SWIN: Towards Type-Safe Java Program Adaptation between APIs

Jun Li[1,2] Chenglong Wang[1,2] Yingfei Xiong[1,2] Zhenjiang Hu[3,1,2]

[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education
[2]Software Engineering Institute, Peking University, Beijing, 100871, China
[3]National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
{lij, chenglongwang, xiongyf}@pku.edu.cn, hu@nii.ac.jp

Abstract

Java program adaptation between different APIs is a common task in software development. When an old API is upgraded to an incompatible new version, or when we want to migrate an application from one platform to another platform, we need to adapt programs between different APIs. Although different program transformation tools have been developed to automate the program adaptation task, no tool ensures type safety in transforming Java programs: given a transformation program and any well-typed Java program, the transformed result is still well-typed. As a matter of fact, it is often observed that a dedicated adaptation tool turns a working application into a set of incompatible programs.

We address this problem by providing a type-safe transformation language, SWIN, for Java program adaptation between different APIs. SWIN is based on Twinning, a modern transformation language for Java programs. SWIN enhances Twinning with more flexible transformation rules, formal semantics, and, most importantly, full type-safe guarantee. We formally prove the type safety of SWIN on Featherweight Java, a known minimal formal core of Java. Our experience with three case studies shows that SWIN is as expressive as Twinning in specifying useful program transformations in the case studies while guaranteeing the type safety of the transformations.

1. Introduction

Modern programs often depend on different APIs (Application Programming Interfaces), and it is a common task for the developers to adapt programs between alternative APIs. One example is *API update*: when an old API is updated to a new version with incompatible changes, we need to transform client programs with the old API to new programs using the new API. Another example is *API switching*: we often need to migrate programs between different platforms, such as from the Android platform to iOS, or from Java Swing to SWT. In such cases, we need to transform the programs with the API on one platform to new ones with the API on another platform. However, manually adapting programs is not easy: we need to examine every use of the source API and replace them with the suitable target API. Thorough knowledge of the source and target APIs as well as the client program is required.

Given the importance of program migration, it would be helpful and beneficial for tool vendors to provide automated tools to assist application adaptations. When API upgrades, API providers could provide tools to automate the upgrade of client applications, preventing potential loss of users from the incompatibility of the new API. Similarly, platform providers could provide tools to facilitate the migration of application from other platforms to their own platforms, attracting more applications and users on their platform. For example, Microsoft has provided the Visual Basic upgrade wizard tool, to facilitate the transition from Visual Basic to Visual Basic.Net. RIM has provided a tool suite to transform Android applications into blackberry applications. These tools work in the form of program transformation: they take a client program as input, and produce a new program that preserves the behavior of the source program as much as possible while targeting the new API.

However, providing a program transformation tool is not easy. Among the large body of API adaptations performed in practice, only a small portion has transformation tool supports, and it is common for the transformation tools to introduce bugs in the transformed programs. A particular type of bugs we are concerned with in this paper is type error: a well-typed program becomes *not* well-typed after the transformation. For example, Python has provided an official 2to3 script to transform Python programs from Python 2.x to 3.x. However, as discovered in a case study by Pilgrim and Willison [21], the script will introduce a type error in the transformed code whenever the original code contains a call to the "`file()`" method.

To overcome the difficulty of providing transformation tools, a large number of program transformation languages [1, 11–13, 20] have been proposed. These languages provide high-level constructs for specifying transformations between programs, reducing the development cost and preventing certain kinds of errors. For example, a number of program transformation languages prevent the possibility of introducing grammatical errors in transformation, either by specifying the transformation on top of context-free grammars [11, 12] or by designing the transformation language specifically for a programming language [1]. However, as far as we know, no transformation language for mainstream object-oriented programs ensures *type safety*: for any transformation program p and a well-typed source program s, the transformed program $p(s)$ is still well-typed. As a result, given a program transformation, we have no guarantee that a well-typed program will still be well-typed after the transformation.

PEPM '15, January 13–14, 2015, Mumbai, India.
Copyright © 2015 ACM 978-1-4503-3297-2/15/01…$15.00.
http://dx.doi.org/10.1145/2678015.2682534

It is not easy to ensure type safety in transformation languages. We highlight two challenges here. First, typing is one of the most complex components in modern programming language design, involving many interleaving of issues. The design of a transformation language needs to carefully check each intersection of the issues, which is not an easy job. Second, type safety involves two aspects: correctness and completeness. Correctness means that every transformed piece in the program is well-typed, while completeness means that all unchanged pieces are still well-typed under the new API. It is easy to ignore one aspect in transformation language design. As a matter of fact, Twinning [1], a modern transformation language for Java programs, have introduced strict rules for checking types in the transformation program to prevent the introduction of type errors. However, as our motivation section will show later, these rules still fail to establish full type safety.

In this paper we report our first attempt to design a type-safe transformation language for Java. As the first attempt, we focus on the class of one-to-many mappings between APIs. One-to-many mappings mean one method invocation in the source API will be replaced as one or multiple method invocations in the target API with possible gluing code. We choose this class for two reasons. 1) One-to-many mappings are dominant in the migration between alternative APIs. An empirical study [22] shows that 95.3% of the required changes are one-to-many mapping in the API update of struts, log4j, and jDOM. 2) Studying one-to-many mappings is a necessary step toward more general many-to-many mappings. Since one-to-many mappings are a sub class of many-to-many mappings, type safety in many-to-many mappings requires type safety in one-to-many mappings. As a matter of fact, the language Twinning is designed for one-to-many mappings, and is known for its simplicity and usefulness in many adaptation applications. Our approach is built upon Twinning, where we add extra conditions to ensure type safety.

More concretely, our contributions are summarized as follows.

- We propose a new transformation language, SWIN (Safe tWINning), for Java program adaptation between alternative APIs. The SWIN language is based on Twinning [1], a modern program adaptation language for Java. Compared with Twinning, SWIN includes a set of type checking rules to ensure type safety. These type checking rules enable a cross-checking over the source API, the target API, and the transformation program, and ensure that any well-typed Java program using the source API will be transformed into a well-typed Java program only using the target API, if the transformation program is well-typed under the type checking rules. SWIN also has more flexible replacement rules than Twinning.
- We formalize a core part of SWIN, known as core SWIN. Core SWIN works on Featherweight Java (FJ) [7], a formal model of the core Java language often used to reason typing-related properties of Java. We formally prove the type safety of core SWIN on FJ. We also informally describe the rest of SWIN and discuss the type safety of full SWIN.
- We have implemented SWIN[1] and have evaluated SWIN by implementing three real world transformation programs in SWIN. These programs ranges from web APIs accessed by HTTP requests [19] to local APIs, including both API updating and API switching. Our case study shows that SWIN is able to specify a range of useful program transformations in practice. More importantly, compared with Twinning, the additional type checking rules in SWIN does not confine the expressiveness of the language.

[1] https://github.com/Mestway/SWIN-Project

The rest of our paper is structured as follows. Section 2 briefly introduces Twinning, and then give two motivating examples to show why Twinning is not type-safe in program adaptation. Section 2 also discusses how to maintain type safety in program adaptation. Section 3 presents core SWIN, with an introduction to Featherweight Java. Section 4 gives the type system for core SWIN, as well as the proof of type safety while transforming programs using rules in SWIN on Featherweight Java. Section 5 explains how to extend core SWIN to full SWIN. Section 6 presents three case studies which demonstrate the expressiveness of SWIN. Finally, Section 7 discusses related work and Section 8 concludes the paper.

2. Motivating Examples

Before explaining SWIN, we briefly explain the type safety problem in the existing systems. We shall first briefly describe Twinning [1], a typical API adaptation language. Then, we will give some examples to show why Twinning cannot preserve the type correctness in program transformation. Finally, we will informally present an overview of our work.

Twinning is a rule-based language for adapting programs between alternative APIs. The design goal of Twinning is to be easy to use while allowing a reasonable set of adaptation tasks to be specified. A Twinning program basically consists of a set of replacement rules in the form of

$$\begin{array}{l}[\\ \quad \mathtt{T_{10}}(\mathtt{T_{11}}\ \mathtt{x_1}, \ldots, \mathtt{T_{1n}}\ \mathtt{x_n})\ \{\ \textbf{return}\ \text{javaExp}_1;\ \}\\ \quad \mathtt{T_{20}}(\mathtt{T_{21}}\ \mathtt{y_1}, \ldots, \mathtt{T_{2n}}\ \mathtt{y_n})\ \{\ \textbf{return}\ \text{javaExp}_2;\ \}\\ ]\end{array}$$

which means (1) $\mathtt{T_{1i}}$ will be replaced by $\mathtt{T_{2i}}$ for all $\mathtt{i}$ (the set of pairs $[\mathtt{T_{1i}}, \mathtt{T_{2i}}]$ from all replacement rules are called a *type mapping*); (2) $\mathtt{x_i}$ is a meta variable that will match a Java expression of type $\mathtt{T_{1i}}$ in the source code and instantiates $\mathtt{y_i}$ with that expression; (3) javaExp_1, which is a Java expression of type $\mathtt{T_{10}}$ that uses meta variables $\mathtt{x_1} \ldots \mathtt{x_n}$, will be used to match Java expressions, and these expressions will be replaced by javaExp_2 of type $\mathtt{T_{20}}$, where the meta variables $\mathtt{y_i}$ are instantiated with the matched expressions by $\mathtt{x_1} \ldots \mathtt{x_n}$ [2].

As a simple example, consider the following replacement rule

```
[
 Enumeration(Hashtable x)
   {return x.elements();}
 Iterator(HashMap x)
   {return x.values().iterator();}
]
```

which will match any call to `elements` in class `Hashtable`, and replace it by a call to `values().iterator()` in class `HashMap`. For instance, given the following piece of code,

```
void f(Hashtable t) {
  Enumeration e = t.elements():
  ...
}
```

the replacement rule will produce the following piece of code, where the meta variable `x` in the replacement rule matches the expression `t`.

```
void f(HashMap t) {
  Iterator e = t.values().iterator():
  ...
}
```

[2] Strictly speaking, Twinning also allows replacing a block of statements rather than a single expression. For the ease of presentation, we shall only consider expression replacement in this paper. All discussions apply to statements replacement as well.

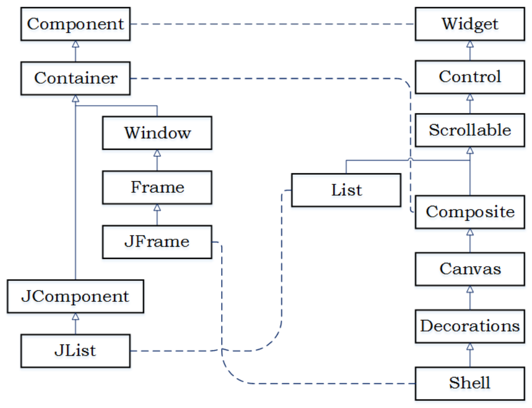

Figure 1. Swing (left) and SWT (right) type mapping: boxes represent classes, arrows indicate the class hierarchy, and dotted lines indicate the type mapping relations.

Twinning mainly checks two conditions to avoid introducing new type errors in the code. First, Twinning requires each replacement must be well-typed under the typing rules of Java. In this way, we can ensure the replacement of expressions does not introduce new type errors. Second, Twinning requires that one type is only mapped to one type in the type mapping (i.e., one type cannot be mapped to different types by replacement rules). This condition ensures that the replacement of types can be correctly performed.

Unfortunately, these two conditions cannot fully ensure type safety. First, type errors may be introduced when subtyping relations are involved. To see this, consider a practical example to adapt programs from Java Swing API to SWT API [2], where the correspondences between types of the two APIs are summarized in Figure 1, and the following presents part of the rules for replacing type constructors to their counterparts.

```
[
 Container () {return new Container();}
 Composite ()
  {return new Composite(new Shell(), 0);}
]

[ JList () { return new JList(); }
  List  () { return new List(); } ]
```

The typing problem happens if we apply the above rules to the following piece of code:

```
Container x = new JList();
```

Clearly, it will yield the code

```
Composite x = new List();
```

which actually contains a type error: JList is a subtype of Container, but List is not a subtype of Composite, so we cannot assign a `List` object to a `Composite` variable. This example shows that, although the two conditions used in Twinning ensure the replacement of expressions and the replacement of types are correct by themselves, the *intersection* of the two replacements would introduce type errors.

Second, Twinning has no guarantee the replacement rules cover all necessary changes. When there are components appearing only in the old API but are not transformed by any transformation rule, type errors may be introduced. For instance, consider the upgrade of Java SDK from v1.0 to v1.2: class `Hashtable` (Figure 2) is replaced by `HashMap` (Figure 3). For this change, we write a set of replacement rules (Figure 4). To be sure that any program using `Hashtable` can be transformed in a type-safe way, we must guarantee that all methods and classes in `Hashtable` have their replacements. However, the method `contains` in class `Hashtable` has no such replacements in the above set of rules.

```
class Hashtable {
    Enumeration elements() { }
    boolean contains(Object v) { }
    ...
}
class Enumeration {
    ...
}
```

Figure 2. Hashtable API

```
class HashMap {
    Collection values() { }
    boolean containsValue(Object v) { }
    ...
}
class Collection {
    Iterator iterator() { }
    ...
}
class Iterator {
    ...
}
```

Figure 3. HashMap API

```
[ Hashtable () { return new Hashtable();}
  HashMap () { return new HashMap(); } ]

[ Enumeration (Hashtable x)
    { return x.elements(); }
  Iterator (HashMap x)
    { return x.values().iterator(); } ]
```

Figure 4. Replacement Rules From Hashtable to HashMap

In summary, the conditions of Twinning are not enough to ensure type-safety of the transformation program. We need additional conditions to prevent the above two problems. For the first case, we need to ensure that the type mapping does not break the subtyping relations. For the second case, we need to ensure the replacements cover the full API changes. Putting them together with the original two conditions from Twinning, we have the following four conditions.

- For each code snippet introduced in a replacement rule, the code snippet itself must be well-typed.
- The type mapping must form a function, i.e., no type in the source API is mapped to two or more types in the target API.
- The type mapping must preserve the subtyping relation. If X is a subtype of Y in the source API and m is the mapping, $m(X)$ must be a subtype of $m(Y)$ in the target API.
- The replacement rules must cover all type changes between the source API and target API.

It will be interesting to see later that these four conditions are sufficient to ensure type safety. However, as Twinning is presented informally in the original publication [1], to reason about type safety, we need to first build a formal model of the Twinning semantics. A particular challenge of presenting this formal model is to understand how the replacement rules can be sequentially applied. For example, to transform the following piece of code

```
new Hashtable().elements()
```

into

```
new HashMap().values().iterator()
```

we need to begin with the second rule in Figure 4 to replace "`elements()`" and then apply the first rule to replace "`new Hashtable()`". If we begin with the first rule, we shall get an expression

```
new HashMap().elements()
```

where the second rule cannot be applied because "`new HashMap()`" has a type `HashMap` that cannot be matched by the meta variable `x` of type `Hashtable`. In other words, the transformation is not *confluent* since applying the rules in different orders gives us different results.

A related issue is that some sequences of rule applications may be infinite. For example, let us consider the following rule.

```
[A (A x) {return x.a();}
 A (A x) {return x.a().a();}]
```

Since the target side of the right also contains the call to `a()`, the rule can be applied again after the transformation, forming a *non-terminating* transformation. A terminating and confluent transformation is called a *convergent* transformation. A well-formed transformation language should always produce convergent transformations. However, the publication on Twinning [1] provides no information how Twinning deals with these issues.

Another usability issue of Twinning is that Twinning allows only exact type matching, i.e., a meta variable of type T matches a Java expression only when the expression has exactly type T but not a subtype of T. This design eases the analysis as we can infer all type changes from the type mapping, but also makes transformation more difficult to write. For example, in Java v1.0 class `Properties` is a sub class of `Hashtable`, and thus any call to `Properties.elements()` should be transformed in the same way as `Hashtable.elements()`. However, the second rule in Figure 4 does not apply to calls to `Properties.elements()` because the meta variable `x` has type `Hashtable`. As a result, for any replacement rule for a class `C`, we need to repeat the rule for each sub class of `C`, which is quite tedious.

To overcome this problem, we design a new language, SWIN (Safe tWINning). SWIN is based on Twinning but with the following differences.

- SWIN has full formal semantics.
- SWIN has more flexible rule application behavior, allowing a meta variable to match an expression of its sub type.
- SWIN is convergent. A well-typed SWIN program can act on any Java program confluently and free from non-terminating problems.
- SWIN includes a set of type checking rules to check the four conditions presented above.

In the following sections we shall introduce SWIN formally and present our proof of type safety.

3. Syntax and Semantics of Core SWIN

Before explaining full SWIN for Java, which will be discussed in Section 5, we start with core SWIN for Featherweight Java [7], a known minimal core of Java. If no confusion will be caused, we shall directly use SWIN to refer core SWIN. We shall briefly review Featherweight Java, and explain the syntax and semantics of our transformation language SWIN for it.

3.1 Background: Featherweight Java

Featherweight Java (FJ for short) is a minimal core calculus for Java [7]. FJ is small enough that a concise proof of the type-safety property is possible while it can be easily extended to full Java.

Class Declaration

$$\mathtt{CL} ::= \mathtt{class\ C\ extends\ C}\{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\}$$

Constructor Declaration

$$\mathtt{K} ::= \mathtt{C}\ (\bar{\mathtt{C}}\ \bar{\mathtt{f}})\ \{\mathtt{super}(\bar{\mathtt{f}}); \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}}\}$$

Method Declaration

$$\mathtt{M} ::= \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{return\ t};\}$$

Term

$$\mathtt{t} ::= \mathtt{x} \mid \mathtt{t.f} \mid \mathtt{t.m}(\bar{\mathtt{t}}) \mid \mathtt{new\ C}(\bar{\mathtt{t}}) \mid (\mathtt{C})\ \mathtt{t}$$

Figure 5. Syntax of Featherweight Java

Figure 5 shows the syntax of FJ. The class declaration

$$\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\}$$

introduces a class named `C` with superclass `D`. The class has fields $\bar{\mathtt{f}}$ with types $\bar{\mathtt{C}}$, a single constructor `K`, and a suite of methods $\bar{\mathtt{M}}$.

In the formal notations, we use the bar notation adopted by Pierce [28] for repetitive elements: $\bar{a}$ to indicate a vector a, and all operations defined on single values expand componentwisely to vectors. For example, let x_i be the ith element in $\bar{x}$, we have $\bar{a} < \bar{b}$ is equal to $\forall i.\ a_i < b_i$ and $\bar{a} \in S$ is equal to $\forall i.\ a_i \in S$. Here, we write $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$, for $\mathtt{C}_1\ \mathtt{f}_1, \cdots, \mathtt{C}_n\ \mathtt{f}_n$, where $\mathtt{n}$ is the length of $\bar{\mathtt{C}}$ and $\bar{\mathtt{f}}$. Similarly, $\bar{\mathtt{M}}$ denotes $\mathtt{M}_1 \cdots \mathtt{M}_n$.

The constructor declaration

$$\mathtt{C}\ (\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\}$$

defines the way to initialize a Java object, including a call to superclass constructor and assignments to class fields.

The method declaration

$$\mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\ \mathtt{return\ t};\ \}$$

introduces a method named `m` with return type `C` and parameters $\bar{\mathtt{x}}$ of types $\bar{\mathtt{C}}$. The body of the method is just a single term `return t`.

There are only five terms in FJ, variable `x`, field access `t.f`, method invocation $\mathtt{t.m}(\bar{\mathtt{t}})$, object creation $\mathtt{new\ C}(\bar{\mathtt{e}})$, and cast operation $(\mathtt{C})\mathtt{e}$. The key simplification in FJ is the omission of assignment. This implies that an object's field is initialized by its constructor and never changed afterwards. This restricts FJ to a "functional" fragment of Java.

The typing rules of FJ are the same as those of plain Java. One exception is that FJ does not support method overloading. We refer the reader to the original paper [7] and Appendix A for the typing rules.

3.2 Core SWIN

In this subsection we describe the syntax and evaluation rules of SWIN formally. The type checking rules and the proof of the type-safety property will be presented in Section 4 later.

3.2.1 Syntax

The formal definition of SWIN is presented in Figure 6. Similar to Twinning, a SWIN program Π is a set of transformation rules, and each transformation rule ($\pi = (\bar{\mathtt{d}})\ [\mathtt{l} : \mathtt{C}_l \ \rightarrow\ \mathtt{r} : \mathtt{C}_r]$) consists of three parts: 1) meta variable declarations ($\bar{\mathtt{d}}$), 2) left hand side source code pattern `l` and 3) right hand side target code pattern `r`. The source code pattern `l` will be used to match an expression in

Π	$::=$	$\{\bar{\pi}\}$	Transformation program
π	$::=$	$(\bar{d})\,[\mathtt{l} \rightarrow \mathtt{r}]$	Transformation rule
$\mathtt{d}$	$::=$	$\mathtt{x} : \mathtt{C}_1 \hookrightarrow \mathtt{C}_2$	Variable declaration
$\mathtt{l}$	$::=$	$\mathtt{x.f} \mid \mathtt{new\ C}(\bar{\mathtt{x}}) \mid \mathtt{x.m}(\bar{\mathtt{x}})$	Code pattern
$\mathtt{r}$	$::=$	$\mathtt{t}$	FJ term

Figure 6. Syntax of SWIN

an old client program, and the target code pattern r is an FJ term using a new API with meta-variables bounded in d, which is used to generate updated client code. And the variable declaration part ($\mathtt{d} = \mathtt{x} : \mathtt{A} \hookrightarrow \mathtt{B}$) associates a metavariable with its type migration information: x is of type A in l and of type B in r.

An informal explanation of the rule can be seen from its correspondence with the replacement rule in Section 2. For example, the mapping rule

$$\pi = (\mathtt{x} : \mathtt{A} \hookrightarrow \mathtt{L}, \mathtt{y} : \mathtt{B} \hookrightarrow \mathtt{M})\ [\, \mathtt{x.m(y)} : \mathtt{C} \rightarrow \mathtt{x.h(y)} : \mathtt{D}\,]$$

can be seen as the following replacement rule:

```
[
 C (A x, B y) { return x.m(y); }
 D (L x, M y) { return x.h(y); }
]
```

Now if there is a client source code term $(\mathtt{new\ A}()).\mathtt{m}(\mathtt{new\ B}())$, the rule will match the term as x binds to new A(), y binds to new B(), and the method name m matches the method name in the term. It results in that the updated term $(\mathtt{new\ A}()).\mathtt{h}(\mathtt{new\ B}())$ is of type D. Note that this rule does not match the term $(\mathtt{new\ C}()).\mathtt{m}(\mathtt{new\ B}())$, as the type of the variable x (type A) does not match the type of the term new C() (type C).

To ensure convergence, we do not allow the left hand sides of two rules to be the same. If two rules have the same left hand side, they will always match the same term, resulting in possibly divergent results.

3.2.2 Semantics: Evaluation Rules

We assume that an FJ program, which is a set of class declarations, can be divided into two parts: $\{\overline{\mathtt{CL}}_{\mathtt{API}}\}$ and $\{\overline{\mathtt{CL}}_{\mathtt{client}}\}$, where $\{\overline{\mathtt{CL}}_{\mathtt{API}}\}$ is the source API, consisting of class definitions that are type-correct by themselves; $\{\overline{\mathtt{CL}}_{\mathtt{client}}\}$ is the client program to be transformed, consisting of class definitions that depends on $\{\overline{\mathtt{CL}}_{\mathtt{API}}\}$. A transformation on an FJ program is to apply the transformation rules on $\{\overline{\mathtt{CL}}_{\mathtt{client}}\}$ to get $\{\overline{\mathtt{CL}}'_{\mathtt{client}}\}$, and then replace $\{\overline{\mathtt{CL}}_{\mathtt{API}}\}$ with the target API $\{\overline{\mathtt{CL}}'_{\mathtt{API}}\}$, such that $\{\overline{\mathtt{CL}}'_{\mathtt{API}}\}$ and $\{\overline{\mathtt{CL}}'_{\mathtt{client}}\}$ form a type-correct program.

In formal notations, we use API to denote $\{\ \overline{\mathtt{CL}}_{\mathtt{API}}\ \}$, and operations on APIs are naturally set operations (e.g. $\mathtt{API}_1 - \mathtt{API}_2$ is set substraction, which excludes class declarations in $\mathtt{API}_2$ from $\mathtt{API}_1$). In particular, we use the notation $\mathtt{API}_\mathtt{s}$ to denote the source API, $\overline{\mathtt{CL}}_{\mathtt{API}}$, and $\mathtt{API}_\mathtt{d}$ to denote the target API, $\overline{\mathtt{CL}}'_{\mathtt{API}}$, respectively.

Figure 7 summarizes the formal semantics of SWIN. In the rules, $\mathtt{A} <: \mathtt{B}$ indicates that A is a subtype of B. A transformation program Π is formalized as a transformation from source code to target code on both types and terms. This transformation consists of the following three steps.

1. *Transformation Promotion*: The first three rules (E-DECLARATION, E-CONSTRUCTOR, E-METHOD) are used to promote Π up to types and terms through a class declaration, a construction definition, and a method definition, respectively.
2. *Type Transformation*: The next E-CLASS rule is used to transform source types in the source API to target types in target API based on the type mappings defined in Π. Those types which are not involved in the type mapping of Π will stay the same according to the rule E-ALTER-CLASS. An important components of the two rules is **TypeMapping**, which records how types in $\mathtt{API}_\mathtt{s}$ is mapped to $\mathtt{API}_\mathtt{d}$ by the transformation program, and it is defined in Figure 8.
3. *Term Transformation*: The rest of the rules are used to transform source code terms. As the syntactic definitions in Figure 6 show, an FJ term takes five forms. The form x and (C)t are evaluated by E-T-VAR and E-T-CAST, respectively, which basically further applies Π to sub terms. The other three forms are handled by E-T-FIELD, E-T-NEW, E-T-INVOKE, respectively. The three evaluation rules apply matched SWIN transformation rules to the current term. A term is matched by a rule when the signature of the rule matches the term and there is no rule that more "closely" matches the term. A rule r more closely matches a term than another rule r' when any of the meta variable in r has closer type to the matched term than r'. To deal with client defined classes, evaluation rules E-ALTER-FIELD, E-ALTER-INVOKE, E-ALTER-NEW are designed and they will apply Π to sub terms. In the definitions, we use $\mathbf{Type}(\mathtt{t})$ to get the type of a term t based on FJ typing rules.

To be concrete, let us see an example. Suppose that we want to switch from the old API ($\mathtt{API}_\mathtt{s}$) to a new one ($\mathtt{API}_\mathtt{d}$)[3]

$$\mathtt{API}_\mathtt{s} = \{\mathtt{class\ A}\ \{\ \mathtt{A}()\{\ldots\};\ \mathtt{A\ h(A\ a)}\{\ldots\};\};\}$$
$$\mathtt{API}_\mathtt{d} = \{\mathtt{class\ B}\ \{\ \mathtt{B}()\{\ldots\};\ \mathtt{B\ k(B\ b, B\ c)}\{\ldots\};\};\}$$

and we use the following SWIN transformation program

$$\begin{aligned}
&\Pi = [\pi_1, \pi_2] \\
&\mathbf{where} \\
&\quad \pi_1 = ()\ [\, \mathtt{new\ A}() : \mathtt{A} \rightarrow \mathtt{new\ B}() : \mathtt{B}\,] \\
&\quad \pi_2 = (\mathtt{x} : \mathtt{A} \hookrightarrow \mathtt{B}, \mathtt{u} : \mathtt{A} \hookrightarrow \mathtt{B}) \\
&\qquad\qquad [\, \mathtt{x.h(u)} : \mathtt{A} \rightarrow \mathtt{x.k(u, new\ B())} : \mathtt{B}\,]
\end{aligned}$$

to transform the following source client Java code.

$$(\mathtt{new\ A}()).\mathtt{h}(\mathtt{new\ A}())$$

The transformation is done as follows:

$$\begin{aligned}
& \Pi((\mathtt{new\ A}()).\mathtt{h}(\mathtt{new\ A}()) \\
= & \quad \{ \text{ by E-T-INVOKE with rule } \pi_2 \ \} \\
& [\mathtt{x} \rightarrow \Pi(\mathtt{new\ A}()), \mathtt{u} \rightarrow \Pi(\mathtt{new\ A}())](\mathtt{x.k}(\mathtt{u}, \mathtt{new\ B}())) \\
= & \quad \{ \text{ replace } \mathtt{x} \text{ and } \mathtt{u} \text{ in } \mathtt{x.k(u, new\ B())} \ \} \\
& \Pi(\mathtt{new\ A}()).\mathtt{k}(\Pi(\mathtt{new\ A}()), \mathtt{new\ B}()) \\
= & \quad \{ \text{ by E-T-NEW with rule } \pi_1 \ \} \\
& [\,](\mathtt{new\ B}()).\mathtt{k}([\,](\mathtt{new\ B}()), \mathtt{new\ B}()) \\
= & \quad \{ \text{ since } [\,](\mathtt{new\ B}()) = \mathtt{new\ B}() \ \} \\
& \mathtt{new\ B}().\mathtt{k}(\mathtt{new\ B}(), \mathtt{new\ B}())
\end{aligned}$$

Thus it results in the target code $\mathtt{new\ B}().\mathtt{k}(\mathtt{new\ B}(), \mathtt{new\ B}())$.

4. Type Checking System for Core SWIN

Now we turn to our type system that is used to check the type safety of transformation programs in SWIN. Given two APIs ($\mathtt{API}_\mathtt{s}$ and $\mathtt{API}_\mathtt{d}$), and a transformation program (Π), mapping from $\mathtt{API}_\mathtt{s}$ to $\mathtt{API}_\mathtt{d}$, if Π passes our type checking, we can guarantee that Π will transform *any* FJ program using $\mathtt{API}_\mathtt{s}$ to a well-typed FJ program using $\mathtt{API}_\mathtt{d}$ instead.

[3] We omit the API method bodies here as it is not necessary to see the details of how an API method is implemented; it is sufficient to show the input types and the return type of each method in API. And this kind of omission is also used in later sections.

$$\frac{\mathtt{CL} = \mathtt{class}\ \mathtt{C}_1\ \mathtt{extends}\ \mathtt{C}_2\ \{\ \bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\ \}}{\Pi(\mathtt{CL}) = \mathtt{class}\ \Pi(\mathtt{C}_1)\ \mathtt{extends}\ \Pi(\mathtt{C}_2)\ \{\ \Pi(\bar{\mathtt{C}})\ \bar{\mathtt{f}};\ \Pi(\mathtt{K})\ \overline{\Pi(\mathtt{M})}\ \}}\ \text{(E-DECLARATION)}$$

$$\frac{\mathtt{K} = \mathtt{C}_1\ (\bar{\mathtt{C}}_2\ \bar{\mathtt{f}}_2)\ \{\mathtt{super}(\bar{\mathtt{f}}_3);\ \mathtt{this}.\bar{\mathtt{f}}_\mathtt{i} = \bar{\mathtt{f}}_\mathtt{j}\}}{\Pi(\mathtt{K}) = \Pi(\mathtt{C}_1)\ (\Pi(\bar{\mathtt{C}}_2)\ \bar{\mathtt{f}}_2)\ \{\mathtt{super}(\bar{\mathtt{f}}_3);\ \mathtt{this}.\bar{\mathtt{f}}_\mathtt{i} = \bar{\mathtt{f}}_\mathtt{j}\}}\ \text{(E-CONSTRUCTOR)}$$

$$\frac{\mathtt{M} = \mathtt{C}_1\ \mathtt{m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{t};\}}{\Pi(\mathtt{M}) = \Pi(\mathtt{C}_1)\ \mathtt{m}(\Pi(\bar{\mathtt{C}})\ \bar{\mathtt{x}})\ \{\mathtt{return}\ \Pi(\mathtt{t});\}}\ \text{(E-METHOD)} \qquad \frac{\mathtt{C}_0 \hookrightarrow \mathtt{C}_1 \in \mathbf{TypeMapping}(\Pi)}{\Pi(\mathtt{C}_0) = \mathtt{C}_1}\ \text{(E-CLASS)}$$

$$\frac{\forall \mathtt{C}.\ \mathtt{C}_0 \hookrightarrow \mathtt{C} \notin \mathbf{TypeMapping}(\Pi)}{\Pi(\mathtt{C}_0) = \mathtt{C}_0}\ \text{(E-ALTER-CLASS)} \qquad \frac{}{\Pi(\mathtt{x}) = \mathtt{x}}\ \text{(E-T-VAR)}$$

$$\frac{\begin{array}{c}(\mathtt{x} : \mathtt{C}_1 \hookrightarrow \mathtt{C}_2)[\ \mathtt{x.f} : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}\ ] \in \Pi \qquad \mathbf{Type}(\mathtt{t}) <: \mathtt{C}_1 \\ \neg\exists\ (\mathtt{x} : \mathtt{C}_3 \hookrightarrow \mathtt{C}_4)[\ \mathtt{x.f} : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}\ ] \in \Pi.(\mathbf{Type}(\mathtt{t}) <: \mathtt{C}_3 <: \mathtt{C}_1 \wedge \mathtt{C}_3 \neq \mathtt{C}_1)\end{array}}{\Pi(\mathtt{t.f}) = [\ \mathtt{x} \mapsto \Pi(\mathtt{t})\ ]\mathtt{r}}\ \text{(E-T-FIELD)}$$

$$\frac{}{\Pi((\mathtt{C})\ \mathtt{t}) = (\Pi(\mathtt{C}))\ \Pi(\mathtt{t})}\ \text{(E-T-CAST)} \qquad \frac{\begin{array}{c}(\bar{\mathtt{d}})[\ \mathtt{new}\ \mathtt{C}_0(\ \bar{\mathtt{x}}\ ) : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}] \in \Pi \\ \{\ \bar{\mathtt{x}} : \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_2}\ \} \subseteq \bar{\mathtt{d}} \qquad \mathbf{Type}(\bar{\mathtt{t}}_\mathtt{u}) <: \bar{\mathtt{C}}_1\end{array}}{\Pi(\mathtt{new}\ \mathtt{C}_0(\bar{\mathtt{t}}_\mathtt{u})) = [\ \bar{\mathtt{x}} \mapsto \overline{\Pi(\mathtt{t}_\mathtt{u})}\ ](\mathtt{r})}\ \text{(E-T-NEW)}$$

$$\frac{\begin{array}{c}(\bar{\mathtt{y}} : \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_2},\ \mathtt{x}_0 : \mathtt{C}_3 \hookrightarrow \mathtt{C}_4)[\ \mathtt{x}_0.\mathtt{m}_0(\ \bar{\mathtt{y}}\ ) : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}] \in \Pi \\ \mathbf{Type}(\mathtt{t}_0) <: \mathtt{C}_3 \qquad \mathbf{Type}(\bar{\mathtt{t}}_\mathtt{u}) <: \bar{\mathtt{C}}_1 \\ \neg\exists\ (\bar{\mathtt{y}} : \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_2},\ \mathtt{x}_0 : \mathtt{C}_5 \hookrightarrow \mathtt{C}_6)[\ \mathtt{x}_0.\mathtt{m}_0(\ \bar{\mathtt{y}}\ ) : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}] \in \Pi.(\mathbf{Type}(\mathtt{t}_0) <: \mathtt{C}_5 <: \mathtt{C}_3 \wedge \mathtt{C}_5 \neq \mathtt{C}_3)\end{array}}{\Pi(\mathtt{t}_0.\mathtt{m}_0(\bar{\mathtt{t}}_\mathtt{u})) = [\ \mathtt{x}_0 \mapsto \Pi(\mathtt{t}_0),\ \bar{\mathtt{y}} \mapsto \overline{\Pi(\mathtt{t}_\mathtt{u})}\ ](\mathtt{r})}\ \text{(E-T-INVOKE)}$$

$$\frac{\text{no other inference rule can be applied}}{\Pi(\mathtt{new}\ \mathtt{C}_0(\bar{\mathtt{t}}_\mathtt{u})) = \mathtt{new}\ \mathtt{C}_0(\ \overline{\Pi(\mathtt{t}_\mathtt{u})}\ )}\ \text{(E-ALTER-NEW)}$$

$$\frac{\text{no other inference rule can be applied}}{\Pi(\mathtt{t}_0.\mathtt{m}_0(\bar{\mathtt{t}}_\mathtt{u})) = \Pi(\mathtt{t}_0).\mathtt{m}(\ \overline{\Pi(\mathtt{t}_\mathtt{u})}\ )}\ \text{(E-ALTER-INVOKE)}$$

$$\frac{\text{no other inference rule can be applied}}{\Pi(\mathtt{t.f}) = \Pi(\mathtt{t}).\mathtt{f}}\ \text{(E-ALTER-FIELD)}$$

Figure 7. Evaluation Rules of SWIN

$$\mathbf{TypeMapping}((\ \bar{\mathtt{x}} : \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_2}\ )[\mathtt{l} : \mathtt{C}\ \rightarrow\ \mathtt{r} : \mathtt{D}]) = \{\mathtt{C} \hookrightarrow \mathtt{D}\} \cup \{\ \overline{\mathtt{C}_1 \hookrightarrow \mathtt{C}_2}\ \}$$

$$\mathbf{TypeMapping}(\{\bar{\pi}\}) = \bigcup_{\pi} (\mathbf{TypeMapping}(\pi)) \qquad \text{(Extract type migration information)}$$

$$\mathbf{Decl}(\mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{...\}) = \mathtt{C} \qquad \text{(Extract the declared class name)}$$

Figure 8. Auxiliary Functions used in Figure 7 and Figure 10

In the following sections, we will define our type-checking rules and prove the type-safety property of SWIN.

4.1 Type Checking Rules

We present the rules in Figure 9 and Figure 10. Figure 9 depicts the rule for checking a single transformation rule π. Figure 10 depicts the rules for checking a transformation program Π.

Checking Rule for π This rule checks whether the types declared in a transformation rule conforms to the actual types inferred using FJ typing rules. In the formal notation, we use $\Gamma \vdash^{\mathtt{API_s}}_{\mathtt{FJ}} \mathtt{t} : \mathtt{C}$ to denote that the term $\mathtt{t}$ has type $\mathtt{C}$ under context Γ by FJ typing rules when considered together with $\mathtt{API_s}$. When checking the left hand sides, we introduce modified type checking rules $*\mathtt{FJ}$, which is used to ensure the exact matching on the parameters, so that our rules are only declared on valid methods.

$$
\begin{array}{l}
\mathbf{RuleOK}(\Pi) = \forall\, \pi.(\pi \in \Pi \Rightarrow \pi\ ok)\\
\mathbf{ConstrCover}(\Pi, \mathtt{API_s}, \mathtt{API_d}) =\\
\quad \forall\, \mathtt{C_1}, \bar{\mathtt{C}}.(\mathtt{class\ C_1\ extends}\ \_\ \{\mathtt{C_1}(\bar{\mathtt{C}}\ \bar{\_})\ ...\ \} \in (\mathtt{API_s} - \mathtt{API_d})\\
\qquad \Rightarrow\ \exists\, \mathtt{C_2}, \bar{\mathtt{C}}', \bar{\mathtt{x}}, \mathtt{r}.((\ \bar{\mathtt{x}} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}'}\ )[\mathtt{new\ C_1}(\bar{\mathtt{x}}) : \mathtt{C_1} \to \mathtt{r} : \mathtt{C_2}] \in \Pi))\\
\mathbf{MethCover}(\Pi, \mathtt{API_s}, \mathtt{API_d}) =\\
\quad \forall\, \mathtt{C_1}, \mathtt{C_2}, \mathtt{m}, \bar{\mathtt{C}}.(\mathtt{class\ C_1\ extends}\ \_\ \{\ \mathtt{C_2\ m}(\ \bar{\mathtt{C}}\ \bar{\_}\ )\{...\}\ ...\ \} \in (\mathtt{API_s} - \mathtt{API_d})\\
\qquad \Rightarrow\ \exists\, \mathtt{x}, \bar{\mathtt{y}}, \mathtt{C_1'}, \mathtt{C_2'}, \bar{\mathtt{C}}', \mathtt{r}.((\mathtt{x} : \mathtt{C_1} \hookrightarrow \mathtt{C_1'},\ \ \bar{\mathtt{y}} : \overline{\mathtt{C} \hookrightarrow \mathtt{C}'}\ )[\mathtt{x.m}(\bar{\mathtt{y}}) : \mathtt{C_2} \to \mathtt{r} : \mathtt{C_2'}] \in \Pi))\\
\mathbf{FieldCover}(\Pi, \mathtt{API_s}, \mathtt{API_d}) =\\
\quad \forall\, \mathtt{C_1}, \mathtt{C_2}, \mathtt{f}.(\mathtt{class\ C_1\ extends}\ \_\ \{\mathtt{C_2\ f};...\} \in (\mathtt{API_s} - \mathtt{API_d})\\
\qquad \Rightarrow\ \exists\, \mathtt{x}, \mathtt{C_1'}, \mathtt{C_2'}.((\mathtt{x} : \mathtt{C_1} \hookrightarrow \mathtt{C_1'}\ )[\mathtt{x.f} : \mathtt{C_2} \to \mathtt{r} : \mathtt{C_2'}] \in \Pi))\\
\mathbf{MapChecking}(\Pi, \mathtt{API_s}, \mathtt{API_d}) =\\
\quad \forall\, \mathtt{C}, \mathtt{D}.(\mathtt{C} \hookrightarrow \mathtt{D} \in \mathbf{TypeMapping}(\Pi)\\
\qquad \Rightarrow\ (\exists\, \mathtt{CL} \in \mathtt{API_s} \cap \mathtt{API_d}.(\mathbf{Decl}(\mathtt{CL}) = \mathtt{C} \wedge \mathtt{D} = \mathtt{C}))\\
\qquad\quad \vee(\exists\, \mathtt{CL} \in \mathtt{API_s} - \mathtt{API_d}.(\mathbf{Decl}(\mathtt{CL}) = \mathtt{C})))\\
\mathbf{Subtyping}(\Pi, \mathtt{API_s}, \mathtt{API_d}) =\\
\quad \forall\, \mathtt{C_i}, \mathtt{D_i}, \mathtt{C_j}, \mathtt{D_j}.(\mathtt{C_i} \hookrightarrow \mathtt{D_i}, \mathtt{C_j} \hookrightarrow \mathtt{D_j} \in \mathbf{TypeMapping}(\Pi)\ \ \Rightarrow\ \ (\mathtt{C_i} <: \mathtt{C_j} \Rightarrow \mathtt{D_i} <: \mathtt{D_j}))
\end{array}
$$

Figure 10. Checking rules (or checking funtions) for Π. A SWIN program Π with specified source API ($\mathtt{API_s}$) and destination API ($\mathtt{API_d}$) should pass these checking rules to maintain type safety. Underscore(_) is a wildcard and apostrophe (...) represents omitted declaration sequences (field declarations or method declarations). And a special use of the notations used in **ConstrCover** etc. is $\bar{\mathtt{C}}\ \bar{\_}$, which represents $\mathtt{C_1}\ \_$, $\mathtt{C_2}\ \_$, ..., $\mathtt{C_n}\ \_$, as their types are known while the variable names are not necessary.

$$\frac{\{\ \bar{\mathtt{x}} : \bar{\mathtt{C}}\ \} \vdash^{\mathtt{API_s}}_{*\mathtt{FJ}} \mathtt{l} : \mathtt{C_1} \qquad \{\ \bar{\mathtt{x}} : \bar{\mathtt{D}}\ \} \vdash^{\mathtt{API_d}}_{\mathtt{FJ}} \mathtt{r} : \mathtt{C_2}}{(\bar{\mathtt{x}} : \overline{\mathtt{C} \hookrightarrow \mathtt{D}})[\mathtt{l} : \mathtt{C_1} \to \mathtt{r} : \mathtt{C_2}]\ ok} \ \text{(T-}\pi\text{)}$$

$$\frac{\Gamma \vdash^{\mathtt{API}}_{\mathtt{FJ}} \mathtt{x} : \mathtt{C_0} \quad \mathtt{mtype}(\mathtt{m}, \mathtt{C_0}) = \bar{\mathtt{D}} \to \mathtt{C} \quad \Gamma \vdash^{\mathtt{API}}_{\mathtt{FJ}} \bar{\mathtt{y}} : \bar{\mathtt{D}}}{\Gamma \vdash^{\mathtt{API}}_{*\mathtt{FJ}} \mathtt{x.m}(\bar{\mathtt{y}}) : \mathtt{C}} \ \text{(T-L1)}$$

$$\frac{\mathtt{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \quad \Gamma \vdash^{\mathtt{API}}_{\mathtt{FJ}} \bar{\mathtt{x}} : \bar{\mathtt{D}}}{\Gamma \vdash^{\mathtt{API}}_{*\mathtt{FJ}} \mathtt{new\ C}(\bar{\mathtt{x}}) : \mathtt{C}} \ \text{(T-L2)}$$

$$\frac{\mathtt{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \quad \Gamma \vdash^{\mathtt{API}}_{\mathtt{FJ}} \mathtt{x} : \mathtt{C}}{\Gamma \vdash^{\mathtt{API}}_{*\mathtt{FJ}} \mathtt{x.f_i} : \mathtt{D_i}} \ \text{(T-L3)}$$

Figure 9. The checking rule for π

Please note that this rule also indicates that we can drop the type declarations in the transformation rules, i.e., instead of writing $[\ \mathtt{x.m(y)} : \mathtt{C} \to \mathtt{x.h(y)} : \mathtt{D}\ ]$, we can write $[\ \mathtt{x.m(y)} \to \mathtt{x.h(y)}\ ]$ and deduce $\mathtt{C}$ and $\mathtt{D}$ using FJ typing rules. However, we decide to keep these declarations in the code because with these declarations, **TypeMapping**(Π) becomes more explicit, avoiding subtle bugs on erroneous type mappings.

Checking Rules for Π The main goal of the type checking rules is to check the four conditions presented in Section 2. Next we explain how this is achieved.

1. All rules are well-typed themselves. (Rule **RuleOK**)
2. The class mapping in **TypeMapping**(Π) should be a function, i.e. one class in the old API should be mapped to only one class in new API. In fact, this property is covered by the subtyping relationship check, as type equality can be treated as a bi-directional subtyping relation. (Rule **Subtyping**)
3. The class transformation preserves the subtyping relationship in the old API. (Rule **Subtyping**)
4. The transformation program covers all classes/methods/constructors/fields that only exist in the old API but not the new API (Rules **ConstrCover**, **MethCover**, **FieldCover**), and no unnecessarily type conversion is introduced (Rule **MapChecking**). Note that the above three rules are declared on constructors, methods, and fields directly. The coverage of classes is implied by rule **ConstrCover** and the definition of **TypeMapping**.

We say a SWIN program is *well-typed* iff it satisfies the checking rules presented in Figure 10. As will be proved in Section 4.3, a well-typed transformation program Π is type-safe, guaranteeing the well-typedness of the target code when Π is applied to any client code with old API. Otherwise, there must exist some client code that cannot be transformed to a well-typed target code with this transformation program.

4.2 Convergence Theorem

Our checking rules and evaluation rules ensure the convergence of any SWIN program, which is discussed in the following theorem and its proof sketch.

Theorem 1. *Any SWIN program is convergent.*

Proof sketch. SWIN employs a normal order evaluation semantics. First, the evaluation rules visit a term leftmost and outermost. After performing the transformation on that term, the evaluation rules recursively visit the sub terms of the term, and for each visit, the transformation will be applied on the original sub terms, and produce the transformation result by combining the transformed sub terms. In this way we can ensure each recursive visit will be terminated as the length of the sub terms are always shorter than the term. Also, we can ensure the transformation on a term is confluent, as each program element is transformed by exactly one rule according to the restrictions on π (checking rules for π) and the definitions of the evaluation rules. □

4.3 Type-Safety Theorem

In this subsection, we reason type safety of SWIN formally and outline the key theorems and lemmas here.

Intuitively, SWIN is type-safe if and only if a well-typed SWIN program can transform *any* well-typed FJ program to a well-typed FJ program. The proof needs to bridge the type inference tree on an old API to the new type inference tree on a new API, and we need to generate a derivation tree based on conditions in checking rules and the derivation tree on original client code.

Because of the space limit, we cannot present the full proofs here. Instead, we present four key lemmas that can stepwise lead to the final theorem. The full proofs of lemmas and the theorem can be found in the technical report on the formal definition of SWIN [23].

In our lemmas, $\Gamma = \bar{\mathtt{x}} : \bar{\mathtt{C}}$ represents the typing context of an FJ term $\mathtt{t}$, which designates each variable $\mathtt{x}$ in the term with a type $\mathtt{C}$. Specially, given a term $\mathtt{t}$ in client code and a transformation program Π, $\Gamma_\mathtt{s}$ represents the variable environment for $\mathtt{t}$ (before transformation) and $\Gamma_\mathtt{d}$ represents the environment of the transformed term $\Pi(\mathtt{t})$. The proof also depends on the typing rules of FJ, which is presented in Appendix A.

Lemma 1 (Typing Context). *Suppose the typing context for a term* $\mathtt{t}$ *is* $\Gamma_\mathtt{s} = \bar{\mathtt{x}} : \bar{\mathtt{C}}$. *Given a SWIN program* Π *acting on* $\mathtt{API_s}$ *to* $\mathtt{API_d}$, *we know that the typing context for* $\Pi(\mathtt{t})$ *is* $\Gamma_\mathtt{d} = \bar{\mathtt{x}} : \overline{\Pi(\mathtt{C})}$.

Proof sketch. Note that an FJ typing context Γ will be created in the rule FJ-M-OK and will not change during the type deriving of a term. According to the rule E-DELCARATION and E-METHOD, the types of the method argument and the variable "`this`" (which include all variables binded in Γ) will both be updated to $\Pi(\mathtt{C})$. □

Lemma 2 (Subtyping). *Suppose a well-typed SWIN program* Π *transforms an FJ program* P *with* $\mathtt{API_s}$ *to a new program* P' *with* $\mathtt{API_d}$, *then the following holds.*
$\mathtt{C_1} <: \mathtt{C_2}$ *in* $P \implies \Pi(\mathtt{C_1}) <: \Pi(\mathtt{C_2})$ *in* P'.

Proof sketch. The subtype relation between classes have the following two cases:

- $\mathtt{C_1}$ is declared in client code: E-DECLARATION will guarantee that the subtype relation will be preserved in transformation.
- $\mathtt{C_1}$ is declared in API: the checking rule **Subtyping** guarantees it.

Combining these two cases and the transitivity of subtype relation, we know that the lemma holds. □

Lemma 3 (Variable Substitution). *Suppose that an FJ term* $\mathtt{t}$ *is well-typed under context* $\Gamma = \Gamma_1, \{\bar{\mathtt{x}} : \bar{\mathtt{C}}_\mathtt{x}\}$, *i.e.* $\Gamma \vdash_{\mathtt{FJ}} \mathtt{t} : \mathtt{C_t}$. *After substituting terms* $\bar{\mathtt{t}}_\mathtt{u}$ *for variables* $\bar{\mathtt{x}}$, *with the property that* $\Gamma_1 \vdash_{\mathtt{FJ}} \bar{\mathtt{t}}_\mathtt{u} : \bar{\mathtt{C}}_\mathtt{u}$ *and* $\bar{\mathtt{C}}_\mathtt{u} <: \bar{\mathtt{C}}_\mathtt{x}$, $\mathtt{t}$ *can be typed to* $\mathtt{C_t}$ *or a sub-class of* $\mathtt{C_t}$. *Namely,*

$$\Gamma_1, \{\bar{\mathtt{x}} : \bar{\mathtt{C}}_\mathtt{x}\} \vdash_{\mathtt{FJ}} \mathtt{t} : \mathtt{C_t} \Longrightarrow \Gamma_1 \vdash_{\mathtt{FJ}} [\bar{\mathtt{x}} \mapsto \bar{\mathtt{t}}_\mathtt{u}]\mathtt{t} : \mathtt{C}'_\mathtt{t},\ \mathtt{C}'_\mathtt{t} <: \mathtt{C_t}$$

Proof sketch. By induction on the derivation of a term $\mathtt{t}$, we have fives cases to discuss. ($\mathtt{x}$, $(\mathtt{C})\mathtt{t}$, $\mathtt{t.f}$, $\mathtt{new\ C}(\bar{\mathtt{t}})$ and $\mathtt{t.m}(\bar{\mathtt{t}})$). The first three cases ($\mathtt{x}$,$(\mathtt{C})\mathtt{t}$ and $\mathtt{t.f}$) are obvious according to their evaluation rules.

For case 4 and case 5, the following properties are used in proof:

- The arguments in the method invocation will be substitute by terms whose types are subtypes of the original argument variables (Arguments are compatible).
- The target term (the caller) is of a type that is subtype to the original caller variable (The method can be found in the new caller term).

With subtype relation cleared, the proof is also obvious according to the rule FJ-METHOD and FJ-CONSTRUCTOR. □

Lemma 4 (Term Formation). *Given a well-typed SWIN program* Π, *if a term* $\mathtt{t}$ *in the original typing context can be typed to* $\mathtt{C}$, *then after transformation by* Π, *the term is well-typed and its type is a subtype of* $\Pi(\mathtt{C})$. *i.e.*

$$\Gamma_\mathtt{s} \vdash^{\mathtt{API_s}}_{\mathtt{FJ}} \mathtt{t} : \mathtt{C} \Longrightarrow \Gamma_\mathtt{d} \vdash^{\mathtt{API_d}}_{\mathtt{FJ}} \Pi(\mathtt{t}) : \mathtt{C}',\ \mathtt{where}\ \mathtt{C}' <: \Pi(\mathtt{C})$$

Proof sketch. Induction on the term derivation. Again we have five cases to prove. ($\mathtt{x}$, $(\mathtt{C})\mathtt{t}$, $\mathtt{t.f}$, $\mathtt{new\ C}(\bar{\mathtt{t}})$ and $\mathtt{t.m}(\bar{\mathtt{t}})$)

The first two cases ($\mathtt{x}$, $(\mathtt{C})\mathtt{t}$) are obvious according to Lemma 1 and their evaluation rules (E-T-VAR, E-T-CAST). The last three cases are not trivial in proof, we simply mention some points for case 5 (method invocation) as an example, and the full proof can be found in the technical report [23].

For case $\mathtt{t} = \mathtt{t_0.m}(\bar{\mathtt{t}}_\mathtt{u})$, we have two subcases to deal with:

- The method is defined in a class which is defined in client code: to prove that arguments and the caller terms are well-formed terms whose types are subtypes of the original ones.
- The method is defined in a class defined in old API: to prove that the rule π to transform the term will finally leads to a well-typed term according to the Substitution Lemma and Subtyping Lemma.

And with these five cases proved, we have the property that a well-typed SWIN program can correctly transform FJ terms. □

Theorem 2 (Type-Safety). *Any FJ program is well-typed after a transformation by a well-typed SWIN program* Π. *i.e. For any* $\mathtt{CL}$,

$$\Pi(\mathtt{CL}) = \mathtt{class}\ \Pi(\mathtt{C_1})\ \mathtt{extends}\ \Pi(\mathtt{C_2})\ \{\ \Pi(\bar{\mathtt{C}}_\mathtt{i})\ \bar{\mathtt{f}}_\mathtt{i};\ \Pi(\mathtt{K})\ \overline{\Pi(\mathtt{M})}\ \}$$

is well-typed with new API if Π *is well-typed.*

Proof sketch. We need to prove that method calls are well formed in the transformed FJ program and the class declarations are well formed.

This can be a direct result from: 1) all terms are well formed after transforamtion (Lemma 4), 2) arguments and super class declarations are well formed (this can be checked through E-METHOD-DECLARATION, E-CLASS-DECLARATION and **TypeMapping**). □

5. From Core SWIN to Full SWIN

In this section, we present the way to extend core SWIN on Featherweight Java to full SWIN on full Java language formally. Generally, the extension is based on the term extension and type extension. By extending source code pattern and target code pattern to a term in full Java and extending types to full Java in variable declaration part of update rules, we are able to match a Java term and then transform it to a term with new API by meta-variable substitution.

Extending SWIN to full SWIN, we need some special treatments of the following key points :

Package Full Java supports the `package` and `import` commands for name organization. Pacakges support modularity naturally and APIs in a full Java program should be stored in different packages. When we transform Java programs with packages, we simply need to transform the client codes and then replace the corresponding API packages without touching other API packages.

To ease the writing of transformation rules, we also support `import` command in SWIN, yet all internal processing is based on fully qualified names.

Field and Assignment FJ has no assignment statements and all fields are read-only. When assignments are introduced, expressions in Java can be distinguished into L-value and R-value. To ensure type safety, we need to ensure the transformation does not change an L-value into an R-value. The most common L-value is field access. For example, given "`a.x = b`", if a transformation rule transforms "`a.x`" into "`new A()`", the new code will fail to compile because "`new A()`" is not a L-value. This check can be implemented by applying the Java rules for distinguishing L-value and R-value on the source patterns and the destination patterns.

Static Method Access In full Java, a method can be defined as a static method, and we can access it by `C.m(a, b, ...)`. We treat the application of full SWIN on static method access as a normal method invokation, except that we need to apply the term directly on the class identifier. As the transformation of a class definition is by class name replacement, type safety can be guaranteed.

Interface In FJ, the subtyping relation is linear and full order. So during pattern matching, there is always a "closest" parent class for each class and this ensure that the term will always match that class to ensure confluent.

In full Java, there may exist multiple parent classes, which will then lead to no single "closest" parent class exists. For example, if class `A` has two super types, class `B` and interface `C`, and all three classes declare a method `m`. If there are two transformation rules declared on `B.m` and `C.m`, respectively, we cannot find a closest rule on `A.m`. In such case, we will report an error when executing the transformation, and the programmers could resolve the error by adding a rule on `A.m`, which is always closer than any other rules.

Overload When method overloading is considered, we need to match a method not only using its name, but also the type of its input parameters. Also, the subtyping relation should be considered in the same way as Java: when there are several overloaded methods that can be matched, we choose the one with the closest subtyping relation on the parameters. For example, if we have a relationship `A` <: `B` <: `C`, and in class `D`, we have methods `f(B x)` and `f(C x)`. Then `(new D()).f(new A())` is a call to the first method as they have a closer subtyping relationship. A pattern matching `f(C x)` should not match this term.

Generics Generics in full Java affects the evaluation rules E-CLASS and E-T-NEW. We have two extending rules to solve this problem.

1. During pattern matching, a rule matches a generic type without considering its type parameters.
2. After performing transformation on a generic type, the rules recursively visit the type parameters.

The type safety is guaranteed because we require the preservation of subtyping relation, and thus the constraints on generic parameters will not be broken. Note that our rules always treat the generic type and its parameters independently, and thus do not allow the change in the number of type parameters from the source API to the target API. This design choice keeps our language simple, and has a limited negative effect on expressiveness: we have never observe the change in the number of type parameters in practice.

6. Case Studies

6.1 Research Questions

Since SWIN puts two more conditions on the replacement rules than Twinning, a natural question to ask is whether these two additional conditions confine the expressiveness of the language. In other words, there are programs that can be written in Twinning but not in SWIN, but are these programs useful in practice? Furthermore, beyond Twinning, we also want to understand the expressiveness of SWIN in general. These considerations lead to two research questions.

1. Does the extra conditions confine the expressiveness of SWIN compared with Twinning?
2. In general, how much expressive is SWIN?

6.2 Study Setup

To answer these two research questions, we perform three case studies. To answer the first research question, we need to compare SWIN with Twinning. To do this, we repeat a case study in Twinning that migrate programs from Crimson v1.1.3[4] to dom4j v1.6.1[5]. Crimson and dom4j are both Java libraries for manipulating XML files, but Crimson is no longer supported. Thus, developers may want to migrate programs from Crimson to dom4j.

To answer the second research question, we perform two more case studies, one is about migration from one API to another API, the other one is to upgrade clients for incompatible API upgrade. More concretely, we chose the program migration from Twitter4J v4.0.1[6] to Sina Weibo Java API v2[7], and the client upgrade from Google Calendar API[8] v2 to v3. Twitter4J is a Java wrapper for the RESTful Twitter API. Sina Weibo is the Chinese counterpart of Twitter, and it provides an official Java library for accessing its web API. Google Calendar API is the official Java library for accessing the data in Google Calendar.

The two case studies of program migration (from Crimson to dom4j, from Twitter4J to Sina Weibo API) both involve large APIs, and it is difficult for us to cover the full APIs. In the case study from Crimson to dom4j, the Twinning authors [1] chose a client (log4j v1.2.14[9]) and only wrote transformations for the part of the API covered by the client. We followed the same step as their case study. In the case study from Twitter4J to Sina Weibo API, we consider three example clients on manipulating the timeline provided in the example directory in the Twitter4J source package, and cover only the part of the API used in these examples.

To perform the case studies, we implemented SWIN in Java using the Polyglot compiler framework [24]. Both our implementation and all evaluation data are available at the project web site[10].

6.3 Results

6.3.1 General Expressiveness

In total, we wrote 94 rules for the three case studies, each transforming a method call to the old API into an expression using the new API. Our rules cover 97% of the total API methods that needed to be transformed in the three case studies. This results indicate that, though our approach deals only with one-to-many mappings, it is able to perform a significant portion of program adaptation tasks in practice.

6.3.2 Comparison with Twinning

The only uncovered API changes are three method changes in Google Calendar API, consisting of 3% of the total API methods that needs to be transformed. In the three uncovered method changes, one method splits into several methods, and we need to decide which new method to replace the original one based on the calling context, which is not supported in SWIN.

[4] `http://xml.apache.org/crimson/`

[5] `http://www.dom4j.org/`

[6] `https://github.com/yusuke/twitter4j/`

[7] `https://code.google.com/p/weibo4j/`

[8] `https://developers.google.com/google-apps/calendar/`

[9] `http://logging.apache.org/log4j/1.2/`

[10] `https://github.com/Mestway/SWIN-Project`

More concretely, method "`EventWho.getAttendeeType()`" in Google Calendar v2 returns a string that may contain either "attendee" or "organizer". Google Calender v3 replaces this method with two methods: "`boolean getSelf()`" which returns true when "attendee" should be returned and "`boolean getOrganizer()`" which returns true when "organizer" should be returned. To migrate the client, we may need to transform the code as follows, where "`getSelf()`" is a client-written method to test whether the argument is equal to "attendee",

```
String attendeeType = attendee.getAttendeeType();
boolean isSelf = isAttendee(attendeeType);
```

into the code as follows.

```
boolean isSelf = attendee.getSelf();
```

This example shows two fundamental limitations of SWIN. First, to perform the above transformation, we need to match a sequence of statements and transform them into one method calls. This requires many-to-one mapping and is not supported by SWIN. Second, we need to perform a semantic analysis on the implementation code of `isAttendee` to decide whether to transform the code into `getSelf()` or `getOrganizer()`. This kind of conditional transformation is not supported by SWIN.

Clearly, Twinning also has these limitations and cannot handle the three split methods in Google Calendar API as well. This result indicates that SWIN is as expressive as Twinning on our three case studies. Please note that many API classes have sub classes, and thus the SWIN programs should be much shorter than Twinning, as in Twinning we need to repeat the rules for the parent class also on each sub class.

6.3.3 Interesting Transformation Patterns

In the implementation of the three case studies, we also found that many transformations are not direct method replacement, but can still be expressed in SWIN by flexible use of the transformation rules. We summarize three patterns below.

Method $\leftrightarrow$ Constructor. We may need to map between class constructors and methods, and in SWIN we can directly specify such a replacement. For example, in the case from Crimson to dom4j, we write the following piece of code. This program is in the text form of SWIN, where we use `->>` to denote $\hookrightarrow$ and `->` to denote $\rightarrow$.

```
(f : DocumentBuilderFactory ->> DocumentFactory)
[ (f.newDocumentBuilder()):DocumentBuilder ->
    (new SAXReader(f)):SAXReader ]
```

Type Merging. Sometimes a set of classes in the old API become one class in the new API. In class `CalendarEvent` in Google Calendar v2, there is a method `getTitle()`. Developers can use this method to acquire the title of a source, but the type of the title is `TextConstruct`. Class `TextConstruct` is a wrapper of a string, and there is a method `getPlainText()` which returns the internal string. In Google Calendar v3, the class `CalendarEvent` becomes `Event`, which directly contains a method `getSummary()` to return the string of title. As a result, we may need to transform a sequence of method invocations "`x.getTitle().getPlainText()`" into a single invocation "`x.getSummary()`".

Although such a transformation implies a many-to-many mapping, it can be implemented in SWIN because `TextConstruct` is only used in the return type of `getTitle()` in Google Calendar API. We can consider the API upgrade as merging classes `CalendarEvent` and `TextConstruct` into `Event` and merging methods `getTitle()` and `getPlainText()` into `getSummary()`. As a result, we can remove the call to `getPlainText()` and replace `getPlainText()` with `getSummary()`. The rules are as follows.

```
(x : CalendarEvent ->> Event)
    [ (x.getTitle()):TextConstruct -> x:Event ]
(l : TextConstruct ->> Event)
    [ (l.getPlainText()):String
        -> (l.getSummary()):String ]
```

This pattern indicates that though SWIN is design for one-to-many mappings, many-to-many mappings can also be supported in a limited form from the flexibility of the rules.

Type Deletion. A class in the old API may become totally useless in the new API. In twitter4j, a `Twitter` object can be obtained by first creating a factory `TwitterFactory` and then invoking the `getInstance()` method, but in Sina Weibo API class `Weibo`, the counterpart of `Twitter`, can be directly created. In other words, the class `TwitterFactory` is deleted. Similar to the previous case, we may need to merge a sequence of method invocations "`new TwitterFactory().getInstance()`" into one single invocation "`new Weibo()`".

To implement this transformation in SWIN, we use the dummy class method [1]. We introduce a dummy class `NoF` into the client code to represent the deleted `TwitterFactory`. This dummy class has no class body and can be added to the client code before the transformation. In this way we can delete a class while maintaining the type safety. The transformation rules are as follows.

```
()[ (new TwitterFactory()):TwitterFactory
    -> (new NoF()):NoF ]
(f : TwitterFactory ->> NoF)
    [ (f.getInstance()):Twitter
     -> (new Weibo()):Weibo ]
```

7. Related work

General Transformation Frameworks. A number of general-purpose program transformation languages/frameworks have been proposed. To be independent of any programming languages, most of these languages work on the grammatical level, defining transformations on top of syntax trees. For example, TXL [11] and Stratego/XT [12] are general-purpose and grammar-oriented transformation languages, which allow the definitions of a set of rules to rewrite the abstract syntax trees of a program. Tom [13] is a language extension for Java designed to manipulate tree structures. In Tom, term rewriting and plain Java code can be mixed to write more powerful program transformations. Compared with these general-purpose transformation languages, SWIN mainly focuses on transforming Java programs in the scope of API evolution and API switching. By using Java features, SWIN allows more concise programs to be written for these tasks. Furthermore, none of the general transformation languages guarantees type-safety, for type-safety is difficult to specify in a language-independent way.

Transformation Frameworks for Java. Besides Twinning [1], several transformation languages/frameworks for Java programs are proposed. For example, Spoon [26] is a transformation framework for Java programs, providing the ability to directly read and modify program elements in Java programs. As far as we know, these transformation frameworks for Java do not consider type safety either, and there is no guarantee that the transformation does not introduce compilation errors. Refaster [27] uses compilable before-and-after examples of Java code to specify a Java refactoring. Similarly to our work, this work also mainly focuses on solving the method replacement which is useful in real API migration. Moreover, using direct Java examples to describe the transformation is convenient. However, Refaster cannot assure the well-typedness of the whole program during transformation, as it only requires that each transformed expression is well-typed.

Type-Safe Transformations. Approaches for ensuring type safety also exist. Hula [15] is a rule-based update (or transformation) language for Haskell, ensuring updates are performed in a type-safe manner. The type-safe transformation depends on a core calculus–update calculus [16], which provides type-safe transformation over lambda calculus. This work distinguishes program changes into declaration changes, definition changes, and application changes, and requires the three changes to be consistent. Compared with our work, update calculus allows the dynamic change of type definitions during transformation while our approach focuses on static type mappings as the difference between the old API and the new API are already known during program adaptation for different APIs. On the other hand, update calculus allows only the replacement of a type to a more generic type, while our approach supports more type mapping between independent types, such as Vector to ArrayList, because these type changes are dominant in program adaptation between APIs.

The work of Balaban [14] et al. focuses on a particular problem in the adaptation of program between APIs: when some part of the program cannot be changed, how to change the other parts while preserving well-typedness and other properties. This work extracts the type constraints from the Java program, then solves the constraints using a constraint solver to prevent type incorrect program transformations. Different from this work that considers the well-typedness of a particular client program, our work focuses on the type-safety of the transformation itself, taking into account all possible client programs. The work of Spoon [26] focuses on the well-typedness of a program using API with forthcoming or deprecated methods. This work extends FJ with forthcoming and deprecated methods, and proves the soundness of extended FJ. However, this work only allows update on methods, rather than update on classes.

Semantic-Preserving Transformations. Refactoring-based approaches [3, 4] treat the API changes as a set of refactorings. The API developers records their changes on the API as a set of refactorings, and the later these refactorings can be replayed on the client programs to transform the client programs to the new API. In this way, the adaptation of the client programs is not only type-safe but also semantic-preserving. However, this approach has limitations. First, this approach cannot support API changes that cannot be expressed as refactorings. Second, this approach only applies to API update, and cannot support migrating programs between alternative APIs, which are independently developed. The work of Leather [17] et al. provides an approach to preserve semantics of a program while changing terms involving type A to terms involving type B using type-changing rewrite rules. This work mainly focuses on conversion between isomorphic types, whereas our work focuses on transformation between any two types. Moreover, unlike this work performing transformations on lambda calculus with let-polymorphism, our work performs transformations on Featherweight Java which need to solve problems introduced by object orientation, such as subtyping.

Package templates [18] is an extension to Java to write reusable and adaptable modules. Since the template instantiation process in package templates includes operations like renaming and class merging, it can be considered as a semantics-preserving program transformation process. Different from our work, the program transformation in package templates mainly focuses on the changes on the class level, and does not consider the replacement of method invocations. A key point in package templates is to avoid name collision in transformation. Our approach does not consider this issue because in Java language, the client code and the API are usually in different packages, and the names are almost impossible to collide.

Heuristic-based Transformations. Several approaches try to further reduce the cost of program adaptation between APIs by automatically discovering the transformation program using heuristic rules. The heuristic rules range from comparing API source code [5], analyzing existing client source code [6, 8, 9], and discovering similar code pieces [10]. Since these approaches are heuristic-based, there is no guarantee the discovered transformations are type-safe.

8. Conclusion and Future Work

In this paper, we have proposed a type-safe transformation language SWIN for program adaptation in the scope of API switching and API updating. Different from the existing language Twinning, SWIN provides a full type-safe guarantee, more flexible rule matching, and formal semantics of its core part. The type safety of core SWIN is proved about formally in Featherweight Java, and the case studies show that SWIN is expressive enough to handle many useful transformations in practice and is as expressive as Twinning on the cases.

In future, the inability of SWIN to handle the three method splitting changes as discussed in Section 6.3 needs to be addressed. This could possibly be handled by adding the dataflow information into SWIN to handle many-to-many mapping, adding semantic conditions to allow semantic checking, and loosing the restriction on the type mapping to allow one-to-many type mapping.

Acknowledgments

We are grateful for the fruitful discussions with Prof. Martin Erwig at Oregon State University on update calculus [16] and Prof. James R. Cordy at Queen's University on TXL [11].

This work is supported by the National Basic Research Program of China (973) under Grant No. 2011CB302604, the High-Tech Research, the Development Program of China under Grant No.2013AA01A605, the National Natural Science Foundation of China under Grant No.61202071, 61121063, U1201252, and the JSPS Grant-in-Aid for Scientific Research (A) Grant Number 25240009.

References

[1] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs, in: *Proc. ICSE*, 2010.

[2] T. Bartolomei, K. Czarnecki, and R. Lämmel. Swing to SWT and Back: patterns for API migration by wrapping, in: *Proc. ICSM*, 2010.

[3] D. Dig, S Negara, and R. Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries, in: *Proc. ICSE*, 2008.

[4] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution, in: *Proc. ICSE*, 2005.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components, in: *Proc. ECOOP*, 2006.

[6] H. Nguyen, T. Nguyen, G. Jr, A. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to API usage adaptation, in: *Proc. OOPSLA*, 2010.

[7] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ, *ACM Trans. Program. Lang. Syst*, 2001.

[8] Y. Padioleau, J. Lawall, R. R Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers, in: *Eurosys*, 2008.

[9] J. Andersen and J. L. Lawall. Generic patch inference, in: *Proc. ASE*, 2008.

[10] N. Meng, M. Kim, and K. S. Mckinley. Systematic editing: generating program transformations from an example, in: *Proc. PLDI*, 2011.

[11] J. R. Cordy. The TXL source transformation language, *Science of Computer Programming*, 2006.

[12] E. Visser. Program transformation in stratego/xt: rules, strategies, tools and systems in stratego xt/0.9, *Domain Specific Program Generation*, 2004.

[13] E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. Tom: piggybacking rewriting on Java, in: *Proc. RTA*, 2007.

[14] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration, in: *Proc. OOPSLA*, 2005.

[15] M. Erwig and D. Ren. A rule-based language for programming software updates, *SIGPLAN Notices.*, 2002.

[16] M. Erwig and D. Ren. An update calculus for expressing type-safe program update, *Science of Computer Programming*, 2007.

[17] S. Leather, J. Jeuring, A. Löh, and B. Schuur. Type-changing rewriting and semantics-preserving transformation, in: *Proc. PEPM*, 2014.

[18] E. W. Axelsen and S. Krogdahl. Package templates: a definition by semantic-preserving source-to-source transformations to efficient Java code, in: *Proc. GPCE*, 2012.

[19] J. Li, Y. Xiong, X. Liu, and L. Zhang. How does web service API evolution affect clients?, in: *Proc. ICWS*, 2013.

[20] E. Visser. A survey of strategies in rule-based program transformation systems, *Journal of Symbolic Computation*, 2005.

[21] M. Pilgrim. Dive into Python 3, *2nd edition, APress*, 2009.

[22] B. E. Cossette and R. J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries, in: *Proc. FSE*, 2012.

[23] C. Wang, J. Li, Y. Xiong, and Z. Hu. Formal Definition of SWIN language, *Technical Note, available at https://github.com/Mestway/SWIN-Project/blob/master/docs/pepm-15/TR/TR.pdf*, 2014.

[24] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: an extensible compiler framework for Java, in: *Proc. CC*, 2003.

[25] R. Pawlak, C. Noguera, and N. Petitprez. Spoon: program analysis and transformation in Java, *Technical Report 5901*, INRIA, 2006.

[26] S. A. Spoon. Fined-grained API evolution for method deprecation and anti-deprecation, in: *Proc. FOOL*, 2006.

[27] L. Wasserman. Scalable, example-based refactorings with refaster, in: *Proc. WRT*, 2013.

[28] B. Pierce. Types and Programming Languages, *MIT Press*, 2002.

Appendix

A. Feather Weight Java

A.1 Syntax

This part presents the syntax for FJ.

$$\begin{array}{rcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \\
\mathtt{K} & ::= & \mathtt{C}(\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}};\} \\
\mathtt{M} & ::= & \mathtt{C\ m}(\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ t};\} \\
\mathtt{t} & ::= & \mathtt{x} \mid \mathtt{t.f} \mid \mathtt{t.m}(\bar{\mathtt{t}}) \mid \mathtt{new\ C}(\bar{\mathtt{t}}) \mid (\mathtt{C})\ \mathtt{t} \\
\mathtt{v} & ::= & \mathtt{new\ C}(\bar{\mathtt{v}})
\end{array}$$

A.2 Subtyping

This part presents the derivation of subtype relation in FJ.

$$\frac{}{\mathtt{C} <: \mathtt{C}}\ \text{(S-SELF)}$$

$$\frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}\ \text{(S-TRANS)}$$

$$\frac{\mathtt{CL}\ =\ \mathtt{class\ C\ extends\ D}\ \{...\}}{\mathtt{C} <: \mathtt{D}}\ \text{(S-DEF)}$$

A.3 Typing Rules

In this section we present the typing rules for FJ term and FJ class declaration obtained from [28].

Note that CAST rule in FJ type system is divided into three rules. FJ-UCAST and FJ-DCAST are for cast between two classes with subtype relation while FJ-SCAST is the typing rule for cast between two irrelevant classes, which will generate a "stupid warning" in the typing progress.

$$\frac{\mathtt{x} : \mathtt{C} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{C}}\ \text{(FJ-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \qquad \mathtt{fields}(\mathtt{C}_0) = \bar{\mathtt{C}}\ \bar{\mathtt{f}}}{\Gamma \vdash \mathtt{t}_0.\mathtt{f}_\mathtt{i} : \mathtt{C}_\mathtt{i}}\ \text{(FJ-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{C}_0 \quad \mathtt{mtype}(\mathtt{m}, \mathtt{C}_0) = \bar{\mathtt{D}} \to \mathtt{C} \\ \Gamma \vdash \bar{\mathtt{t}} : \bar{\mathtt{C}} \qquad \bar{\mathtt{C}} <: \bar{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\bar{\mathtt{t}}) : \mathtt{C}}\ \text{(FJ-INVK)}$$

$$\frac{\mathtt{fields}(\mathtt{C}_0) = \bar{\mathtt{D}}\ \bar{\mathtt{f}} \qquad \Gamma \vdash \bar{\mathtt{t}} : \bar{\mathtt{C}} \qquad \bar{\mathtt{C}} <: \bar{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}_0(\bar{\mathtt{t}}) : \mathtt{C}}\ \text{(FJ-NEW)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{D} <: \mathtt{C}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}}\ \text{(FJ-UCAST)}$$

$$\frac{\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} <: \mathtt{D} \qquad \mathtt{C} \neq \mathtt{D}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}}\ \text{(FJ-DCAST)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_0 : \mathtt{D} \qquad \mathtt{C} \not<: \mathtt{D} \qquad \mathtt{D} \not<: \mathtt{C} \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash (\mathtt{C})\mathtt{t}_0 : \mathtt{C}}\ \text{(FJ-SCAST)}$$

$$\frac{\begin{array}{c}\bar{\mathtt{x}} : \bar{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{t}_0 : \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0 \\ \mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{...\} \\ \mathtt{override}(\mathtt{m}, \mathtt{D}, \bar{\mathtt{C}} \to \mathtt{C}_0)\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\ \{\mathtt{return\ t}_0;\}\ \mathtt{OK\ in\ C}}\ \text{(FJ-M-OK)}$$

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}\ (\bar{\mathtt{C}}\ \bar{\mathtt{f}})\{\mathtt{super}(\bar{\mathtt{f}});\ \mathtt{this}.\bar{\mathtt{f}} = \bar{\mathtt{f}}\} \\ \mathtt{fields}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}} \qquad \bar{\mathtt{M}}\ \mathtt{OK\ in\ C}\end{array}}{\mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}}; \mathtt{K}\ \bar{\mathtt{M}}\}\ \mathtt{OK}}\ \text{(FJ-C-OK)}$$

A.4 Auxiliary Definition

This part presents the auxiliary functions used in FJ typing rules.

$$\frac{}{\mathtt{fields}(\mathtt{Object}) = \{\}}\ \text{(FIELD-OBJECT)}$$

$$\frac{\begin{array}{c}\mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{fields}(\mathtt{D}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}}\end{array}}{\mathtt{fields}(\mathtt{C}) = \bar{\mathtt{D}}\ \bar{\mathtt{g}},\ \bar{\mathtt{C}}\ \bar{\mathtt{f}}}\ \text{(FIELD-LOOKUP)}$$

$$\frac{\begin{array}{c}\mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{B\ m}\ (\bar{\mathtt{B}}\ \bar{\mathtt{x}})\ \{\mathtt{return\ t};\} \in \bar{\mathtt{M}}\end{array}}{\mathtt{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{B}} \to \mathtt{B}}\ \text{(METHOD-LOOKUP1)}$$

$$\frac{\begin{array}{c}\mathtt{CT}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{m}\ \text{is not defined in}\ \bar{\mathtt{M}}\end{array}}{\mathtt{mtype}(\mathtt{m}, \mathtt{C}) = \mathtt{mtype}(\mathtt{m}, \mathtt{D})}\ \text{(METHOD-LOOKUP2)}$$

$$\frac{\mathtt{mtype}(\mathtt{m}, \mathtt{D}) = \bar{\mathtt{D}} \to \mathtt{D}_0\ \mathtt{implies}\ \bar{\mathtt{C}} = \bar{\mathtt{D}}\ \mathtt{and}\ \mathtt{C}_0 = \mathtt{D}_0}{\mathtt{override}(\mathtt{m}, \mathtt{D}, \bar{\mathtt{C}} \to \mathtt{C}_0)}\ \text{(OVERRIDE)}$$