Inferring and Applying Type Changes

Ameya Ketkar[*]
Uber Technologies Inc.
USA
ketkara@uber.com

Oleg Smirnov
JetBrains Research
St Petersburg University
Russia
oleg.smirnov@jetbrains.com

Nikolaos Tsantalis
Concordia University
Canada
nikolaos.tsantalis@concordia.ca

Danny Dig
University of Colorado Boulder
USA
danny.dig@colorado.edu

Timofey Bryksin
JetBrains Research
HSE University
Russia
timofey.bryksin@jetbrains.com

ABSTRACT

Developers frequently change the type of a program element and update all its references to increase performance, security, or maintainability. Manually performing type changes is tedious, error-prone, and it overwhelms developers. Researchers and tool builders have proposed advanced techniques to assist developers when performing type changes. A major obstacle in using these techniques is that the developer has to manually encode rules for defining the type changes. Handcrafting such rules is difficult and often involves multiple trial-error iterations. Given that open-source repositories contain many examples of type-changes, if we could infer the adaptations, we would eliminate the burden on developers. We introduce TC-Infer, a novel technique that infers rewrite rules that capture the required adaptations from the version histories of open source projects. We then use these rules (expressed in the Comby language) as input to existing type change tools. To evaluate the effectiveness of TC-Infer, we use it to infer 4,931 rules for 605 popular type changes in a corpus of 400K commits. Our results show that TC-Infer deduced rewrite rules for 93% of the most *popular* type change patterns. Our results also show that the rewrite rules produced by TC-Infer are highly effective at applying type changes (99.2% precision and 93.4% recall). To advance the existing tooling we released IntelliTC, an interactive and configurable refactoring plugin for IntelliJ IDEA to perform type changes.

KEYWORDS

Refactoring, source code mining, type change, type migration

ACM Reference Format:
Ameya Ketkar, Oleg Smirnov, Nikolaos Tsantalis, Danny Dig, and Timofey Bryksin. 2022. Inferring and Applying Type Changes. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510115

[*] Ameya Ketkar performed this work as part of his PhD at Oregon State University.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510115

1 INTRODUCTION

As programs evolve, the types of program elements are changed for several reasons, such as improving *performance* [13–15] (e.g., `String`→`StringBuilder`), *maintainability* [9] (e.g., `String`→`Path`), introducing *concurrency* [10] (e.g., `HashMap`→`ConcurrentHashMap`), handling *deprecation* or performing *library migration* [1, 29, 55] (e.g., `org.apache.commons.logging.Log`→`org.slf4j.Logger`). Such a refactoring where the type of a program element (i.e., variable, field, or method) is updated, and then 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 [31] than popular refactorings such as *rename*, tool support for type changes is negligible compared to refactoring automation. Developers predominantly perform type changes by hand [31]. This can be tedious, error-prone and it can easily overwhelm the developers. Researchers [5, 22, 30, 36, 41, 54, 58, 59] and tool builders [2, 16, 28, 43] 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 developers to encode the transformations. When a developer is unfamiliar with some 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 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 [7, 32]). This introduces a barrier to the adoption of these techniques.

Given that many software evolution tasks are repetitive by nature [23, 45, 46], our key insight is that developers from multiple open-source projects apply similar type changes in their projects. In our previous study [31] over a corpus of 400,000 type changes performed in 130 open source projects, we observed that 68% of

Table 1: Motivating Examples

	Element Before	Element After	Usages Before	Usages After	REWRITERULE
1.	int x;	**long** x;	x = 0;	x = **0L**;	:[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, fName))	Files.**newOutputStream**(x.**resolve**(fName))	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]> xs;	**Set**<:[t]> xs;	xs = new ArrayList<>(items);	xs = new **HashSet**<>(items);	new ArrayList<>(:[a])→ new HashSet<>(:[a])
12.			xs.get(0)	xs.**iterator().next()**	:[r].get(0) → :[r].iterator().next()

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 paper 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 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 Migration[28], or state-of-the-art tools that use type constraints [5] or type-fact graphs [30]. We leverage two state-of-the-art techniques : (i) REFACTORINGMINER [31, 56] to identify refactorings and (ii) COMBY [57] to represent and perform lightweight 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 4,931 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 3,060 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 INTELLITC [53], we run INTELLITC on four performance-critical open-source projects. INTELLITC generated 98 type changes which compile and pass tests successfully. At the time of writing, the original developers have already accepted 43 of them.

In summary the paper makes the following contributions:

(1) TC-INFER analyses the previously performed type changes and deduces the required adaptations as rewrite rules.
(2) INTELLITC assists developers at performing type changes by surfacing the rules produced by TC-INFER in an IDE.
(3) We empirically evaluated our TC-INFER to demonstrate its *applicability*, *effectiveness*, *trustworthiness*, and *utility*, and make our tools and data publicly available [52].

2 MOTIVATING EXAMPLES

Table 1 showcases a few scenarios that highlight the intricacies associated with inferring the rewrite rules. The first two columns (*Elements Before/After*) show the element whose type was changed, the next two columns (*Usages Before/After*) present the adapted usage of the element, and the last column presents the *Rewrite Rules* encoding the adapted usages using COMBY template syntax[8]. For instance, in row 9, the type change from AtomicLong to LongAdder involves renaming the call site 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 expression 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.

Developers apply a wide variety of edit patterns to adapt the usages of the element to the type change: Adding the `L` suffix (Table 1, row 1), replacing an instance method with a static method invocation (Table 1, row 2), updating a static method invocation (Table 1, row 8), or updating a class instance creation (Table 1, row 11). Often these edits adapt a commonly used idiom of a type. For instance, in Table 1, row 12, when the type change from `List` to `Set` is performed, the idiom `xs.get(0)` is replaced with the idiom `xs.iterator().next()`. Similarly in Table 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 method `Utils::trx`. The adaptations can also involve a composition of two edits. For instance, in Table 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 alter semantics (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. [5]. For instance, Dig et al. [10] proposed special preconditions for introducing concurrency (`Map`→`ConcurrentMap`), and Ketkar et al. [30] proposed special preconditions for eliminating *boxing*. One can imagine that type changes like `List`→`Set`, `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 (altering semantics or not), it can be useful if a tool can suggest (and apply) the transformations needed to adapt common idioms. For instance, when performing a type change `List`→`Set`, developers usually adapt the idiom `new ArrayList<>()` to `new HashSet<>()` and adapt `xs.get(0)` to `xs.iterator().next()`. 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.

3 TECHNIQUE

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`) in the input commits. Figure 1 gives an overview of the TC-Infer pipeline. First, TC-Infer collects all type change instances and other refactorings identified by RefactoringMiner in each input commit. RefactoringMiner uses its state-of-the-art statement matching algorithm to match statements across commits that accounts for refactorings like move class or method that rearrange the statements in the program. It then groups the reported type change instances by the type change pattern they relate to. Note that each type change instance contains the associated statement adaptations from the input commits. TC-Infer then preprocesses each type change instance to account for overlapping refactorings, such as renaming and extracting variables on top of the statement adaptations. Finally, TC-Infer infers the rewrite rules capturing each adaptation, and identifies relevant and safe edits (see Section 3.4.4 and Section 3.4.5). The final set of rewrite rules expresses the syntactic transformations required to adapt the source code elements to perform a particular type change.

At the heart of TC-Infer is the AST differencing algorithm InferRules (introduced in Algorithm 2) 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.

3.1 Basic Concepts

We will now describe some basic concepts.

Definition 3.1 (Abstract Syntax Tree, 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 3.2 (Template)**.** A lightweight way of matching syntactic structures of a program's parse tree, like expressions and function blocks. For Java, it is basically an arbitrary Java expression with template variables (or *holes*), that is matched to a program AST.

Recently, researchers van Tonder and Le Goues [57] proposed Comby, a multi-language syntax transformation technique for declaratively rewriting syntax with templates. We use the Java instantiation of Comby as our templating engine. Details of the syntax and matching behavior can be found on its website [8].

Definition 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 (like .*? in regex) within its level of balanced delimiters. The code snippet is matched to these 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.
- `:[n]` — if none of the above.

These specific kinds of template variables capture richer context when inferring rewrite rules, and minimize the spurious application of a rewrite rule.

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

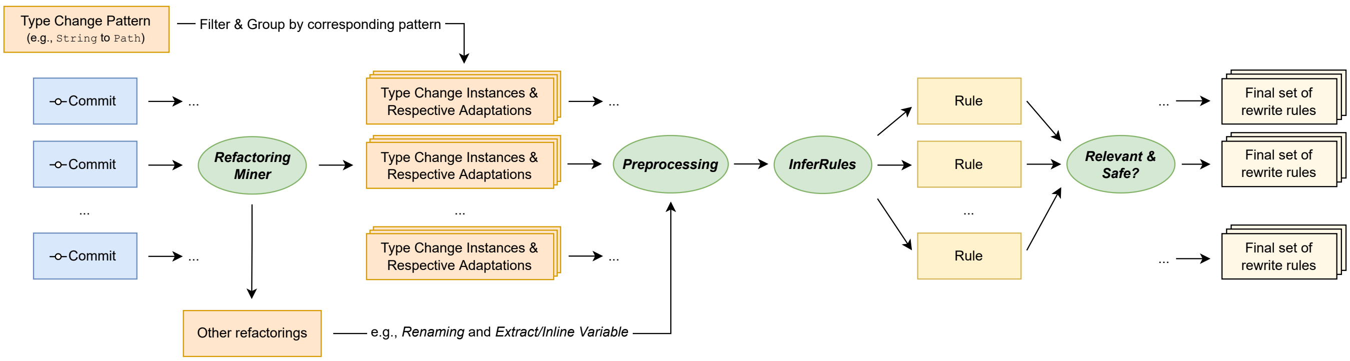

Figure 1: The high-level overview of the TC-Infer pipeline.

Definition 3.5 (GetTemplateFor). Given a code snippet c, this operation returns a template that captures the structure of an 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 special Java token(s) (e.g., keywords like `new` or `return`, or special characters like , or ;) (as shown in Example 3.1). This idea of inferring structural templates for code snippets is inspired from recent work by Luan et al. [38].

Definition 3.6 (Match). Given a template T and a code snippet c, Match returns a mapping between the TemplateVariables in T and syntactically valid sub-expressions of c iff the template T matches the entire snippet c (as shown in Example 3.1). This idea of using templates to infer edit patterns is inspired from recent work by Bader et al. [4].

Definition 3.7 (Substitute). Given a Template T and mappings from TemplateVariables in T to syntactically valid Java expressions, Substitute returns the template T' where the TemplateVariables in T are replaced with the corresponding expressions (as shown in Example 3.1).

Definition 3.8 (Rewrite). Given a rewrite rule $L \longrightarrow R$ or a list of rules $L_1 \longrightarrow R_1, \ldots L_n \longrightarrow R_n$ and a code snippet c, this operation applies (sequentially) the input rewrite rule on c.

Definition 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 (as shown in Example 3.1).

Definition 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 $m1$ (as shown in Example 3.1).

Definition 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 (as shown in Example 3.1).

Example 3.1. Some basic operations with Templates:

```
c1 = x.substr(1)
c2 = x.get().substr(1)
t1 = GetTemplateFor(c1)      # :[r].:[m](:[a~\d+])
t2 = GetTemplateFor(c2)      # :[r'].:[m'](:[a'~\d+])
m1 = Match(c1, t1)           # {r:x,m:substr,a:1}
m2 = Match(c2, t2)           # {r':x.get(),m':substr,a':1}
s1 = Substitute(t1, {r:foo()})   # foo().:[m](:[a])
m1 ∩ m2 ⟶ {m:m', a:a'}
m2 ∩^s m1 ⟶ {r':r}
m1 − m2 ⟶ [r]
RenameTemplateVars(t1, {r : x}) ⟶ :[x].:[m](:[a])
```

3.2 Input

We use RefactoringMiner to collect type changes and other refactorings performed. Recently Tsantalis et al. [56] 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 2). These matched statements could be a subset of all the 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 of *type change instances* reported by RefactoringMiner.

3.2.1 ***Pre-processing.*** It has been observed by previous researchers [31] 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 produced.

3.3 Output

For each type change pattern (i.e., `int`⟶`long` or `String`⟶`Optional<String>`) performed in the input type

```
- File fldr;
+ Path fldr;
- readfldr(fldr, mode, extensions)
+ readfldr(fldr.toFile(), extensions.toString())
- new ResourceHandler(dir,new Handler(
-                          new File(fldr)))
+ new ResourceHandler().set(new Handler(
+                        Paths.get(fldr)),dir)
- new FileOutputStream(new File(fldr,"test.txt"))
+ Files.newOutputStream(fldr.resolve("test.txt"))
```

Figure 2: Type Change Instance reported by RefactoringMiner for the type change pattern File→Path

change instances, our technique will produce a set of rewrite rules that can adapt the usages (type dependent idioms) to the new type.

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

As shown above, `TransformationSpec` contains a TypeChangePattern which is basically a RewriteRule like `int`→`long` or `List<:[t]>`→`Set<:[t]>`, and the RewriteRules that capture the necessary adaptation. In RewriteRules, each RewriteRule is associated to `Guards`, where these guards constrain the code snippet that binds to the TemplateVariables, either based on *regular expressions* and/or the return type of the code snippet. We obtain this information from the type inference provided in *Eclipse* JDT. For instance, for the rule `:[r].exists()`→ `Files.exists(:[r])` from Table 1, 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 [48] to the Comby language with additional *regex* based guards. The rewrite rules encoded in the Comby syntax can be losslessly translated to the IntelliJ Platform's structural replacement templates [27] or to the DSL proposed by Balaban et al. [5] and Ketkar et al. [30], since all of these are closely related to the Twining syntax. For each rewrite rule, TC-Infer also reports the real-world instances where the rewrite rule were performed.

3.4 TC-Infer

3.4.1 ***Generating the RewriteRules.*** Given two versions of a code snippet, the goal of GenerateRewriteRule in Algorithm 1 is to deduce the rewrite rule applied across them. The higher level intuition is the following: (1) capture the structure of the before and after code snippets as templates (T_1 and T_2), and (2) infer rewrite rules by mapping the holes of T_1 to the holes of T_2, if possible.

Example 3.2. Lets consider a simple example (Table 1, row 4).

```
- x = true;
+ x.set(true);
```

As described in Algorithm 1, we first construct a structural template (Definition 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 from Definition 3.9), 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])`.

Algorithm 1 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(c, GetTemplateFor(c)) for c in [c1, c2]]
11:     return RefineRule(LHS, RHS)
```

Example 3.3. Let's consider the adaptation (Table 1, row 7) applied to perform the type change from :[t]→Optional<:[t]>.

```
- Utils.trx(s)
+ 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 ∩ RHS = `{a:r'}`, we update the RHS to `:[[a]].:[[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', m: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])` → `:[a].map(:[[r]]:::[m])`.

3.4.2 ***Establishing Mappings.*** As described in Section 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 2, that looks for mappings between the two matched statements reported by RefactoringMiner. This Algorithm 2 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.

Example 3.4. Let's consider the following statement from Figure 2 adapted to perform the type change `File`→`Path`.

```
- new ResourceHandler(dir,new Handler(new File(fldr)))
```

Algorithm 2 The INFERRULES procedure

```
1: function GETWEIGHTS(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 []
```

```
+ new ResourceHandler().set(new Handler(
+                               Paths.get(fldr)),dir)
```

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 (Algorithm 1). The variable `fldr` was not generalized here because GENERATEREWRITERULE only decomposes the two template variables LHS and RHS if they intersect (Definition 3.9) or intersect-subtree (Definition 3.10). 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* [34] 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 node kinds do not match, but also accounts for reordering or alteration of the children list. The methods GETWEIGHTS and INFERRULES invoke INFERRULES (Line 11). TC-INFER tabulate the inferred templates against the offsets of the updated location to prevent this redundant computation.

3.4.3 *Inferring Composite Rewrite Rules.*

Example 3.5. Lets consider the adaptation from Table 1, row 3.

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

While the operation GENERATEREWRITERULE 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 yields 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 to merge the rewrite rules inferred for the 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). TC-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 into composite rewrite rules of the fine-grained rules.

3.4.4 *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), and (ii) if the rewrite rule contains template variables that match an expression (e.g., variable reference) of type S.

3.4.5 *Eliminating Unsafe REWRITERULES.*

The problem of expressing a 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, TC-INFER cannot generalize the adaptation 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 will make the rule context specific. Moreover, it is unclear how these scenarios could be expressed as rewrite rules. TC-INFER eliminates such *unsafe* rules from the output.

3.5 Comparison with Previous Work

Instead of INFERRULES (i.e. Algorithm 2), TC-INFER could also use other techniques that apply hierachical or greedy clustering techniques (Bader et al. [4], Rolim et al. [51]) suggested for inferring recurring edit patterns. For instance, for a type change pattern we could have (1) clustered all the corresponding adaptations from our dataset and generalize the tree patterns, (2) then applied anti-unification to generalize edit patterns, (3) then cluster the edit patterns, and (4) finally identify the *relevant* edit patterns. This would produce rewrite rules required for the `TransformationSpec`.

However, we did not adopt this strategy because (1) clustering tree patterns is an overkill for our problem which is constrained to expression- and statement-level transformations, (2) we have to account for overlapping refactorings and unrelated changes, (3) anti-unification for terms may not infer composite rules, and (4) many instances for each edit pattern applied to adapt a type change are unavailable (in most cases we have at most two examples).

4 EVALUATION

To understand the effectiveness, the real-world relevance, and the utility 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 practices for performing type changes? 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? We compare the application of rewrite rules inferred by TC-Infer to the changes performed by real-world developers, to highlight the benefits and the pitfalls of TC-Infer.
RQ4. Did developers find the RewriteRules useful? We investigate whether the rules produced by TC-Infer are useful to the developers to perform type changes in their IDEs.

4.1 Dataset

Previously, we conducted the first large-scale and the most fine-grained empirical study [31] on type changes performed in open source Java repositories on Github. In this previous work [31] we mined 297,543 type changes and their subsequent code adaptations from a diverse corpus of 129 Java projects containing 416,652 commits. With this rich dataset we answered research questions about the practice of type changes. This dataset contains instances of types with diverse characteristics with respect to their visibility (public, private), namespace (internal or application-specific, external, or JDK), kind (array, parameterized, simple, wildcard), and the relationship between the source and target types. We base our evaluation on the same dataset, because the diversity of the types involved in the type changes of this dataset will help to generalize our findings. We had identified 605 *popular type changes* performed in our dataset[1]. We considered a type change *popular* if it was performed in at least two unique projects. In this study, we evaluate the applicability and effectiveness of TC-Infer at inferring rules for these *popular* 605 type changes.

[1]https://zenodo.org/record/3906503#.Yfbnyy-B3T8

4.2 RQ1: 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 is TC-Infer for these type change patterns. We first applied TC-Infer upon all 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 at least one *prevalent rule*. We consider a rule *prevalent* if it is applied to adapt to the same type change in more than one commit. We identified 832 *prevalent* rules for the 274 type changes. By investigating the remaining 13.72% of type changes with no reported rule, we found: (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 it needed no update (e.g. replacing with super type, primitive widening, or boxing).

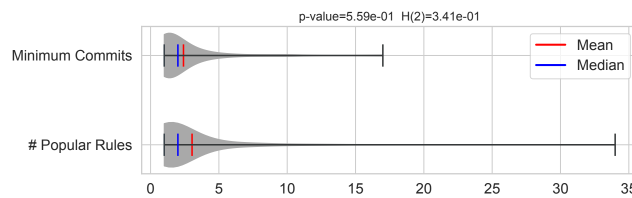

Figure 3: Distribution of the number of *prevalent* rules reported for each type change and the minimum number of commits required to infer the prevalent rules for each type change.

Figure 3 plots the distribution of the number of prevalent rules reported for the 274 type changes. The mean *prevalent* rules inferred for each type change is 3.48. TC-Infer produced one *prevalent* rule for the type change `Function<X,Integer>`→`ToIntFunction<X>`, while for `long`→`int` it produced 34 *prevalent* rules.

Analyzing a single commit where a particular type change is performed will not surface all *prevalent* rules, because the updated code may not use all corresponding APIs. The number of commits required to infer all prevalent 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 *prevalent* rules for each type change. Computing this smallest set of commits can be viewed as a *Set Cover* problem. Given a set of elements $\{1, 2, \ldots, n\}$ (called the universe, in our case, all *prevalent* rewrite rules for each type change) and a collection m of sets (in our case, these are the *prevalent* rewrite rules applied in the 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 the minimum weight. While this problem is *NP-Complete*, its greedy approximation algorithm [61] suffices for our purpose, since the cardinality of our universe is not very large (≤ 34). Figure 3 plots the distribution of the cardinality of the minimum set for each of the 274 type changes. On average, TC-Infer required approximately two commits to infer all prevalent rewrite rules for a type change, and required at most 18 commits to infer the 34 rules for the `int`→`long` type change.

The number of *prevalent* rules depend on various factors, such as the source and target type, the availability of examples, and the 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 possible rewrite rules for a type change are complete when they cover all the instance methods/constructors/fields in the source type. Previous researchers Li et al. [37] adopt this convention in their formalization of API migrations. However, this is not applicable to type changes, since type changes have to consider *all* possible usages of a particular type. For instance, when updating wrapper methods (`Integer.toString(x)`→`Long.toString(x)`), it is not possible to enumerate all common static method invocations that could act as such wrappers. Type changes can be semantics altering and sometimes no mappings are found for any member method or field (e.g., `x.trim()`→`x.get().trim()`).

TC-Infer deduced rules for diverse type change patterns:

(1) Involved variety of `AST Node Kinds` like primitive, simple, parameterized, or array types (e.g., `int`→`long`, `int` → `OptionalInt`), or `byte[]` → `ByteBuffer`)
(2) Involved `JDK` types, project-specific `Internal` types or `External` third party library types (e.g., `Predicate`→`IntPredicate`, Java's `List`→Guava's `ImmutableList`, or `String`→`hadoop.Path`)
(3) Involved `Interoperable` (e.g.,`File`→`Path`) and `non-Interoperable` (e.g., `List`→`Set`) type changes.

4.3 RQ2: Can we trust the existing practices for performing type changes?

In this question we want to understand if the current practice of performing type changes is 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 *prevalent* 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 *prevalent* 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 introduce a commonly disregarded code idiom. This list was obtained from the IntelliJ IDEA's Java Code Inspections [26].

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* nonconforming rewrite rules, as shown in Table 2 for four type changes. The first two rewrite rules in the second column of Table 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 an `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 for type changes using rewrite rules that are verified by domain experts.

4.4 RQ3: How effective are the RewriteRules for performing type changes?

To evaluatate the *effectiveness* of 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 IntelliTC, that is built upon *IntelliJ's Type Migration* framework [28], and can be configured via the `TransformationSpec` produced by TC-Infer. We then compare the changes performed by IntelliTC to those performed by the original developers.

4.4.1 ***IntelliTC***. This is our industry-strength tool [52] to perform type changes by leveraging *IntelliJ's Type Migration framework*. It allows the developers to express rewrite rules as IntelliJ Platform's structural replacement templates. Moreover, it operates in multiple modalities: (1) the *inspection mode* suggests the user to perform type changes based on the recommendations from Effective Java and other popular developer forums, (2) in the *classic mode*, developer can invoke IntelliTC as an *intention action* [25] (like rename refactoring), and (3) IntelliTC overcomes the *discoverability* and *late awareness* [18, 19] problem by surfacing certain type change refactorings through the *Suggested Refactoring* interface [24]. It also collects detailed *telemetry* information capturing how the developer is using IntelliTC. More details about IntelliTC and its usability can be found in our accompanying tool demonstration paper.

4.4.2 ***Identifying Test Scenarios***. Choosing commits for evaluating the *effectiveness* of our technique is not as straightforward as in the case of API Migration [35, 60], because randomly selected commits might not be using 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, based on the minimum sets of commits identified in Section 4.2. To replicate the type changes, we invoked IntelliTC for each instance in the 245 commits and manually compare the replicated type changes to the ones applied by the original developers.

4.4.3 ***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 [17] upon the files containing s_p and s_c. 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 2) upon the *real mapping* and collect

Table 2: Identified spurious rewrite rules introducing commonly disregarded idioms and the corresponding recommended rewrite rule (n: number of type change instances, C: number of commits, P: number of projects each rule is found in)

Type Change	Spurious Rule	n/C/P	Recommended Rule	n/C/P
`File→Path`	`:[v].getCanonicalPath()→:[v].toString()` `:[v].getCanonicalPath()→:[v].toAbsolutePath().toString()`	12/7/5 8/6/3	`:[v].getCanonicalPath()` `→:[v].toRealPath().toString()`	15/8/3
`File→Path`	`:[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

the rewrite rules (R). We then apply R (if $R \neq \emptyset$) upon the identified idioms in commit p, and manually validate:

(1) *True Positive:* the rule(s) R applied on s_p correctly adapts to the type change. In some scenarios, despite applying the correct change, s_p cannot be transformed to s_c, 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:* $R = \emptyset$ and the performed adaptation involves usage of new additional functionality or other unrelated changes.
(4) *False Negative:* $R = \emptyset$ 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.

4.4.4 ***Results***. In Table 3 we summarize the results of our experiment, which evaluated 245 instances of type changes belonging to 60 diverse kinds. 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 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 2). 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 `java.io.File → fs.hadoop.Path` to 100% for `AtomicLong→LongAdder`. We manually investigated each false negative and found three main reasons leading to them:

(1) **Additional context is required**. The most common reason for TC-Infer to produce no rules across a given statement mapping ($s_p \to s_c$) 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

Table 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	35	16	33	0	1.00	0.95
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	58	1	0.95	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
TemporaryFolder→File	9	34	2	14	8	1.00	0.54
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
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 Note that n, A and the ratio n/A vary based on the usage of the elements in the program of the type Channel from the context to replace the static method invocation with instance method invocation.

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

Capturing such edits will require comparing the changed data/control-flow across the commit or reason about more source code surrounding the applied edit. Previous researchers [4, 35, 60] 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.

(2) **Additional knowledge about the types is required**. 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 the scope of 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. With latest developments in *language server protocols* this challenge is surmountable.

(3) **Additional inference is required**. 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. Program synthesis techniques for string manipulations can easily overcome this challenge [21].

```
- 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.

4.5 RQ4: Did developers find the RewriteRules useful?

To answer this question, we perform popular type changes from our corpus that are also recommended by *Effective Java* [6], using IntelliTC 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 (Items 44 & 61 from [6]). We obtain the required specifications for eliminating these misuses from the rewrite rules collected in RQ1 (Section 4.2). Finding any missed opportunity to specialize interfaces in such projects is an important contribution because it eliminates boxing (un-boxing), thus improving the performance.

IntelliTC performed 98 instances of type changes belonging to 14 type change patterns that eliminate misuses of the Java 8 interfaces. These type changes updated 46 source code files and affected 213 SLOC. After IntelliTC 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 (limitation of the current implementation). Next, we sent out these type changes as pull requests to the maintainers of the projects. At the time of writing the paper, *two* PRs containing 43 type changes were accepted, and the rest are still under review.

5 LIMITATIONS AND THREATS TO VALIDITY

(1) **Preconditions**: Balaban et al. [5] laid out the basic preconditions for safely performing a type change involving interchangeable types (e.g., Vector⟶ArrayList). However, they are not always enough. Dig et al. [10] proposed additional preconditions to safely update HashMap to ConcurrentHashMap. 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. In our proposed workflow, we tradeoff safety for broader applicability by relying on the developer's wisdom in determining whether it is safe to update the type.

(2) **Version Awareness:** 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 build system configuration files (like pom.xml and build.gradle) to identify the required version of Java and other third party libraries.

(3) **Language Independence:** Currently TC-Infer is targetting the Java language, however conceptually it is language independent (note that Comby is also a multi-language syntax transformation technique). The only language dependent modules are (a) RefactoringMiner and (b) GetTemplateFor (Definition 3.5). While developing GetTemplateFor for other languages is straightforward, language-agnostic refactoring detection is also tractable. For example, recently researchers Atwi et al. [3] reimplemented RefactoringMiner in Python to support the Python language accounting for its dynamic nature, whereas Dilhara [11] proposed a technique that *Java-fies* Python programs and enables Java based AST analysis tools to process Python.

(4) **External Validity:** Do our results generalize? We studied 130 projects on Github from a wide range of application domain, making the results of the study *generalizable* to other projects. Moreover, the type changes we used for evaluating the applicability and the effectiveness of our technique are diverse in nature (w.r.t. syntactic category, name space or inter-operability). We show that the produced rules can achieve high precision and recall.

(5) **Internal Validity:** Does our tool produce valid results? We thoroughly evaluate the accuracy of the rewrite rules produced by TC-Infer. To understand if the inferred rules can be trusted,

the authors manually validate the *prevalent* rules to identify non-conforming ones. Moreover, we create an extensive setup that semi-automatically validates the application of rewrite rules for a large and diverse variety of type change patterns.
(6) **Verifiability:** The collected data, source code, and executable of TC-Infer and IntelliTC are publicly available [52].

6 FUTURE WORK

As seen in Section 4.2, TC-Infer could infer at least one rule for 86% of the popular type changes applied in the open source Java repositories. In Section 4.3, we replicate the type changes performed by developers, and show that the rules produced by TC-Infer are very effective (99.2% precision and 93.4% recall). While Section 4.4 shows that these applicable and effective rules should undergo manual vetting, because they cannot be blindly trusted. For TC-Infer to make impact in the real world it is important to reason about (1) storing and accessing the inferred rules, (2) policies for contributing new rules, and (3) maintainence of these rules.

We envision that our central database will contain two views: (1) a general view that contains rules inferred for the common and popular type changes performed in the version history of all the participating projects, (2) a project-specific view that contains the rules inferred for the type changes performed in the version history of a particular project. This database can be continuously updated with each new commit. Each rule will be associated with the exact location in the version history where it was performed, along with some confidence metrics based on the number of projects, commits and developers who performed it. The users could also manually submit new rules to the database and be able to upvote or downvote rewrite rules based on their understanding of the APIs/operators (i.e., community-driven confidence). The majority of type changes in the project-specific view will be application specific type changes that will be useful to the developers of the project and the other dependent projects (in case of breaking changes). In case of competing rules (i.e., same LHS, different RHS), the user can rank these rules based on the community's perspective and empirical evidence.

The most crucial aspect of maintaining and evolving rules is the version awareness. To make the rules version aware, we could take a conservative approach by annotating the rules with the versions in the associated real world example. However, this will thoroughly reduce the applicability of these type change rules. We believe we need further research to infer if the rewrite rules are backward or forward compatible. We believe human insight will be required to maintain the quality of the rules. Therefore, our envisioned tool leverages the community's perspective and empirical evidence.

7 RELATED WORK

(1) **Program Transformation Systems:** Researchers have proposed an array of advanced program transformation systems and impressive meta-programming languages: (a) `JunGL` [58] is an ML-style functional programming language that facilitates AST manipulation with higher order functions and tree matching, (b) `Refcola` [54] is a constraint language where refactorings are specified by constraint rules, (c) `Wrangler` [36] provides refactoring commands to locate program elements and a DSL to execute the commands in the context, (d) `Rascal` [22] is a scripting language to execute Eclipse JDT refactorings, and (e) `Error-Prone` [20] is a static analysis tool to catch and fix common programming mistakes at compile time. While these advanced systems can be used to encode type changes, Kim et al. [32] have shown that encoding refactorings in these *domain specific languages* has an unncessary overhead and a steep learning curve (weeks to months). Other researchers Balaban et al. [5], Ketkar et al. [30], Wright [59] have developed frameworks specifically to perform type changes based on input transformation specifications. In contrast to all these systems, the goal of our work is to remove the burden on the developers to encode type changes in these DSLs.
(2) **Inferring and Applying Edit patterns:** Researchers have proposed a plethora of techniques that can infer and apply a variety of edit patterns from commit-level changes and finer IDE-level changes: (a) `GetAFix` [4], `Revisar` [51], and `DeepDelta` [42] infer fixes for bugs and compilation errors from commit histories of the project using clustering, `anti-unification` or deep learning techniques, (b) `Refazer` [50] applies program synthesis to fix incorrect student assignments, while `BluePencil` [44] learns repetitive code changes on-the-fly in an IDE, (c) `CPATMiner` and `Py-CPATminer`[12] identify the repetitive and frequent applied edit patterns in a code reposiory (d) `LibSync` [47], `A3` [35] and `MEditor` [60] infer the adaptations required to perform library migration by analyzing the changed control/data flow across the commit, (e) Kim et al. [33] discover and represent systematic changes as logic rules with the goal to enhance developer's understanding about the program's evolution, and (f) `Sydit` [39], `LASE` [40], `Repertoire` [49] perform systematic code changes by creating a context-aware edit script, finding potential locations and transforming the code. In contrast to these works, the TC-Infer deduces rewrite rules for adapting common syntactic idioms and IntelliTC automates them in the IDE.

8 CONCLUSIONS

Type change is a crucial activity in evolving code bases. While performing type changes manually is tedious, using the current state-of-the-art type change automation techniques is not straightforward because it requires the developer to encode the adaptations in a DSL. This paper eliminates this burden on the developers. We present TC-Infer that deduces the rewrite rules required to perform the type change from the version history. We evaluate the TC-Infer's applicability for inferring rules for *popular* type changes, and show the effectiveness of these rules at performing 3,060 instances of 60 diverse type change patterns. We also developed IntelliTC and applied it to eliminate 98 misuses of the Java 8 APIs in four large open source projects.

9 ACKNOWLEDGEMENT

We would like to thank Rijanrd van Tonder, Martin Erwig, Ali Mesbah, the CUPLV group at CU Boulder and other anonymous reviewers for their insightful and constructive feedback to improve the work. This research was partially supported by NSF grants CCF-1553741 and CNS-1941898, NSERC grant RGPIN2018-05095, and by the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.

REFERENCES

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

[2] Apache. 2019. *Netbeans Refactoring.*

[3] H Atwi, B Lin, N Tsantalis, Y Kashiwa, Y Kamei, N Ubayashi, G Bavota, and M. Lanza. SCAM. PyRef: Refactoring Detection in Python Projects. In *SCAM, 2021*. https://doi.org/PyRef/PyRef

[4] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360585

[5] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 265–279. https://doi.org/10.1145/1094811.1094832

[6] Joshua Bloch. 2018. *Effective Java* (3 ed.). Addison-Wesley, Boston, MA. https://www.safaribooksonline.com/library/view/effective-java-third/9780134686097/

[7] Marat Boshernitsan, Susan L. Graham, Susan L. Graham, and Marti A. Hearst. 2007. Aligning Development Tools with the Way Programmers Think About Code Changes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. ACM, New York, NY, USA, 567–576. https://doi.org/10.1145/1240624.1240715

[8] Comby. 2021. *Comby Syntax Reference.* https://comby.dev/docs/syntax-reference Accessed: 3 Sep 2021.

[9] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. 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* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 107–117. https://doi.org/10.1145/3236024.3236042

[10] Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 397–407. https://doi.org/10.1109/ICSE.2009.5070539

[11] Malinda Dilhara. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1683–1685. https://doi.org/10.1145/3468264.3473493

[12] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *International Conference on Software Engineering* (Pittsburgh, United States) *(ICSE '22)*. ACM/IEEE. https://doi.org/10.1145/3510003.3510225 To appear.

[13] Java Platform Documentation. 2019. *Autoboxing and unboxing.*

[14] Java Platform Documentation. 2019. *StringBuffer.*

[15] Java Platform Documentation. 2019. *StringBuilder.*

[16] Eclipse. 2019. *Refactoring Actions.*

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

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

[19] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 211–221.

[20] Google. 2011. *Error Prone.* https://github.com/google/error-prone

[21] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*. https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

[22] Mark Hills, Paul Klint, and Jurgen J. Vinju. 2012. Scripting a Refactoring with Rascal and Eclipse *(WRT '12)*. Association for Computing Machinery, New York, NY, USA, 40–49. https://doi.org/10.1145/2328876.2328882

[23] Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the Naturalness of Software. *Commun. ACM* 59, 5 (April 2016), 122–131. https://doi.org/10.1145/2902362

[24] IntelliJ. 2021. *IntelliJ Inplace Rename.* https://www.jetbrains.com/help/idea/rename-refactorings.html#inplace_rename Accessed: 3 Sep 2021.

[25] IntelliJ. 2021. *IntelliJ Intention Actions.* https://www.jetbrains.com/help/idea/intention-actions.html Accessed: 3 Sep 2021.

[26] IntelliJ. 2021. *IntelliJ Java Inspections.* https://www.jetbrains.com/help/idea/list-of-java-inspections.html#probable-bugs Accessed: 3 Sep 2021.

[27] IntelliJ. 2021. *IntelliJ: Structural Search and Replace.* https://www.jetbrains.com/help/idea/structural-search-and-replace.html Accessed: 3 Sep 2021.

[28] JetBrains. 2019. *Type Migration.*

[29] Suhas Kabinna, Cor-Paul Bezemer, Weiyi Shang, and Ahmed E. Hassan. 2016. 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*. 154–164. https://doi.ieeecomputersociety.org/10.1109/MSR.2016.025

[30] Ameya Ketkar, Ali Mesbah, Davood Mazinanian, Danny Dig, and Edward Aftandilian. 2019. Type Migration in Ultra-large-scale Codebases. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 1142–1153. https://doi.org/10.1109/ICSE.2019.00117

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

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

[33] Miryung Kim, David Notkin, Dan Grossman, and Gary Wilson. 2013. Identifying and Summarizing Systematic Code Changes via Rule Inference. *IEEE Transactions on Software Engineering* 39, 1 (Jan 2013), 45–62. https://doi.org/10.1109/TSE.2012.16

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

[35] Maxime Lamothe, Weiyi Shang, and Tse-Hsun Peter Chen. 2020. A3: Assisting Android API Migrations Using Code Examples. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2988396

[36] Huiqing Li and Simon Thompson. 2012. A Domain-Specific Language for Scripting Refactorings in Erlang. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 501–515.

[37] Jun Li, Chenglong Wang, Yingfei Xiong, and Zhenjiang Hu. 2015. SWIN: Towards Type-Safe Java Program Adaptation Between APIs. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. ACM, New York, NY, USA, 91–102. https://doi.org/10.1145/2678015.2682534

[38] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 152 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360578

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

[40] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. LASE: Locating and Applying Systematic Edits by Learning from Examples. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 502–511.

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

[42] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 925–936. https://doi.org/10.1145/3338906.3340455

[43] Microsoft. 2021. Visual Studio. (2021). At https://www.visualstudio.com.

[44] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

[45] H. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton. 2019. Graph-Based Mining of In-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 819–830. https://doi.org/10.1109/ICSE.2019.00089

[46] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A Study of Repetitiveness of Code Changes in Software Evolution. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering* (Silicon Valley, CA, USA) *(ASE'13)*. IEEE Press, 180–190. https://doi.org/10.1109/ASE.2013.6693078

[47] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. 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*. 302–321. https://doi.org/10.

1145/1869459.1869486

[48] Marius Nita and David Notkin. 2010. Using Twinning to Adapt Programs to Alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1806799.1806832

[49] Baishakhi Ray, Christopher Wiley, and Miryung Kim. 2012. 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* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 8, 4 pages. https://doi.org/10.1145/2393596.2393603

[50] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

[51] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. (2018). http://arxiv.org/abs/1803.03806

[52] Oleg Smirnov, Ameya Ketkar, Timofey Bryksin, Nikolaos Tsantalis, and Danny Dig. 2021. *IntelliTC: Automating Type Changes in IntelliJ IDEA.* https://type-change.github.io/index.html Accessed: 10 Feb 2022.

[53] Oleg Smirnov, Ameya Ketkar, Timofey Bryksin, Nikolaos Tsantalis, and Danny Dig. 2022. IntelliTC: Automating Type Changes in IntelliJ IDEA. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion)* (Pittsburgh, United States) *(ICSE '22 Companion)*. ACM/IEEE. https://doi.org/10.1145/3510454.3516851

[54] Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. 2011. A Refactoring Constraint Language and Its Application to Eiffel. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 255–280.

[55] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A Study of Library Migrations in Java. *J. Softw. Evol. Process* 26, 11 (Nov. 2014), 1030–1052. https://doi.org/10.1002/smr.1660

[56] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 1–21. https://doi.org/10.1109/TSE.2020.3007722

[57] Rijnard van Tonder and Claire Le Goues. 2019. Lightweight Multi-Language Syntax Transformation with Parser Parser Combinators. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 363–378. https://doi.org/10.1145/3314221.3314589

[58] Mathieu Verbaere, Arnaud Payement, and Oege de Moor. 2006. Scripting refactorings with JunGL. In *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*, Peri L. Tarr and William R. Cook (Eds.). ACM, 651–652. https://doi.org/10.1145/1176617.1176656

[59] Hyrum K. Wright. 2020. Incremental Type Migration Using Type Algebra. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 756–765. https://doi.org/10.1109/ICSME46990.2020.00085

[60] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, Piscataway, NJ, USA, 335–346. https://doi.org/10.1109/ICPC.2019.00052

[61] Guangtun Zhu. 2016. A New View of Classification in Astronomy with the Archetype Technique: An Astronomical Case of the NP-complete Set Cover Problem. arXiv:1606.07156 [astro-ph.IM]