org/10.1145/3360578.

[88] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184.

[89] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. *Proc. ACM Program. Lang.*, 1 (OOPSLA):85:1–85:31, October 2017. ISSN 2475-1421. doi: 10.1145/3133909. URL http://doi.acm.org/10.1145/3133909.

[90] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Supplemental material, 2017. URL http://dmazinanian.me/conference-papers/oopsla/2017/07/04/oopsla17.html.

[91] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 70–79, Sep. 2013. doi: 10.1109/ICSM.2013.18.

[92] Na Meng, Miryung Kim, and Kathryn S. Mckinley. Sydit: Creating and applying a program transformation from an example. In *in ESEC/FSE'11, 2011*, pages 440–443.

[93] Na Meng, Miryung Kim, and Kathryn S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 502–511, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486855`.

[94] T. Mens and T. Tourwe. A declarative evolution framework for object-oriented design patterns. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 570–579, 2001. doi: 10.1109/ICSM.2001.972774.

[95] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, pages 286–301, 2002. doi: 10.1007/3-540-45832-8_22. URL `https://doi.org/10.1007/3-540-45832-8\_22`.

[96] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 925–936, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5572-8. doi: 10.1145/3338906.3340455. URL `http://doi.acm.org/10.1145/3338906.3340455`.

[97] Microsoft. Visual Studio, 2021. At `https://www.visualstudio.com`.

[98] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360569. URL `https://doi.org/10.1145/3360569`.

[99] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, Jan 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.41.

[100] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642390371. doi: 10.1007/978-3-642-39038-8_23. URL https://doi.org/10.1007/978-3-642-39038-8_23.

[101] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568317. URL http://doi.acm.org/10.1145/2568225.2568317.

[102] Neo4J. Neo4j, 2018. URL https://neo4j.com/. Accessed: 23 August 2018.

[103] Netbeans. Netbeans - jackpot wiki, 2011. URL http://wiki.netbeans.org/Jackpot.

[104] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 511–522, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950333. URL http://doi.acm.org/10.1145/2950290.2950333.

[105] H. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830, Los Alamitos, CA, USA, may 2019. IEEE Computer Society. doi: 10.1109/ICSE.2019.00089. URL https://doi.ieeecomputersociety.org/10.1109/ICSE.2019.00089.

[106] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. *SIGPLAN Not.*, 45(10):302–321, October 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869486. URL http://doi.acm.org/10.1145/1932682.1869486.

[107] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, page 180–190. IEEE Press, 2013.

ISBN 9781479902156. doi: 10.1109/ASE.2013.6693078. URL https://doi.org/10.1109/ASE.2013.6693078.

[108] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 819–830, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICSE.2019.00089. URL https://doi.org/10.1109/ICSE.2019.00089.

[109] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. ISBN 3642084745, 9783642084744.

[110] Kazuki Nishizono, Shuji Morisaki, Rodrigo Vivanco, and Kenichi Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks - an empirical study with industry practitioners. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 473–481, Sep. 2011. doi: 10.1109/ICSM.2011.6080814.

[111] Marius Nita and David Notkin. Using twinning to adapt programs to alternative apis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 205–214, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806832. URL http://doi.acm.org/10.1145/1806799.1806832.

[112] Oracle. Autoboxing. URL {https://docs.oracle.com/javase/8/docs/technotes/guides/language/autoboxing.html}. Accessed: 2018-06-24.

[113] Oracle. Type inference, 2015. URL https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html.

[114] Presto. Prestodb, 2018. URL https://prestodb.io/. Accessed: 23 August 2018.

[115] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–10, Sep. 2010. doi: 10.1109/ICSM.2010.5609577.

[116] The Joda project. Joda-time, 2019. URL {https://www.joda.org/joda-time/}.

[117] Baishakhi Ray, Christopher Wiley, and Miryung Kim. Repertoire: A cross-system porting analysis tool for forked software projects. In *Proceedings of*

the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316149. doi: 10.1145/2393596.2393603. URL https://doi.org/10.1145/2393596.2393603.

[118] Juergen Rilling and Tuomas Klemola. Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 115–124, USA, 2003. IEEE Computer Society. ISBN 0769518834. doi: 10.1109/WPC.2003.1199195.

[119] Martin P. Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Softw. Engg.*, 16(6):703–732, December 2011. ISSN 1382-3256. doi: 10.1007/s10664-010-9150-8. URL https://doi.org/10.1007/s10664-010-9150-8.

[120] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, May 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.63.

[121] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 404–415, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.44. URL https://doi.org/10.1109/ICSE.2017.44.

[122] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. Learning quick fixes from code repositories. http://arxiv.org/abs/1803.03806, 2018. URL http://arxiv.org/abs/1803.03806.

[123] John Rose. Value objects, 2019. URL {https://openjdk.java.net/jeps/169}.

[124] Rubén Saborido, Rodrigo Morales, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Getting the most from map data structures in android. *Empirical Software Engineering*, 2018. doi: 10.1007/s10664-018-9607-8. URL https://doi.org/10.1007/s10664-018-9607-8.

[125] Danilo Silva and Marco Tulio Valente. Refdiff: Detecting refactorings in version histories. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*, pages 269–279, May 2017. doi: 10.1109/MSR.2017.14.

[126] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 858–870, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950305. URL http://doi.acm.org/10.1145/2950290.2950305.

[127] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, pages 1–17, 2020. ISSN 2326-3881. doi: 10.1109/TSE.2020.2968072.

[128] SonarQube. Sonarqube, 2018. URL https://www.sonarqube.org/. Accessed: 23 August 2018.

[129] Speedment. Speedment, 2018. URL https://www.speedment.com/. Accessed: 23 August 2018.

[130] Stanford. Stanford-corenlp, 2018. URL https://stanfordnlp.github.io/CoreNLP/. Accessed: 23 August 2018.

[131] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A refactoring constraint language and its application to eiffel. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 255–280, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22655-7.

[132] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 377–380, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321688. URL http://doi.acm.org/10.1145/1321631.1321688.

[133] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in Java. *J. Softw. Evol. Process*, 26(11):1030–1052, November 2014. ISSN 2047-7473. doi: 10.1002/smr.1660. URL http://dx.doi.org/10.1002/smr.1660.

[134] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *J. Softw. Evol. Process*, 26(11):1030–1052, November 2014. ISSN 2047-7473. doi: 10.1002/smr.1660. URL http://dx.doi.org/10.1002/smr.1660.

[135] Andreas Thies and Eric Bodden. Refaflex: Safer refactorings for reflective Java programs. In *Proceedings of the 2012 International Symposium on Software Testing*

and Analysis, ISSTA 2012, pages 1–11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1454-1. doi: 10.1145/2338965.2336754. URL http://doi.acm.org/10.1145/2338965.2336754.

[136] Frank Tip and Peter F. Sweeney. Class hierarchy specialization. *ACM SIGPLAN Notices*, 32(10):271–285, 1997. doi: 10.1145/263700.263748. URL https://doi.org/10.1145/263700.263748.

[137] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. *ACM SIGPLAN Notices*, 38(11):13–26, 2003. doi: 10.1145/949343.949308. URL https://doi.org/10.1145/949343.949308.

[138] Frank Tip, Robert M. Fuhrer, Adam Kieżun, Michael D. Ernst, Ittai Balaban, and Bjorn De Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, May 2011. ISSN 0164-0925. doi: 10.1145/1961204.1961205. URL http://doi.acm.org/10.1145/1961204.1961205.

[139] TouK. Throwingfunction, 2017. URL https://github.com/TouK/ThrowingFunction.

[140] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. Clone Refactoring with Lambda Expressions. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 60–70, 2017. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.14.

[141] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180206. URL http://doi.acm.org/10.1145/3180155.3180206.

[142] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, pages 1–21, 2020. doi: 10.1109/TSE.2020.3007722.

[143] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017. doi: 10.1002/smr.1838. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1838.

[144] Rijnard van Tonder and Claire Le Goues. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International*

Conference on Software Engineering, ICSE '20, page 824–834, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380343. URL https://doi.org/10.1145/3377811.3380343.

[145] Rijnard van Tonder and Claire Le Goues. Lightweight multi-language syntax transformation with parser parser combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 363–378, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314589. URL https://doi.org/10.1145/3314221.3314589.

[146] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated js names. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 683–693, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5105-8. doi: 10.1145/3106237.3106289. URL http://doi.acm.org/10.1145/3106237.3106289.

[147] Mathieu Verbaere, Arnaud Payement, and Oege de Moor. Scripting refactorings with jungl. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 651–652. ACM, 2006. doi: 10.1145/1176617.1176656. URL https://doi.org/10.1145/1176617.1176656.

[148] Victor L. Winter and Azamatbek Mametjanov. Generative programming techniques for Java library migration. In *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, Proceedings*, pages 185–196, 2007. doi: 10.1145/1289971.1290001. URL http://doi.acm.org/10.1145/1289971.1290001.

[149] Hyrum K. Wright. Incremental type migration using type algebra. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 756–765, 2020. doi: 10.1109/ICSME46990.2020.00085.

[150] Hyrum K. Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. Large-scale automated refactoring using clangmr. In *2013 IEEE International Conference on Software Maintenance (ICSM)*, pages 548–551, 2013. doi: 10.1109/icsm.2013.93. URL https://doi.org/10.1109/icsm.2013.93.

[151] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: A hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE

'10, pages 325–334, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806848. URL http://doi.acm.org/10.1145/1806799.1806848.

[152] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. AURA: a hybrid approach to identify framework evolution. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 325–334, 2010. doi: 10.1145/1806799.1806848. URL http://doi.acm.org/10.1145/1806799.1806848.

[153] Zhenchang Xing and E. Stroulia. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, 2007. doi: 10.1109/tse.2007.70747. URL https://doi.org/10.1109/tse.2007.70747.

[154] Zhenchang Xing and Eleni Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101919. URL http://doi.acm.org/10.1145/1101908.1101919.

[155] Zhenchang Xing and Eleni Stroulia. Api-evolution support with diff-catchup. *IEEE Transactions on Software Engineering*, 33(12):818–836, Dec 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70747.

[156] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of api migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 335–346, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/ICPC.2019.00052. URL https://doi.org/10.1109/ICPC.2019.00052.

[157] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander Gaunt. Learning to represent edits. In *ICLR 2019*, May 2019. URL https://www.microsoft.com/en-us/research/publication/learning-to-represent-edits/. arXiv:1810.13337 [cs.LG].

[158] Guangtun Zhu. A new view of classification in astronomy with the archetype technique: An astronomical case of the np-complete set cover problem, 2016.