Type Migration in Ultra-Large-Scale Codebases

Ameya Ketkar*, Ali Mesbah† ‡, Davood Mazinanian†, Danny Dig* and Edward Aftandilian‡
*Oregon State University – {ketkara, digd}@oregonstate.edu
†University of British Columbia – {amesbah, dmazinanian}@ece.ubc.ca
‡Google Inc. – eaftan@google.com

***Abstract*—Type migration is a refactoring activity in which an existing type is replaced with another one throughout the source code. Manually performing type migration is tedious as programmers need to find all instances of the type to be migrated, along with its dependencies that propagate over assignment operations, method hierarchies, and subtypes. Existing automated approaches for type migration are not adequate for ultra-large-codebases – they perform an intensive whole-program analysis that does not scale. If we could represent the type structure of the program as graphs, then we could employ a MAPREDUCE parallel and distributed process that scales to hundreds of millions of LOC. We implemented this approach as an IDE-independent tool called T2R, which integrates with most build systems. We evaluated T2R's accuracy, usefulness and scalability on seven open source projects and one proprietary codebase of 300M LOC. T2R generated 130 type migration patches, of which the original developers accepted 98%.**

***Index Terms*—Refactoring; Type Migration; MapReduce.**

I. 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 [1]. 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 [2], fixing API-breaking changes in clients (e.g., updating a method's signature [3]), or correcting inefficient uses of an API [4]. These refactorings are generally hard to automate [5]. 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 [6]–[13] 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 [7]–[9], [14]. However, type constraints analysis is resource intensive [7] 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 paper, 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 [15] 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 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 paper we use T2R to *specialize* the usage of Functional Interfaces across Java codebases, e.g., replacing `Function<Integer, Integer>` with

```
1  import java.util.function.Function;
2  interface LinearSearcher {
3    double minimize(Function<Double, Double> f);
4  }
5  class GoldenSectionLineSearcher implements LinearSearcher {
6    @Override double minimize(Function<Double, Double> f) {
7      double val = getValue();
8      return f.apply(val);
9    }
10 }
11 class SVMLightFactory {
12   LinearSearcher minimizer = new GoldenSectionLineSearcher()
13   public double heldOutC() {
14     Function<Double, Double> sq = x -> x*x;
15     return minimizer.minimize(sq);
16   }
17 }
```

(a) Before Type Migration

```
1  import java.util.function.DoubleUnaryOperator;
2  interface LinearSearcher {
3    double minimize(DoubleUnaryOperator f);
4  }
5  class GoldenSectionLineSearcher implements LinearSearcher {
6    @Override double minimize(DoubleUnaryOperator f) {
7      double val = getValue();
8      return f.applyAsDouble(val);
9    }
10 }
11 class SVMLightFactory {
12   LinearSearcher minimizer = new GoldenSectionLineSearcher();
13   public double heldOutC() {
14     DoubleUnaryOperator sq = x -> x*x;
15     return minimizer.minimize(sq);
16   }
17 }
```

(b) After Type Migration

Figure 1: Motivating Example

`IntUnaryOperator`, or `BiFunction<U,V,Boolean>` with `BiPredicate<U,V>`. We were inspired by our previous work [4] 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 [16]. 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 paper 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.

II. Motivating Example

We show the intricacies associated with type migration through a real-world example. Figure 1 shows a simplified view of a T2R-generated patch that we sent to the open-source project CoreNLP [17]. 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 [4] 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 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 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 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 expressions body, if there existed any (e.g., `x -> x.doubleValue()`).

Figure 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 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), 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.

III. 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 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

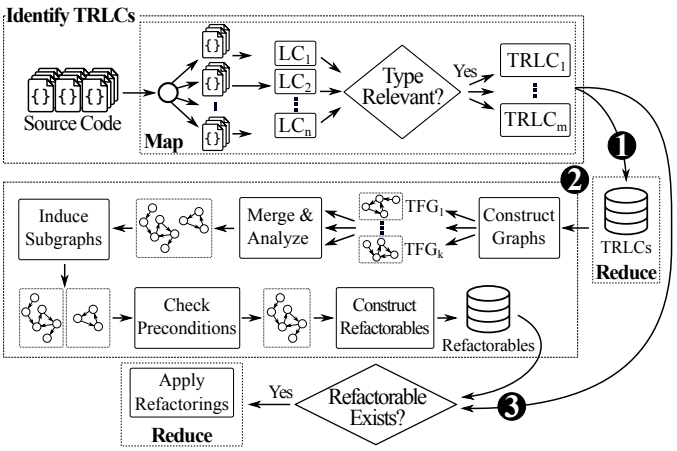

Figure 2: Approach Overview

code spans hundreds of millions of LOC. Each of the steps becomes amenable to distributed execution internally through, for example, MapReduce [15]. 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.

A. 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 1). This reduction allows us to scale to large codebases.

As shown in Figure 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 I 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 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 I, 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 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 3 shows a sample IDENTIFICATION constructed for the lambda expression `x -> x*x` declared in `SVMLightFactory.heldOutC()` (Figure 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

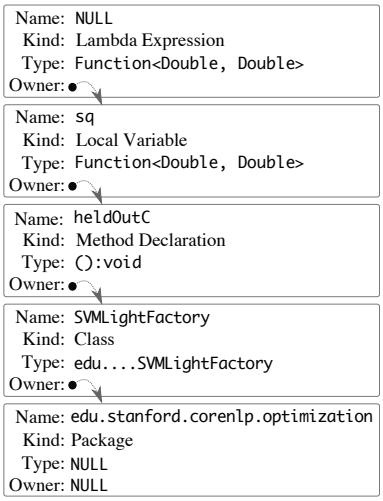

Figure 3: IDENTIFICATION for `x -> x*x` in `Function<Double,Double> sq = x -> x*x`

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

B. 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 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 methods 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`.

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 [18], and type constraints [7].

Table I: 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	

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 [9]. 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 II illustrates a list of such relationships and the corresponding edge labels.

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 key steps in this phase are depicted in Figure 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 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.

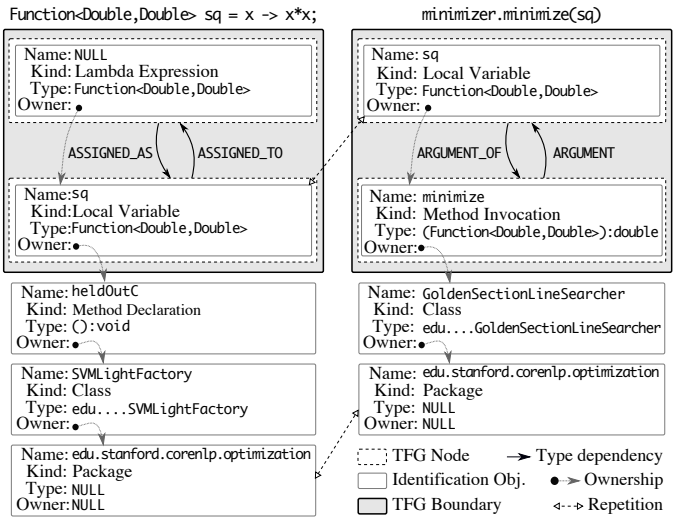

Figure 4: Individual TFGs for `SVMLightFactory`

Constructing the enriched TFG. Our approach then incrementally merges the individual TFGs among themselves. The *Merge & Analyze* block depicted in Figure 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-inferred ones. Figure 5 illustrates the example of applying this operation on the two independent TFGs shown in Figure 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 5 illustrate the result of this analysis and enrichment. Observe that the IDENTIFICATIONs of the declaration and invocation

Table II: 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

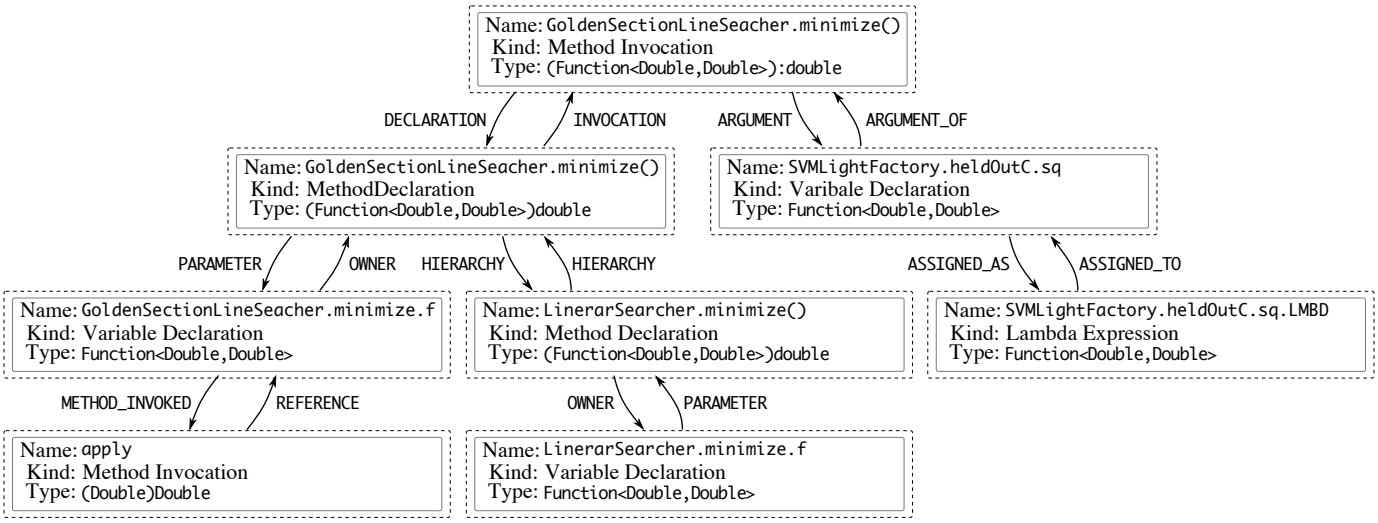

Figure 5: Unified TFG

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 graphs containing these nodes are filtered out by the preconditions.

In Figure 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`.

C. 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 5 shows one such disconnected subgraph obtained from the motivating example (Figure 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.

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

Table III: 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.

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 nodes 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 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.

2) Generating REFACTORABLE*s:* 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 III) 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 III, 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.

D. Applying Refactorings

The final phase is responsible for applying the detected refactorings to the source code. As illustrated in Figure 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.

IV. IMPLEMENTATION

We use JAVAC to parse the Java source code and implement AST node visitors to collect and refactor language

constructs. For TFG representation, we use Guava Graph library [19]. We construct object representations of TRLCs using Protocol Buffers [20], which is an efficient language and platform neutral mechanism for serializing structured data. We have implemented T2R as an open-source project, which is available [21]. We have also implemented and deployed a separate variant of T2R at Google. For this version, we use FlumeJava [22], Google's MapReduce API for implementing the collection and refactoring phases.

V. 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?

A. Software Corpora

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

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 [23]–[25]), SonarQube [26], CoreNLP [27], java-design-patterns [28], and Speedment [29]

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.

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.

B. 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.

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

VI. Results

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

A. Accuracy

Table IV 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 [30] 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"*.

Table IV: Results

Corpus		Candidate TRLCs	Accuracy: Patches Generated	Accuracy: Patches Passed	Usefulness: Patches Accepted	Usefulness: Files Changed	Usefulness: Lines Added	Usefulness: Lines Removed	Scalability: 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.

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.

B. 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 IV 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 [31] 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.

C. 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.

VII. DISCUSSION

A. 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 [4], 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 II). 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 [31].

B. Limitations of our Implementation

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.

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 [32] or APACHE GIRAPH [33].

Java Reflection and Dependency Injection. Our current implementation does not handle reflection or dependency injection. A workaround would be to collect associated annotations such as `@inject`, `@provide`, or `@autowire`, and introduce a precondition for checking these annotations.

Declaring Transformation Specifications. Our experiments revealed that the REFACTORING INSTRUCTIONS (as shown in Table III) are expressive enough for a wide range of refactorings. However, one needs to manually verify the completeness and correctness of the mappings.

VIII. RELATED WORK

We group the related work into two categories: (i) type-level refactorings, and (ii) large-scale refactorings.

Type-level refactorings. Previous work has proposed approaches for type-level refactorings, e.g., for facilitating class library migration by adaptors [34] or intermediate compatibility layers [35], generating new libraries for a constrained environment and migrating to them [36], migration based on manually-defined annotations [37] or capture-and-replay of changes [38], suggesting types in the migrated library based on existing examples and applying them [39], changing references and method invocations after library migration using an Eclipse plugin [40], and applying type changes by adopting a DSL for defining mappings [13].

Among others, a more promising approach for type migration is based on type constraints [7]–[9], [11], which determine the sites to be updated when replacing types, in order to preserve the type correctness of the program. Similarly, Khatchadourian [10] proposed a type checking technique to migrate legacy Java codebases to use enumerated types, designed to effectively handle primitive types.

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 between LCs, essentially capturing type constraints similar to the ones proposed by Tip et al. [7], yet making the analysis highly-scalable through a distributed MAPREDUCE approach. We showed that our IDE-independent 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 [10].

Previous studies [2], [5], [41] have shown that library migration (wherein type migration is the fundamental refactoring) is a frequent activity. Existing work [42]–[46] has proposed automated techniques to identify mappings between the migrated classes (e.g., method mappings) from the source code history. These approaches can relieve the user from manually defining the type migration mappings. We are going to extend T2R to incorporate such techniques in the future.

Large-scale refactorings.

ClangMR [47] 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.

IX. CONCLUSIONS AND FUTURE WORK

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 paper, 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.

ACKNOWLEDGMENTS

We thank Nikolaos Tsantalis, Raffi Khatchadourian, Titus Winters and other anonymous reviewers for their insightful feedback. This research was partially supported through NSF CCF-1553741 grant and a Google Faculty Research Award.

REFERENCES

[1] R. Saborido, R. Morales, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "Getting the most from map data structures in android," *Empirical Software Engineering*, 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9607-8

[2] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in Java," *J. Softw. Evol. Process*, vol. 26, no. 11, pp. 1030–1052, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1002/smr.1660

[3] 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)*, Feb 2014, pp. 64–73.

[4] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig, "Understanding the use of lambda expressions in Java," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 85:1–85:31, Oct. 2017. [Online]. Available: http://doi.acm.org/10.1145/3133909

[5] B. Cossette and R. 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*, 2012, pp. 55:1–55:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393661

[6] "Type migration refactoring," https://www.jetbrains.com/help/idea/migrate.html, accessed: 2018-06-06.

[7] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. De Sutter, "Refactoring using type constraints," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 3, pp. 9:1–9:47, May 2011. [Online]. Available: http://doi.acm.org/10.1145/1961204.1961205

[8] F. Tip, A. Kiezun, and D. Bäumer, "Refactoring for generalization using type constraints," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 13–26, 2003. [Online]. Available: https://doi.org/10.1145/949343.949308

[9] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 265–279. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094832

[10] R. Khatchadourian, "Automated refactoring of legacy Java software to enumerated types," *Automated Software Engineering*, vol. 24, no. 4, pp. 757–787, Dec. 2017.

[11] F. Tip and P. F. Sweeney, "Class hierarchy specialization," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 271–285, 1997. [Online]. Available: https://doi.org/10.1145/263700.263748

[12] J. Li, C. Wang, Y. Xiong, and Z. Hu, "SWIN: Towards Type-Safe Java Program Adaptation Between APIs," in *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '15. New York, NY, USA: ACM, 2015, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/2678015.2682534

[13] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 205–214. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806832

[14] R. M. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. 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*, 2005, pp. 71–96. [Online]. Available: https://doi.org/10.1007/11531142_4

[15] J. Dean and S. Ghemawat, "Mapreduce," *Communications of the ACM*, vol. 51, pp. 107–113, 2008. [Online]. Available: https://doi.org/10.1145/1327452.1327492

[16] Oracle, "Autoboxing," https://docs.oracle.com/javase/8/docs/technotes/guides/language/autoboxing.html, accessed: 2018-06-24.

[17] (2018) Corenlp. Accessed: 19 July 2018. [Online]. Available: https://github.com/stanfordnlp/CoreNLP

[18] T. Mens, S. Demeyer, and D. Janssens, "Formalising behaviour preserving program transformations," in *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings*, 2002, pp. 286–301. [Online]. Available: https://doi.org/10.1007/3-540-45832-8_22

[19] Google. (2010) Guava: Google core libraries for java. Accessed: 17 July 2018. [Online]. Available: https://github.com/google/guava

[20] ——. (2011) Protocol buffers. Accessed: 26 March 2018. [Online]. Available: https://developers.google.com/protocol-buffers/

[21] A. Authors. (2018) The open source implementation for T2R. Accessed: 18 July 2018. [Online]. Available: hidden

[22] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: Easy, efficient data-parallel pipelines," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. ACM, 2010, pp. 363–375. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806638

[23] (2018) Cassandra. Accessed: 23 August 2018. [Online]. Available: http://cassandra.apache.org/

[24] (2018) Prestodb. Accessed: 23 August 2018. [Online]. Available: https://prestodb.io/

[25] (2018) Neo4j. Accessed: 23 August 2018. [Online]. Available: https://neo4j.com/

[26] (2018) Sonarqube. Accessed: 23 August 2018. [Online]. Available: https://www.sonarqube.org/

[27] (2018) Stanford-corenlp. Accessed: 23 August 2018. [Online]. Available: https://stanfordnlp.github.io/CoreNLP/

[28] (2018) Java-design-patterns. Accessed: 23 August 2018. [Online]. Available: http://java-design-patterns.com/

[29] (2018) Speedment. Accessed: 23 August 2018. [Online]. Available: https://www.speedment.com/

[30] A. Thies and E. Bodden, "Refaflex: Safer refactorings for reflective Java programs," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336754

[31] (2019) Companion website. Accessed: 14 February 2019. [Online]. Available: https://ameyaketkar.github.io/T2RResults.html

[32] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[33] "Apache giraph," http://giraph.apache.org/, accessed: 2018-06-07.

[34] T. T. Bartolomei, K. Czarnecki, and R. 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*, 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/ICSM.2010.5610429

[35] D. Dig, S. Negara, V. Mohindra, and R. 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*, 2008, pp. 441–450. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368148

[36] V. L. Winter and A. 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*, 2007, pp. 185–196. [Online]. Available: http://doi.acm.org/10.1145/1289971.1290001

[37] K. Chow and D. 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*, 1996, pp. 359–368. [Online]. Available: https://doi.org/10.1109/ICSM.1996.565039

[38] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support API evolution," in *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062512

[39] Z. Xing and E. Stroulia, "API-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 818–836, 2007. [Online]. Available: https://doi.org/10.1109/tse.2007.70747

[40] P. Kapur, B. Cossette, and R. J. Walker, "Refactoring references for library migration," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 726–738, 2010. [Online]. Available: https://doi.org/10.1145/1932682.1869518

[41] C. Teyton, J. Falleri, and X. Blanc, "Mining library migration graphs," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 289–298. [Online]. Available: https://doi.org/10.1109/WCRE.2012.38

[42] ——, "Automatic discovery of function mappings between similar libraries," in *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*, 2013, pp. 192–201. [Online]. Available: https://doi.org/10.1109/WCRE.2013.6671294

[43] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, 2010, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806831

[44] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," vol. 20, no. 4, 2011, pp. 19:1–19:35. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000805

[45] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. 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, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[46] H. A. Nguyen, T. T. Nguyen, G. W. Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, 2010, pp. 302–321. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869486

[47] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-scale automated refactoring using clangmr," in *2013 IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 548–551. [Online]. Available: https://doi.org/10.1109/icsm.2013.93