A Variation on Java Wildcards – Trading Expressiveness for Global Type Inference

Andreas Stadelmeier
DHBW Stuttgart, Campus Horb, Germany

Martin Plümicke
DHBW Stuttgart, Campus Horb, Germany

Peter Thiemann
Institut für Informatik, Universität Freiburg, Germany

Abstract

In standard Java, wildcards behave like existential types: they must be *opened* before use in a method invocation, a process the compiler performs implicitly via *capture conversion*. We present Java-TX, a dialect of Java that sidesteps this existential encoding and treats wildcards as placeholders for unknown types, used directly without prior opening. This choice simplifies the type system and makes it compatible with an existing global type inference algorithm. The trade-off is that some method calls valid in standard Java become unavailable in Java-TX. In effect, we trade some of Java's wildcard expressiveness for global type inference.

We explore the metatheory of Java-TX through Featherweight Java-TX (FJ-TX), a functional core calculus for Java-TX that extends Featherweight Generic Java with our wildcard interpretation. We prove type soundness for FJ-TX. Finally, we evaluate the practical impact of omitting capture conversion by conducting a study on open-source Java projects by calculating an underapproximation of how much existing Java code is compatible with the Java-TX type system.

2012 ACM Subject Classification Software and its engineering → Syntax

Keywords and phrases type inference, Java, subtyping, wildcards, capture conversion

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2026.26

1 Introduction

Global type inference is a natural goal for Java because explicit type annotations impose a real maintenance burden. Although Java has steadily moved toward inference[1], method and field types must still be annotated by hand, making certain refactorings unnecessarily painful (see Section 2.4 for concrete examples and evidence). A global type inference (GTI) algorithm that infers all type annotations would eliminate most of this overhead.

But global type inference for Java-style generics faces a fundamental obstacle with wildcard types. In Java, wildcard types such as `Box<?>` are treated as bounded existential types that must be *opened* before they can be used in a method call. This opening (capture conversion) replaces each wildcard with a fresh type variable and is performed eagerly at every method invocation. Attempts to extend constraint-based global type inference to support capture conversion have not succeeded: during constraint generation the types of subexpressions are not yet known, making it unclear whether and how capture conversion should be applied at a given call site (see Section 2.3).

Plümicke's global type inference algorithm [13] resolves this impasse by adopting a different wildcard semantics: wildcards are treated as ordinary types that can serve directly as type arguments in method invocations, without any prior opening. The resulting language,

[1] the diamond operator, the `var` keyword, and lambda type inference all reduce boilerplate

Java-TX (Java Type eXtended), is a dialect of Java with global type inference that treats wildcards as ordinary types rather than existentials, foregoing capture conversion entirely. This design is not proposed as a superior alternative to Java's wildcard treatment, but is rather a consequence of enabling global type inference: the wildcard semantics of Java-TX naturally arise from the limits of this global type inference algorithm.

The key technical difference concerns reflexivity. Wildcards denote unknown types, so subtyping is not reflexive on them. If wildcards are to be used directly as type arguments, the corresponding method type parameters must therefore also be treated as non-reflexive. Consequently, Java-TX does not assume reflexivity on method type parameters by default, and wildcards can be substituted for such parameters without capture conversion. However, if the method body requires the judgment `X <: X`, the parameter must be declared with an explicit reflexivity constraint `X extends X`, which wildcards cannot satisfy, ruling them out as type arguments at that position. Java-TX and Java are incomparable: each accepts some programs the other rejects (see Section 2 for a detailed comparison).

Up to now, Java-TX was defined only by its inference algorithm, with no semantic foundation for the language. In a technical report, Plümicke [15] made a first attempt at defining such a calculus, but without a soundness proof. The present paper completes this effort by establishing type soundness for FJ-TX, and assesses the practical impact of the different treatment of wildcards on real-world Java code.

1.1 Contributions of this work

- In depth comparison of the treatment of wildcards in Java and Java-TX.
- Definition of the core calculus Featherweight Java-TX (FJ-TX).
- Proof of type soundness for FJ-TX using the standard preservation/progress structure.
- An evaluation of the practical impact of the different treatment of wildcards.

1.2 Overview

Section 2 introduces the different approaches to wildcards taken by Java and Java-TX along with a detailed comparison based on practical examples. We conclude this section with a brief summary of further features of Java-TX in Section 2.4.

Section 3 defines the calculus Featherweight Java-TX (FJ-TX) by extending and adapting syntax, typing, and operational semantics of Featherweight Generic Java [7].

Section 4 contains the type soundness proof for FJ-TX.

Section 5 evaluates the differences between Java and Java-TX qualitatively on real world examples: the source code of the JDK and a selection of seven popular Java projects.

We close with related work (Section 6) and a brief outlook (Section 7).

2 Java vs. Java-TX

The sound interaction of subtyping and generics in an imperative programming language can be very simple. For instance, the initial proposals for generic Java, as well as the first Java version implementing generics, required generic types to be *invariant*. To illustrate the concept, we consider the class `Box` in Fig. 1a. Invariance means that `Box<A>` is a subtype of `Box<B>` if and only if `A` = `B`. In Fig. 1b the assignment `obox = ibox` violates invariance. Java rejects it to prevent a value of type `Object` to be stored in a box of `Integer`.

Subsequently, the design of Java's collection classes showed that invariance is overly restrictive. If a method only reads from a generic type, then covariant subtyping is sound. If a method only writes to a generic type, then contravariant subtyping is sound. Only if a

```
class Box<T> {
  T elem;
  T get() { return elem; }
  void set(T x) { elem = x; }
}
```

(a) Definition.

```
Box<Integer> ibox = new Box<>(42);
Box<Object> obox = ibox; // illegal
// because set() would put an Object
// into a Box of Integers:
obox.set(new Object());
```

(b) Erroneous Java program.

Figure 1 The class `Box`.

```
<T>void fg (Box<T> b1, Box<T> b2) {
  b1.set(b2.get());
}
Box<?> b1 = new Box<Integer>(42);
Box<?> b2 = new Box<String>("foo");
fg (❶b1, ❷b2);   // Java type error here
```

Listing 1 Rejected Java program involving capture conversion

method both reads and writes from a generic type, invariant subtyping must be used. In consequence, Java introduced *wildcard types* to enable use-site variance for generic types. Use-site variance in programming allows us to specify variance (how subtypes relate) for a generic type where it is used, not where it is declared. Thus, the variance of a generic type can be different at each use (e.g., in Java and Kotlin). In contrast, with definition-site variance the definition of a generic type specifies its variance once and for all (e.g., in Eiffel, C#, OCaml). See Altidor et al. [1] for a in-depth treatment of both concepts.

Since version 5, Java [6] and the existing core calculi for wildcards [3, 4, 18, 19] employ bounded existential types and capture conversion to handle wildcard types.

2.1 Capture Conversion - The Java Approach

Wildcards enable variant subtyping in Java as in `Box<String><: Box<? extends Object>`, where `<:` stands for the subtyping relation (cf. Figure 6) and the wildcard `?` "forgets" the `String` type and only remembers that it is a subtype of `Object`. However, wildcards cannot be treated like regular types. In Listing 1 the method `fg` cannot be called with two arguments of type `Box<?>`. At first glance it seems like the method parameter `T` could be replaced with a wildcard resulting in a method type (`Box<?>`, `Box<?>`) → `void`. But Java rejects this idea as it would lead to unsoundness: In this example the contents of `Box<String>` would end up in `Box<Integer>`.

What Java actually does for every method call involving wildcards is to replace every wildcard type with a fresh capture variable. This process is called capture conversion [3, 4, 18] and is hidden from the programmer. In Listing 1, the first argument ❶ has the type `Box<?>`, which is transformed to `Box<CAP#1>`. Argument ❷ gets the type `Box<CAP#2>`. Now the method call is subject to a regular type check treating `CAP#1` and `CAP#2` as normal types. The method call is rejected, because there is no substitution for `T` in the method type `<T>(Box<T>, Box<T>)` → `void` to fit the argument types (`Box<CAP#1>`, `Box<CAP#2>`).

Capture conversion does not require any participation of the programmer and enables many method calls involving wildcards.

```
<X> void assign(Box<X> b1, Box<X> b2) {
  b1 = b2;
}

Box<?> b1 = ...
Box<?> b2 = ...

❶assign(b1, b2);
```

Listing 2 Non-reflexive method type parameters

2.2 Method-Site Variance - The Java-TX Approach

Java-TX is a dialect of Java with global type inference [12], which includes support for wildcards. This system has been developed and used for a number of years (see Section 2.4). It does not rely on an encoding of wildcards as existentials, but rather supports a notion of wildcard types (different from Java) directly. It basically supports declaration-site variance for method parameters combined with wildcards treated like regular types. In particular, there is no capture conversion and one can substitute wildcards for certain Java-TX type parameters. Whether a type parameter is eligible for instantiation by a wildcard depends on how the corresponding argument is used inside the method body.

The key to this approach is to assume that subtyping is *not* reflexive on type variables in general. If the reflexivity is required on a type variable `X`, it must be declared explicitly by the constraint "`X <: X`". This constraint forbids the instantiation of `X` with a wildcard.

Note: The Java-TX approach works best in combination with global type inference. The constraints generated during inference indicate whether a method type parameter requires the reflexivity constraint. For readability, all the Java-TX examples shown in this chapter are fully annotated with their inferred types.

Listing 2 shows an example where the method type parameter `X` can be instantiated with a wildcard type, because `X` does not have to be reflexive inside the `assign` method. Therefore the call at ❶ is valid. It is possible to instantiate `X` with a wildcard resulting in the method type instance (`Box<?>`, `Box<?>`) → `void`.

On the other hand, Java rejects the call to `assign`, because Java applies capture conversion separately to the method parameters `b1` and `b2`. As their capture types are different, there is no instantiation for `X` to accept the call.

The Java-TX approach comes with some restrictions, too. The next example is a legal Java program fragment which is accepted via capture conversion, but rejected by Java-TX.

```
<T> void m(Box<T> b) {
  b.set(b.get());
}
Box<?> bb = ...;
m(bb);
```

Java uses capture conversion in the call to `m`. The wildcard in `Box<?>` is replaced by a fresh capture type `CAP#1`. In contrast, Java-TX does not accept this method definition. The return type of `get()` is `T` and `set()` expects an argument of type `T`. For `b.set(b.get())` to be accepted the type variable `T` must be reflexive (`T <: T`). Thus, the method `m` requires a reflexive constraint `T extends T` as shown in Listing 3. In Java-TX this method cannot be called with a type `Box<?>` like in line ❶ because subtyping is not reflexive on wildcards.

```
// Java-TX
<T extends T> void repack(Box<T> b) {
  b.set(b.get());
}

Box<?> b = ...;

❶repack(b); // Illegal in Java-TX
```

Listing 3 Reflexive type parameter *Note: Java-TX source code highlighted in yellow*

The next example we discuss is a variation of a method shown in Listing 1.

```
<W, R extends W>
void overwrite(Box<W> x, Box<R> y){
  x.set(y.get());
}
```

This program is accepted by both Java and by Java-TX and type inference infers the same typing. Let's try to use it with wildcard types.

```
Box<? extends Object> b1 = new Box<Integer>(42);
Box<? extends Object> b2 = new Box<String>("foo");
overwrite(b1, b2);
```

Java rejects this program. In the call to `overwrite`, the wildcards in the types of `b1` and `b2` get instantiated to different capture variables, say `CAP#1` and `CAP#2`, which are not in a subtype relation. Hence, the constraint `R <: W` is not satisfied and the method call is rejected.

Type inference in Java-TX obtains the instantiation

[`? extends Object/W`, `? extends Object/R`]

for the method call and has to check

`? extends Object <: ? extends Object`

which leads to a type error as subtyping is not reflexive on wildcard types.

So the method definition is accepted in both languages and the call is rejected in both languages.

We conclude with a somewhat contrived example that is accepted by both, Java and Java-TX, but cannot be expressed in the core calculi based on existential types (e.g., [3]).

```
<X,Y> void equalize(Pair<X,Y> a, Pair<X,Y> b) {}
<Y> Pair<?, Y> left()  { return null; }
<X> Pair<X, ?> right() { return null; }

equalize(left(), right());
```

The difficulty in typing the method call to `equalize` with capture conversion is that the capture variable introduced for `left` instantiates the generic variable of `right` and vice versa. This mutual instantiation pattern cannot be expressed in (e.g.) Bierhoff's calculus [3], but recent Java compilers support it.

The example presents no difficulty for Java-TX: both `X` and `Y` can be instantiated to a wildcard because `equalize()` does not require a reflexivity constraint.

$\exists$X.Box<X> <: Box<a>

$b2$ <: Box<a>

Figure 2 Java's subtype constraints.

Box<?> <: Box<a>

$b2$ <: Box<a>

Figure 3 Java-TX subtype constraints.

2.3 Capture Conversion and Global Type Inference

Capture conversion turns out to be incompatible with global type inference, where the goal is to reconstruct missing type annotations in partially untyped Java programs. Recent work [17] has shown that type inference works well for Java with generics, but without wildcards. When trying to extend type inference to Java-style wildcards, we find that capture conversion is not compatible with constraint-based global type inference, as it is based on first generating type constraints and solving them afterwards. However, during constraint generation it is impossible to determine if a capture conversion is required because the argument types to a method call are not yet known.

In the following example `untypedMethod`, the types of `b1` and `b2` only emerge during the constraint solving process. (See Listing 2 for the definition of `assign`.)

```
untypedMethod(b1, b2){
  return assign(b1, b2);
}
```

Constraints $\Longrightarrow$ $b1$ <: Box<a>, $b2$ <: Box<a>

The variables $b1$ and $b2$ on the right side are *type placeholders* for the types of `b1` and `b2`, respectively. A constraint solver has to find a substitution for those type placeholders that satisfies all constraints. A satisfying substitution can afterwards be translated into a well-typed Java program by adding the type calculated for $b1$ ($b2$) to the variable definition of `b1` (`b2`) in the input program.

In Java, `Box<?>` is treated as an existential type $\exists$`X.Box<X>` which is implicitly opened for this method call. To answer the question if the variable `b1` can have the type `Box<?>` in Java, we would have to find a solution for the constraints in Figure 2. In this case, a solution is a substitution for the type placeholders $b1$ and $b2$ in compliance with the given constraints. But the constraints shown in Figure 2 have no solution.

On the other hand, Java-TX can handle the example easily. In the resulting constraint set in Figure 3, Java-TX treats the wildcard ? like any other type. The substitution $a \rightarrow$? and $b1 \rightarrow$ `Box<?>` and $b2 \rightarrow$ `Box<?>` leads to `Box<?> <: Box<?>`, `Box<?> <: Box<?>` and is a valid solution.

2.4 Java-TX Use Cases

A static type system helps programmers in many ways. Coupled with an IDE [10] it can identify errors on the spot, show additional information, and even propose code completions. But at times it can generate considerable overhead, if type annotations are required in (too) many places as in Java. If one annotation is changed, for example a `List<`*String*`>` to a `Set<`*String*`>`, this change can ensue a cascade of type changes throughout the program. A study that checked the git logs of 129 Java projects [9] found that type changes are even more frequent than renamings. In a survey presented by Negara et al. [11] 86.4% of the 420 participants demanded automated IDE support for type changes in Java.

Consider the *Change Field Type* example in Figure 4. After changing `int` $\rightarrow$ `long` of the field `mileage` the programmer has to manually change the types at ❶ and ❷. A global type inference (GTI) algorithm would render manual type changes obsolete. With full GTI,

```
class TypeChangeExample {
    int mileage;

    int getMileage(){
        return mileage;
    }

    void update(int m){
        mileage += m;
    }
}
```

(a) Before.

$\Longrightarrow$

```
class TypeChangeExample {
    long mileage;

    ❶long getMileage(){
        return mileage;
    }

    void update(❷long m){
        mileage += m;
    }
}
```

(b) After `long` → `int`.

Figure 4 *Change Field Type.* Example from Negara et al [11].

no type annotations are needed and it is sufficient to change the field types in the class definition.

Java-TX provides a global type inference algorithm and adds some further convenience features. For example, Java-TX comes with a predefined bundle of function types of the form `Fun`N`$$<`$T_1, \ldots, T_N, R$`>` (where N is the number of arguments) with contravariant argument types and covariant return types [16]. Function types are assigned to lambda expressions.

In the following we give some examples of programming with Java-TX.

▶ **Example 1.** Consider a lambda-expression that takes three arguments (two values and a function) where the function is applied to the two arguments.

```
x -> y -> f -> f.apply(x,y);
```

In Java, using the package `java.util.function` the principal type would be

```
Function<? super A,
         ? extends Function<? super B,
                            ? extends Function<
                                    ? super BiFunction<? super A,
                                                       ? super B,
                                                       ? extends C>>,
                                    ? extends C>>>
```

As `Function` is an ordinary Java interface, covariance and contravariance have to be expressed by wildcards. The equivalent Java-TX type is much simpler:

```
Fun1$$<A, Fun1$$<B, Fun2$$<A, B, C>, C>>
```

▶ **Example 2.** The example in Listing 4 illustrates the extended overloading mechanism of Java-TX. In the class OL, the method name m is overloaded by two different method declarations. If the types `Integer`, `Double`, `String`, and `Boolean` are visible the type inference algorithm infers intersection types

```
m    :  Integer → Integer          main  :  Integer → Integer
     ∧  Double → Double                   ∧  Double → Double
     ∧  String → String                   ∧  String → String
                                          ∧  Boolean → Boolean
m    :  Boolean → Boolean
```

```
class OL {
    m(x) { return x + x; }
    m(x) { return x || x; }
}
class OLMain {
  main(x) {
      var ol = new OL();
      return ol.m(x);
  }
}
```

Listing 4 Overloading

```
class Matrix extends Vector<Vector<Integer>> {
    mul(m) {
        var ret = new Matrix();
        for(v1 : this) {
            for(j ...) {
                var erg = 0;
                for(k ...) {
                    erg = erg + v1.get(k) * m.get(k).get(j);
                } ...
            } ...
         }
        return ret;}
}
```

Listing 5 Matrix multiplication

which are resolved in bytecode by overloaded methods. While function types are covered by our soundness proof, intersection types are not.

To obtain this result from global type inference, the programmer has to include the following import declarations

```
import java.lang.Integer
import java.lang.Double
import java.lang.String
import java.lang.Boolean
```

A less restrictive import declaration like `java.lang.*` would extend the visibility to `Float`, `Long`, and so on. This declaration would lead to additional conjuncts in the intersection type, but their presence severely affects the performance of type inference. The use of visibility in Java-TX is comparable to the way that Haskell manages type classes: type class instances are only considered by the type checker if the modules containing them are imported.

▶ **Example 3.** Listing 5 shows the matrix multiplication in Java-TX.

The class `Matrix` is implemented as an extension of `Vector<Vector<Integer>>`. The method `mul` implements the multiplication of two matrices `this` and `m`. An obvious typing of `mul` would be `Matrix mul(Matrix m)`. However, this typing is not the only possible typing. In fact, it is easy to see that there are further typings. Here are some alternatives:

$$
\begin{aligned}
\texttt{N} &::= \texttt{C<}\overline{\texttt{E}}\texttt{>} \\
\texttt{T}, \texttt{U}, \texttt{L} &::= \texttt{N} \mid \texttt{X} \\
\texttt{E}, \texttt{F}, \texttt{G} &::= \texttt{T} \mid \texttt{W} \\
\texttt{W} &::= ?^{\texttt{U}}_{\texttt{L}} \\
\texttt{P} &::= \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{>} \lhd \texttt{ N } \{\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{;}\ \overline{\texttt{M}}\} \\
\texttt{M} &::= \texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{T}}\texttt{> T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{return e;\}} \\
\texttt{e} &::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new N(}\overline{\texttt{e}}\texttt{)} \mid \texttt{(T)e} \mid \texttt{null}
\end{aligned}
$$

Figure 5 Syntax of FJ-TX.

```
Matrix mul(Vector<Vector<Integer>> m)
```

```
Vector<Vector<Integer>> mul(Vector<Vector<Integer>> m)
```

```
Matrix mul(Vector<? extends Vector<? extends Integer>> m)
```

Java-TX infers a maximal type for the method wrt. the subtyping ordering of function types where argument types are contravariant and return types are covariant. For this example, type inference determines the maximal type as

```
Matrix mul(Vector<? extends Vector<? extends Integer>> m)
```

Global type inference for Java-TX does not come cheap. The underlying type unification algorithm is NP-hard [17], but there are some heuristics that improve the running time significantly [14].

3 Featherweight Java-TX

In this section, we define FJ-TX, a core calculus for Java-TX in the style of Featherweight Java [8].

3.1 Syntax

Fig. 5 defines the syntax of FJ-TX. It extends Featherweight Generic Java (FGJ) [8] with wildcards. We let `C` range over class names including the predefined names `Object` and $\bot$ (the empty type), which do not take parameters. As wildcards are not allowed in all positions of the syntax, we distinguish several syntactic categories of types:

`X` type variables.
`N` ranges over class types.
`T, U, L` range over all FJ-TX types, except top-level wildcards.
`E, F, G` range over all types including wildcards.
`W` ranges over wildcard types. The notation $?^{\texttt{U}}_{\texttt{L}}$ denotes a wildcard with lower bound `L` and upper bound `U`. For example, we write $?^{\texttt{Integer}}_{\bot}$ for `? extends Integer`, where $\bot$ stands for the lower bound of all types. Analogously, $?^{\texttt{Object}}_{\texttt{Number}}$ stands for `? super Number`.

Well-formed types:

$$\text{WF-BOTTOM}\quad \Delta \vdash \bot\ \texttt{OK} \qquad \text{WF-OBJECT}\quad \Delta \vdash \texttt{Object OK} \qquad \text{WF-VAR}\quad \frac{\texttt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \texttt{X OK}}$$

$$\text{WF-CLASS}\quad \frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}, \overline{\texttt{Y}} \triangleleft \overline{\texttt{Y}}\texttt{>} \triangleleft \texttt{N}\{\ldots\} \quad \Delta = \overline{\texttt{X}} \texttt{<:} \overline{\texttt{N}}, \overline{\texttt{Y}} \texttt{<:} \overline{\texttt{Y}} \quad \Delta \vdash \overline{\texttt{E}}\ \texttt{OK} \quad \Delta \vdash [\overline{\texttt{E}}/\overline{\texttt{X}}]\overline{\texttt{X}} \texttt{<:} [\overline{\texttt{E}}/\overline{\texttt{X}}]\overline{\texttt{N}} \quad \{\overline{\texttt{Y}}\} \subseteq \{\overline{\texttt{X}}\} \quad \Delta \vdash [\overline{\texttt{E}}/\overline{\texttt{X}}]\overline{\texttt{Y}} \texttt{<:} [\overline{\texttt{E}}/\overline{\texttt{X}}]\overline{\texttt{Y}}}{\Delta \vdash \texttt{C<}\overline{\texttt{E}}\texttt{>}\ \texttt{OK}}$$

$$\text{WF-WILDCARD}\quad \frac{\Delta \vdash \texttt{T} \texttt{<:} \texttt{T}'}{\Delta \vdash ?^{\texttt{T}'}_{\texttt{T}}\ \texttt{OK}}$$

Subtyping:

$$\text{S-REFL}\quad \Delta \vdash \texttt{N} \texttt{<:} \texttt{N} \qquad \text{S-TRANS}\quad \frac{\Delta \vdash \texttt{E} \texttt{<:} \texttt{E}' \quad \Delta \vdash \texttt{E}' \texttt{<:} \texttt{E}''}{\Delta \vdash \texttt{E} \texttt{<:} \texttt{E}''} \qquad \text{S-VAR}\quad \frac{(\texttt{X} \texttt{<:} \texttt{T}) \in \Delta}{\Delta \vdash \texttt{X} \texttt{<:} \texttt{T}} \qquad \text{S-BOT}\quad \bot \texttt{<:} \texttt{T}$$

$$\text{S-CLASS}\quad \frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{T}}\texttt{>} \triangleleft \texttt{N}\ \{\ldots\} \quad \Delta \vdash \texttt{C<}\overline{\texttt{E}}\texttt{>}\ \texttt{OK}}{\Delta \vdash \texttt{C<}\overline{\texttt{E}}\texttt{>} \texttt{<:} [\overline{\texttt{E}}/\overline{\texttt{X}}]\texttt{N}} \qquad \text{S-WC-CLASS}\quad \frac{\Delta \vdash \overline{\texttt{E}} \lessdot_? \overline{\texttt{E}'}}{\Delta \vdash \texttt{C<}\overline{\texttt{E}}\texttt{>} \texttt{<:} \texttt{C<}\overline{\texttt{E}'}\texttt{>}} \qquad \text{S-WC-SUPER}\quad \frac{\Delta \vdash \texttt{L} \texttt{<:} \texttt{U}}{\Delta \vdash \texttt{L} \texttt{ <: } ?^{\texttt{U}}_{\texttt{L}}}$$

$$\text{S-WC-EXTENDS}\quad \frac{\Delta \vdash \texttt{L} \texttt{<:} \texttt{U}}{\Delta \vdash ?^{\texttt{U}}_{\texttt{L}} \texttt{<:} \texttt{U}}$$

Use-site variance subtyping:

$$\text{USV-EXTENDS}\quad \frac{\Delta \vdash \texttt{L} \texttt{<:} \texttt{E} \quad \Delta \vdash \texttt{E} \texttt{<:} \texttt{U}}{\Delta \vdash \texttt{E} \lessdot_? ?^{\texttt{U}}_{\texttt{L}}} \qquad \text{USV-EQUALS}\quad \frac{\Delta \vdash \texttt{E} \texttt{<:} \texttt{T} \quad \Delta \vdash \texttt{T} \texttt{<:} \texttt{E}}{\Delta \vdash \texttt{E} \lessdot_? \texttt{T}}$$

Figure 6 Well-formedness and subtyping.

3.2 Typing

A context Γ is a finite mapping from variables to types, written $\overline{\texttt{x}} : \overline{\texttt{E}}$. A type context Δ is a finite relation between type variables and types. Each type variable `X` can be related to at most two types, some type $\texttt{T} \neq \texttt{X}$ and `X` itself (reflexive subtyping constraint). We write $\overline{\texttt{X}} \texttt{<:} \overline{\texttt{T}}$ for the non-reflexive part, which associates each type variable to its declared bound, and treat the reflexive constraints ($\texttt{X} \texttt{<:} \texttt{X}$) separately.

A class definition has its type arguments split in two parts, $\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}, \overline{\texttt{Y}} \triangleleft \overline{\texttt{Y}}\texttt{>}$, but during class instantiation we write just $\texttt{new C<}\overline{\texttt{E}}\texttt{>}$. The list of type arguments $\overline{\texttt{E}}$ has the length of the first part $\overline{\texttt{X}}$ from the class definition. For example an instance of the class definition $\texttt{class Example<X} \triangleleft \texttt{String}, \texttt{X} \triangleleft \texttt{X>}$ looks like `Example<String>`.

Fig. 6 defines the well-formedness judgment for types $\Delta \vdash \texttt{E OK}$ and two subtyping relations: $\Delta \vdash \texttt{E} \texttt{<:} \texttt{E}$ for top-level subtyping and $\Delta \vdash \texttt{W} \lessdot_? \texttt{W}$ for use-site variance subtyping, which is invoked by the former inside of type parameters.

We discuss the subtyping rules emphasizing the differences between FJ-TX and FGJ.

The rules for well-formed types are mostly unchanged. The new rule WF-WILDCARD guarantees that the lower bound of a wildcard is a subtype of its upper bound.

The rule S-Refl is restricted to types of the form `N`. This change enforces that reflexivity does not apply to type variables and wildcards.

Subtyping is transitive as usual (S-Trans). The rules S-Var and S-Class are similar to FGJ.

There are three new rules that manage wildcards.

The rules S-WC-Super and S-WC-Extends relate a wildcard type to its lower and upper bound, respectively.

The rule S-WC-Class introduces use-site variance by invoking a subsidiary relation, use-site variance subtyping $\lessdot_?$. This relation checks if the types in argument position induce covariant, contravariant, or invariant subtyping.

That is, `List<Integer>` is a subtype of `List<? extends Object>` due to use-site variance subtyping of the type arguments `Integer` $\lessdot_?$ $?^{\texttt{Object}}_{\bot}$.

For example, in combination with S-Trans we can conclude from `T <: T'` that $?^{\texttt{T}}_{\texttt{T}''}$ `<:` $?^{\texttt{T}'''}_{\texttt{T}'}$.

There are two rules for use-site variance subtyping. The rule USV-Extends enables covariance and contravariance for wildcards. The rules USV-Equals enforces invariant subtyping in all cases where the right-hand side is not a wildcard.

Finally, we consider the rules for typing classes, methods, fields, and expressions. Fig. 7 defines the judgments for expression typing $\Delta;\Gamma \vdash \texttt{e} : \texttt{W}$. Fig. 8 contains method typing, field typing, and class typing. The rules are similar to those of FGJ although there are two differences. First, the rule T-Field is added. This rule is needed to guarantee the soundness of top-level wildcard instantiation. If the subtyping $\texttt{C<}\overline{\texttt{E}}'\texttt{>} \texttt{<:} \texttt{C<}\overline{\texttt{E}}\texttt{>}$ holds for a class type, its field types `F` must satisfy $[\overline{\texttt{E}}'/\overline{\texttt{X}}]\texttt{F} \texttt{<:} [\overline{\texttt{E}}/\overline{\texttt{X}}]\texttt{F}$. We discuss the rationale for this typing rule in Example 5. Furthermore, we need an extension of the type variable instance, where we distinguish instantiations of type without top-level wildcards `T` and wildcard type `W`.

3.3 Examples

The following examples demonstrate the intuition of the calculus. The first example shows the idea of wildcards in the argument position of generic types.

The first example illustrates why the S-Refl rule in FJ-TX is restricted to ground types `N` (types without top-level type variables), rather than arbitrary types `T` (including generic type variables), and what consequences this has for methods whose type variables require a reflexive bound.

▶ **Example 4.** Consider the following method `shuffle`:

```
<X extends Object> List<List<X>> shuffle(List<List<X>> l){
  l.addElement(addElement(l.get(0).get(0)));
}
```

This method is not accepted. As `l.get(0).get(0)` has type `X` and `addElement` takes a value of type `X`, we need `X <: X`. However, subtyping is not reflexive on type variables!

To make the method acceptable, we have to change the bound of `X` from `Object` to `X` (a reflexive bound).

```
<X extends X> List<List<X>> shuffle(List<List<X>> l){
  l.addElement(addElement(l.get(0).get(0)));
}
```

Now the S-VAR rule enables us to conclude `X <: X`, such that the type checker accepts this method.

Due to the reflexive constraint, we cannot use `shuffle` with wildcards. Consider:

Field lookup:

$$fields(\texttt{Object}) = \emptyset \qquad \frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}\texttt{>} \triangleleft \texttt{N}\{\,\overline{\texttt{T}}\ \overline{\texttt{f}}; \ldots\} \qquad fields([\overline{\texttt{E}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{fields(\texttt{C<}\overline{\texttt{E}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}},\ [\overline{\texttt{E}}/\overline{\texttt{X}}]\overline{\texttt{T}}\ \overline{\texttt{f}}}$$

Method lookup:

M-Super

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}\texttt{>} \triangleleft \texttt{N}\{\,\overline{\texttt{T}}\ \overline{\texttt{f}}; \overline{\texttt{M}}\,\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C<}\overline{\texttt{E}}\texttt{>}) = mtype(\texttt{m}, [\overline{\texttt{E}}/\overline{\texttt{X}}]\texttt{N}) \qquad mbody(\texttt{m}, \texttt{C<}\overline{\texttt{E}}\texttt{>}) = mbody(\texttt{m}, [\overline{\texttt{E}}/\overline{\texttt{X}}]\texttt{N})}$$

M-Class

$$\frac{\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}\texttt{>} \triangleleft \texttt{N}\{\,\overline{\texttt{T}}\ \overline{\texttt{f}}; \overline{\texttt{M}}\,\} \qquad \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}'}\texttt{>}\ \texttt{T}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\{\texttt{return e;}\} \in \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C<}\overline{\texttt{E}}\texttt{>}) = [\overline{\texttt{E}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}'}\texttt{>}\overline{\texttt{T}} \to \texttt{T}) \qquad mbody(\texttt{m<}\overline{\texttt{F}}\texttt{>}, \texttt{C<}\overline{\texttt{E}}\texttt{>}) = \overline{\texttt{x}}.[\overline{\texttt{E}}/\overline{\texttt{X}}][\overline{\texttt{F}}/\overline{\texttt{Y}}]\texttt{e}}$$

Method Overriding:

$$\frac{mtype(m, \texttt{N}) \text{ undefined}}{override(m, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})} \qquad \frac{mtype(m, \texttt{N}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}}{override(m, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})}$$

Expression typing:

T-Var

$$\Delta; \Gamma \vdash \texttt{x} : \Gamma(\texttt{x})$$

T-Null

$$\texttt{null} : \bot$$

T-Access

$$\frac{\Delta; \Gamma \vdash \texttt{e} : \texttt{E} \qquad \Delta \vdash \texttt{E <: N} \qquad fields(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\Delta; \Gamma \vdash \texttt{e.f}_\texttt{i} : \texttt{T}_\texttt{i}}$$

T-New

$$\frac{fields(\texttt{C<}\overline{\texttt{F}}\texttt{>}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}\texttt{>}\{\ldots\} \qquad \Delta \vdash [\overline{\texttt{F}}/\overline{\texttt{X}}](\overline{\texttt{X}} \texttt{ <: } \overline{\texttt{U}}) \qquad \Delta; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \Delta \vdash \overline{\texttt{E}} \texttt{ <: } \overline{\texttt{T}}}{\Delta; \Gamma \vdash \texttt{new C<}\overline{\texttt{F}}\texttt{>}(\overline{\texttt{e}}) : \texttt{C<}\overline{\texttt{F}}\texttt{>}}$$

T-Invk

$$\frac{\begin{array}{c} mtype(\texttt{m}, \texttt{C<}\overline{\texttt{G}}\texttt{>}) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}'}\texttt{>}\overline{\texttt{T}} \to \texttt{T} \\ \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{U}}\texttt{>}\{\ldots\} \qquad \Delta; \Gamma \vdash \texttt{e} : \texttt{E} \qquad \Delta \vdash \texttt{E <: C<}\overline{\texttt{G}}\texttt{>} \\ \Delta \vdash [\overline{\texttt{G}}/\overline{\texttt{X}}](\overline{\texttt{X}} \texttt{ <: } \overline{\texttt{U}}) \qquad \Delta; \Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{E}} \qquad \Delta \vdash \overline{\texttt{E}} \texttt{ <: } [\overline{\texttt{F}}/\overline{\texttt{Y}}]\overline{\texttt{T}} \qquad \Delta \vdash [\overline{\texttt{F}}/\overline{\texttt{Y}}](\overline{\texttt{Y}} \texttt{ <: } \overline{\texttt{U}'}) \end{array}}{\Delta; \Gamma \vdash \texttt{e.<}\overline{\texttt{F}}\texttt{>m}(\overline{\texttt{e}}) : [\overline{\texttt{F}}/\overline{\texttt{Y}}]\texttt{T}}$$

T-UCast

$$\frac{\texttt{e : T} \qquad \texttt{T <: U}}{\texttt{(U)e : U}}$$

T-AnyCast

$$\frac{\texttt{e : T} \qquad \texttt{T} \not<: \texttt{U} \qquad \textit{unchecked cast warning}}{\texttt{(U)e : U}}$$

Figure 7 Expression Typing rules.

Method typing:

$$\textsc{T-Method}\quad \frac{\Delta = \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{T}'}, \overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{T}''} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}},\ \mathtt{this} : \mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e} : \mathtt{E} \qquad \Delta \vdash \mathtt{E} \mathtt{<:} \mathtt{T} \qquad \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T}'}\mathtt{>} \triangleleft \mathtt{N}\{\ldots\} \qquad \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{T}''}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})}{\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{T}''}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T}'}\mathtt{>}}$$

Field typing:

$$\textsc{T-Field}\quad \frac{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{U}}\mathtt{>} \triangleleft \mathtt{N}\{\ldots\} \qquad \Pi = \{\, (\overline{\mathtt{E}'}, \overline{\mathtt{E}}) \mid \emptyset \vdash \mathtt{C}\mathtt{<}\overline{\mathtt{E}'}\mathtt{>} \mathtt{<:} \mathtt{C}\mathtt{<}\overline{\mathtt{E}}\mathtt{>}, \emptyset \vdash \overline{\mathtt{E}} \mathtt{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}, \emptyset \vdash \overline{\mathtt{E}'} \mathtt{<:} [\overline{\mathtt{E}'}/\overline{\mathtt{X}}]\overline{\mathtt{U}} \,\} \qquad \mathit{fields}(\mathtt{C}\mathtt{<}\overline{\mathtt{X}}\mathtt{>}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \forall \overline{\mathtt{E}'}\ \forall \overline{\mathtt{E}} \in \Pi : \forall \mathtt{N}_i \in \overline{\mathtt{T}} : [\overline{\mathtt{E}'}/\overline{\mathtt{X}}]\mathtt{N}_i \mathtt{<:} [\overline{\mathtt{E}}/\overline{\mathtt{X}}]\mathtt{N}_i}{\Delta \vdash \overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}}$$

Class typing:

$$\textsc{T-Class}\quad \frac{\mathtt{class}\ \mathtt{D}\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{N}'}\mathtt{>} \triangleleft \mathtt{N}'\{\ldots\} \qquad \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}} \vdash [\overline{\mathtt{E}}/\overline{\mathtt{Y}}](\overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{N}'}) \qquad \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}\ \mathtt{D}\mathtt{<}\overline{\mathtt{E}}\mathtt{>}\ \mathtt{OK} \qquad \overline{\mathtt{T}}\ \overline{\mathtt{f}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \qquad \overline{\mathtt{M}}\ \mathtt{OK}\ \mathtt{in}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}}{\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D}\mathtt{<}\overline{\mathtt{E}}\mathtt{>}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

$$\textsc{T-Object}\quad \frac{}{\mathtt{class}\ \mathtt{Object} \triangleleft \mathtt{Object}\{\ \}\ \mathtt{OK}}$$

Figure 8 Class Typing rules.

```
void error(List<List<?>> anyL){
  shuffle<?>(anyL);
}
```

The method `error` is not typeable, because the GT-Invk rule requires $\Delta \vdash ? \mathtt{<:} [?/\mathtt{X}]\mathtt{X}$, which is not given. Any non wildcard instantiation of `X` would be accepted.

▶ **Example 5.** The T-Field rule declares that if `A` is a subtype of `B` (`A <: B`) then any field of `A` must be a subtype of the respective field in `B`. The example in Fig. 9 demonstrates why this property is needed to ensure soundness. Variable `b` of type `Box<?>` contains an instance of `Box<String>`, which is possible because `Box<String> <: Box<?>`. The field access expression `b.boxed` has type `Box<Box<?>>`, because `b` is of type `Box<?>`. But as we know, the type behind `b` is actually an instance of `Box<String>` meaning that the field `boxed` is an instance of `Box<Box<String>>`. The resulting problem is that `Box<Box<String>>` is not a subtype of `Box<Box<?>>`, leading to an unsound program. In this example, a `Box<Integer>` would be added to a `Box<Box<String>>`. We address this issue with the new T-Field rule. It forces us to add the reflexive constraint in the class definition of `Box` in Fig. 9 as in `class Box<X extends X>`, which prevents a type like `Box<?>` to exist.

```
class Box<X> {
    Box<Box<X>> boxed;
    void set(X value){ .. }
}

Box<?> b = new Box<String>(new Box<Box<String>>(null));
b.boxed.set(new Box<Integer>(1)); // error!
```

Figure 9 Unsound Field Access Example.

Computation:

$$\textsc{M-Body}\ \frac{\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{T}}\texttt{>}\{\ldots\overline{\texttt{M}},\ldots\}\qquad \texttt{<}\overline{\texttt{Y}}\triangleleft\overline{\texttt{T}}\texttt{>T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)}\{\texttt{ return e; }\}\in\overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{E}}\texttt{>},\texttt{C<}\overline{\texttt{F}}\texttt{>}) = \overline{\texttt{x}}.\texttt{e}[\overline{\texttt{E}}/\overline{\texttt{Y}}][\overline{\texttt{F}}/\overline{\texttt{X}}]}$$

$$\textsc{R-Invk}\ \frac{\texttt{mbody(}\ \texttt{m<}\overline{\texttt{E}}\texttt{>,N}\ \texttt{)} = \overline{\texttt{x}}.\texttt{e}}{\texttt{(new N(}\overline{\texttt{v}}\texttt{)).m<}\overline{\texttt{E}}\texttt{>(}\overline{\texttt{v}}\texttt{)} \longrightarrow [\texttt{(new N(}\overline{\texttt{v}}\texttt{))/this}][\overline{\texttt{v}}/\overline{\texttt{x}}]\texttt{e}}\qquad \textsc{R-Field}\ \frac{\mathit{fields}(\ N\ ) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\texttt{(new N(}\overline{\texttt{v}}\texttt{)).f}_\texttt{i} \longrightarrow \texttt{v}_\texttt{i}}$$

$$\textsc{C-Field}\ \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{e.f} \longrightarrow \texttt{e}'\texttt{.f}}\qquad \textsc{C-New}\ \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{new N(}\overline{\texttt{v}}\texttt{,e,}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{new N(}\overline{\texttt{v}}\texttt{,e}'\texttt{,}\overline{\texttt{e}}\texttt{)}}\qquad \textsc{C-Invk}\ \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{e.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{e}'\texttt{.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}}$$

$$\textsc{C-Invk-Param}\ \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{v.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{v}}\texttt{,e,}\overline{\texttt{e}}\texttt{)} \longrightarrow \texttt{v.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{v}}\texttt{,e}'\texttt{,}\overline{\texttt{e}}\texttt{)}}\qquad \textsc{C-Cast}\ \frac{\texttt{e} \longrightarrow \texttt{e}'}{\texttt{(T)e} \longrightarrow \texttt{(T)e}'}\qquad \textsc{R-Upcast}\ \frac{\texttt{N <: T}}{\texttt{(T)(new N(}\overline{\texttt{v}}\texttt{))} \longrightarrow \texttt{(new N(}\overline{\texttt{v}}\texttt{))}}$$

Errors:

$$\textsc{R-Null-Field}\ \texttt{null.f err}\qquad \textsc{R-Null-Invk}\ \texttt{null.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}\ \texttt{err}\qquad \textsc{R-Cast-Exception}\ \frac{\texttt{N} \not<: \texttt{T}}{\texttt{(T)(new N(}\overline{\texttt{v}}\texttt{))}\ \texttt{err}}\qquad \textsc{Err-Field}\ \frac{\texttt{e err}}{\texttt{e.f err}}$$

$$\textsc{Err-New}\ \frac{\texttt{e err}}{\texttt{new N(}\overline{\texttt{v}}\texttt{,e,}\overline{\texttt{e}}\texttt{)}\ \texttt{err}}\qquad \textsc{Err-Invk}\ \frac{\texttt{e err}}{\texttt{e.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}\ \texttt{err}}\qquad \textsc{Err-Invk-Param}\ \frac{\texttt{e err}}{\texttt{v.<}\overline{\texttt{F}}\texttt{>m(}\overline{\texttt{v}}\texttt{,e,}\overline{\texttt{e}}\texttt{)}\ \texttt{err}}\qquad \textsc{Err-Cast}\ \frac{\texttt{e err}}{\texttt{(T)e err}}$$

Figure 10 FJ-TX: Reduction Rules.

3.4 Operational semantics

Fig. 10 defines small-step operational semantics of FJ-TX. They include null pointer and cast exceptions (`e err`).

4 Soundness

In this section, we prove the type soundness theorem for FJ-TX.

▶ **Theorem 6** (Soundness). *Suppose that* $\emptyset;\emptyset \vdash \texttt{e} : \texttt{E}$ *and* $\texttt{e} \longrightarrow^* \texttt{e}'$*, then either:*

- $\texttt{e}'$ *is a value,*
- *there exists* $\texttt{e}' \longrightarrow \texttt{e}''$ *where* $\emptyset;\emptyset \vdash \texttt{e}'' : \texttt{E}'$ *with* $\texttt{E}' = \texttt{E}$ *or* $\emptyset \vdash \texttt{E}' \texttt{ <: } \texttt{E}$*, or*
- $\texttt{e}'$ *runs into an Exception* $\texttt{e}'\ \texttt{err}$*.*

We define values as a subset of irreducible expressions by the following grammar.

$\mathtt{v} ::= (\mathtt{new}\ \mathtt{N}(\overline{\mathtt{v}})) \mid \mathtt{null}$

We prove this theorem in the usual way by induction on the reduction sequence from type preservation (Thm. 12) and progress (Thm. 13).

We start with some lemmas about substitution. Their proofs are similar to the ones for other Featherweight Java calculi using existential types (e.g. [4], [3]).

Note that in our type system not all types are reflexive. Therefore saying $\emptyset \vdash \mathtt{E}' <: \mathtt{E}$ in the Soundness and the preservation theorem does not necessarily include the case where $\mathtt{E}' = \mathtt{E}$ and we have to state both possibilities. This is the biggest difference between this proof and proofs for similar calculi.

Bierhoff suspected in [3] that requiring wildcards to have a lower bound that is a subtype of their upper bound allows `null` values to be typed with the bottom type $\bot$ without any complications. Our calculus requires this property to achieve soundness regardless of whether a `null` expression is involved, and we can affirm that `null` caused no complications in our soundness proof.

▶ **Lemma 7** (Type substitution preserves subtyping). *If* $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} <: \mathtt{F}$ *and* $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{U}})$ *then* $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{F}$

Proof. By induction on the derivation $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} <: \mathtt{F}$:

Case S-Var. By inversion we obtain $\mathtt{E} = \mathtt{X_i}$ and $\mathtt{F} = \mathtt{U_i}$. From the assumption $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{U}})$, we have $\mathtt{G_i} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{U_i}$, which is the same as $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{X_i} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{U_i}$.

Case S-Refl is immediate.

Case For S-WC-Super and S-WC-Extends we obtain $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{T} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{T}'$ by induction, so the conclusion is immediate.

Case S-Trans $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} <: \mathtt{F}$ with $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} <: \mathtt{E}'$ and $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E}' <: \mathtt{F}$. By induction we have $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}'$ and $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}' <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{F}$. Then $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{F}$ by S-Trans.

Case S-Class. We have $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{C}\texttt{<}\overline{\mathtt{E}}\texttt{>} <: [\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\mathtt{N}$ and $\mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{T}}\texttt{>} \triangleleft \mathtt{N}$. By T-Class we know $fv(\mathtt{N}) \subseteq \overline{\mathtt{Y}}$ leading to $fv([\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\mathtt{N}) \subseteq \{\overline{\mathtt{X}}\}$ and from well-formedness $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \overline{\mathtt{E}} <: [\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$. We have to show well-formedness $[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{E}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$, that is, $[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{E}} <: [[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$, to conclude $\mathtt{C}\texttt{<}[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{E}}\texttt{>} <: [[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\mathtt{N}$ by S-Class. The latter is the same as $[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{C}\texttt{<}\overline{\mathtt{E}}\texttt{>} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{E}}/\overline{\mathtt{Y}}]\mathtt{N}$.

Case S-WC-Class. Suppose that $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{C}\texttt{<}\overline{\mathtt{E}}\texttt{>} <: \mathtt{C}\texttt{<}\overline{\mathtt{E}'}\texttt{>}$ and $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \overline{\mathtt{E}} \lessdot_? \overline{\mathtt{E}'}$. We have to show for every $\mathtt{E_i} \in \overline{\mathtt{E}}$ and $\mathtt{E}'_\mathtt{i} \in \overline{\mathtt{E}'}$ that $[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E_i} \lessdot_? [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}'_\mathtt{i}$. There are two subcases:

Case USV-Extends. Inversion applied to $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} \lessdot_? ?^U_L$ yields $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{L} <: \mathtt{E}$ and $\overline{\mathtt{X}} <: \overline{\mathtt{U}} \vdash \mathtt{E} <: \mathtt{U}$. Then $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{L} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}$ and $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{U}$ by induction hypothesis. Applying USV-Extends yields $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} \lessdot_? [\overline{\mathtt{G}}/\overline{\mathtt{X}}]?^U_L$.

Case USV-Equals is immediate by induction. ◀

▶ **Lemma 8** (Type substitution preserves typing). *If* $\overline{\mathtt{X}} <: \overline{\mathtt{U}}; \Gamma \vdash \mathtt{e} : \mathtt{E}$ *and* $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{U}})$ *then* $\emptyset; [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{e} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}$

Proof. We write $\Delta = \overline{\mathtt{X} <: \mathtt{U}}$ and proceed by induction over the derivation $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$.

Case T-Var. Assuming $\overline{\mathtt{X} <: \mathtt{U}}; \Gamma \cup \{\, \mathtt{x} : \mathtt{T} \,\} \vdash \mathtt{x} : \mathtt{T}$ we get after substitution $\emptyset; [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\Gamma \cup \{\, \mathtt{x} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{T} \,\} \vdash x : [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{T}$ and we finish this case by T-Var.

Case T-Null. Immediate.

Case T-Access. We have $\overline{\mathtt{X} <: \mathtt{U}}; \Gamma \vdash \mathtt{e.f} : \mathtt{E}$ where $\overline{\mathtt{X} <: \mathtt{U}}; \Gamma \vdash \mathtt{e} : \mathtt{E}_0$, $\overline{\mathtt{X} <: \mathtt{U}} \vdash \mathtt{E} <: \mathtt{N}$ and $fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ by assumption. $\emptyset, [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{e} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E}'_0$ by induction hypothesis. $[\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{N}$ by lemma 8 and $fields([\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{N}) = [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\ \overline{f}$ by definition of *fields* finishing the case.

Case T-New. By premise of T-New and induction hypothesis we get $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash \overline{e} : [\overline{G}/\overline{X}]\overline{E}$. By the definition of *fields* we get $\mathit{fields}(C\texttt{<}[\overline{G}/\overline{X}]\overline{F}\texttt{>}) = [\overline{G}/\overline{X}]\overline{T}\ \overline{f}$. By lemma 7 we get $\emptyset \vdash [\overline{G}/\overline{X}]\overline{E} <: [\overline{G}/\overline{X}]\overline{T}$, and $\emptyset \vdash [\overline{G}/\overline{X}][\overline{F}/\overline{X}](\overline{X} <: \overline{U})$. Therefore $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash \texttt{new}\ C\texttt{<}[\overline{G}/\overline{X}]\overline{F}\texttt{>}(\overline{e}) : C\texttt{<}[\overline{G}/\overline{X}]\overline{F}\texttt{>}$ finishing the case.

Case T-Invk. By premise of T-Invk we have $e = e_0.\texttt{<}\overline{F}\texttt{>}m(\overline{e})$ and $\mathit{mtype}(m, C\texttt{<}\overline{S}\texttt{>}) = \texttt{<}\overline{Y} \triangleleft \overline{P}\texttt{>}\overline{T} \to T$ and $\Delta; \Gamma \vdash e : E$ and $\Delta \vdash E <: C\texttt{<}\overline{S}\texttt{>}$ and $\Delta; \Gamma \vdash \overline{e} : \overline{E}$ and $\Delta \vdash \overline{E} <: [\overline{F}/\overline{Y}]\overline{T}$ and $\Delta \vdash \overline{F} <: [\overline{F}/\overline{Y}]\overline{P}$. By the inductive hypothesis we have $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash [\overline{G}/\overline{X}]e : [\overline{G}/\overline{X}]E$ and $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash [\overline{G}/\overline{X}]\overline{e} : [\overline{G}/\overline{X}]\overline{E}$.
By definition of *mtype* we have $\mathit{mtype}(m, [\overline{G}/\overline{X}]C\texttt{<}\overline{S}\texttt{>}) = [\overline{G}/\overline{X}](\texttt{<}\overline{Y} \triangleleft \overline{U}\texttt{>}\overline{T} \to T)$. By lemma 7 we obtain $\vdash [\overline{G}/\overline{X}]E <: [\overline{G}/\overline{X}]C\texttt{<}\overline{S}\texttt{>}$, $\vdash [\overline{G}/\overline{X}]\overline{E} <: [\overline{G}/\overline{X}][\overline{F}/\overline{Y}]\overline{T}$ and $\vdash [\overline{G}/\overline{X}]\overline{F} <: [\overline{G}/\overline{X}][\overline{F}/\overline{Y}]\overline{P}$ finishing the case by application of T-Invk.

Case T-UCast. We have $\overline{X} <: \overline{U}; \Gamma \vdash (U)e : U$ with $\overline{X} <: \overline{U}; \Gamma \vdash e : T$ and $\overline{X} <: \overline{U} \vdash T <: U$. $\emptyset; \Gamma \vdash [\overline{G}/\overline{X}]e : [\overline{G}/\overline{X}]T$ by induction hypothesis and $\vdash [\overline{G}/\overline{X}]T <: [\overline{G}/\overline{X}]U$ by lemma 7. Leading to $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash ([\overline{G}/\overline{X}]U)[\overline{G}/\overline{X}]e : [\overline{G}/\overline{X}]U$ by T-UCast.

Case T-AnyCast. We have $\overline{X} <: \overline{U}; \Gamma \vdash (U)e : U$ with $\overline{X} <: \overline{U}; \Gamma \vdash e : T$. $\emptyset; \Gamma \vdash [\overline{G}/\overline{X}]e : [\overline{G}/\overline{X}]T$ by induction hypothesis Leading to $\emptyset; [\overline{G}/\overline{X}]\Gamma \vdash ([\overline{G}/\overline{X}]U)[\overline{G}/\overline{X}]e : [\overline{G}/\overline{X}]U$ by T-AnyCast. ◀

We can substitute wildcard types, which are not reflexive, for free type variables because type variables are not reflexive either. Lemma 7 works as expected.

▶ **Lemma 9** (Term substitution preserves typing). *If* $\emptyset; \overline{x} : \overline{E} \vdash e : E$ *and* $\emptyset; \emptyset \vdash \overline{e} : \overline{G}$ *where* $\vdash \overline{G} <: \overline{E}$ *then* $\emptyset; \emptyset \vdash [\overline{e}/\overline{x}]e : F$ *with* $\vdash F <: E$

Proof. By induction over the derivation of $\emptyset; \overline{x} : \overline{E} \vdash e : E$.

Case T-Var. By inversion $\emptyset; \overline{x} : \overline{E} \vdash x_i : E_i$. From assumption $\vdash \overline{G} <: \overline{E}$ we have $\vdash S_i <: E_i$ and $[\overline{e}/\overline{x}]x_i = e_i$. Therefore $\emptyset; \emptyset \vdash [\overline{e}/\overline{x}]x_i : S_i$ and $\vdash S_i <: E_i$.

Case T-Null. Immediate.

Case T-New. We have $e = \texttt{new}\ N(\overline{e'})$ and $E = N$ with $\emptyset; \overline{x} : \overline{E} \vdash \overline{e'} : \overline{E'}$, $\mathit{fields}(N) = \overline{T}\ \overline{f}$, $\vdash \overline{E'} <: \overline{T}$. Induction yields $\vdash [\overline{e}/\overline{x}]\overline{e'} : \overline{F'}$ where $\vdash \overline{F'} <: \overline{T}$ by S-Trans and $\vdash \texttt{new}\ N([\overline{e}/\overline{x}]\overline{e'}) : N$ by T-New and induction hypothesis.

Case T-Access is similar to T-New. $\vdash e : F$ by induction where $\vdash F <: N$ by S-Trans and $\vdash e.f_i : T_i$ by T-Access as desired.

Case T-Invk is similar to T-New and T-Access.

Case T-UCast. We have $\emptyset; \overline{x} : \overline{E} \vdash (U)e : U$ with $\emptyset; \overline{x} : \overline{E} \vdash e : T$ and $\vdash T <: U$. $\emptyset; \emptyset \vdash [\overline{e}/\overline{x}]e : F$ with $\vdash F <: E$ by induction hypothesis following $\vdash F <: U$ by S-Trans and finally $\emptyset; \emptyset \vdash (U)[\overline{e}/\overline{x}]e : U$ by T-UCast.

Case T-AnyCast. Immediate. ◀

The following lemma 10 shows that fields can change their types to a subtype when occurring in a subtype. Example:

```
class Test<X>{
  Box<X> f;
}

Test<?> t = new Test<String>(); // t.f has type Box<?>
t.f = new Box<Integer>(); // because Box<Integer> \subeq{} Box<?>
```

The T-Field rule is necessary to address the problem illustrated in Fig. 9. When a new instance of the class `Box<String>` is created the field `boxed` is of type `Box<Box<String>>`. Later when treating this `Box` as a `Box<?>` its `boxed` field changes to `Box<Box<?>>` as well, but the actual value of `boxed` is still a `Box<String>`. This would violate the preservation of subtyping by substitution.

Interestingly, there is no corresponding problem for method calls. This sounds counterintuitive, because a field access can also be seen as a method call with no parameters. So let us rewrite the `Box` example in Fig. 9 to use a method instead of a field:

```
class Box<X extends X>{
  X value;
  Box<Box<X>> boxed(){
    return new Box<Box<X>>(new Box<X>(this.value));
  }
}
```

The method `boxed()` creates the `boxed` value on demand. Here the reflexivity constraint `X extends X` is enforced by the expression `new Box<X>(this.value)`, which demands `value` to have a subtype of `X`.

▶ **Lemma 10.** *If* $\vdash \mathtt{C<\overline{E}>} <: \mathtt{N}$ *and* $\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{f}$ *and* $\mathtt{class\ C<\overline{X} \triangleleft \overline{U}>}$ *and* $\vdash [\overline{\mathtt{E}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{U}})$ *then* $\mathit{fields}(\mathtt{C<\overline{E}>}) = \overline{\mathtt{T'}}\ \overline{f}, \overline{\mathtt{S}}\ \overline{g}$ *where* $\mathtt{T'_i} = \mathtt{T_i}$ *or* $\vdash \mathtt{T'_i} <: \mathtt{T_i}$

Proof. By induction over the subtype relation $\vdash \mathtt{C<\overline{E}>} <: \mathtt{N}$.

Case S-Var, S-WC-Super and S-WC-Extends cannot occur, because there are no type variables in this context.

Case S-Refl immediate.

Case S-Bot cannot occur.

Case S-Trans $\vdash \mathtt{C<\overline{E}>} <: \mathtt{E}$, $\vdash \mathtt{E} <: \mathtt{N}$. `E` can have one of the forms $\mathtt{N'}$ or $?^{\mathtt{U}}_{\mathtt{L}}$.

Case $\mathtt{E} = \mathtt{N'}$. This case is immediate by induction.

Case $\mathtt{E} = ?^{\mathtt{U}}_{\mathtt{L}}$. Then $\vdash \mathtt{C<\overline{E}>} <: ?^{\mathtt{U}}_{\mathtt{L}}$ and $\vdash ?^{\mathtt{U}}_{\mathtt{L}} <: \mathtt{N}$. Additionally $\mathtt{U} = \mathtt{N_U}$ and $\mathtt{L} = \mathtt{N_L}$, because there are no free variables. Only the rules S-WC-Super and S-WC-Extends are applicable and therefore we know $\vdash \mathtt{N_L} <: \mathtt{N_U}$, $\mathtt{C<\overline{E}>} <: \mathtt{N_L}$ and $\vdash \mathtt{N_U} <: \mathtt{N}$. Now induction hypothesis finishes this case.

Case S-Class immediate by definition of *fields*.

Case S-WC-Class $\vdash \mathtt{C<\overline{E}>} <: \mathtt{C<\overline{E'}>}$. This case is immediate by T-Field. ◀

▶ **Lemma 11.** *If* $\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{G}>}) = [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{<\overline{Y} \triangleleft \overline{U}>\overline{T}} \to \mathtt{T}$, $\mathtt{\overline{Y} \triangleleft \overline{U}}, \mathtt{\overline{X} \triangleleft \overline{U'}}; \mathtt{\overline{x}} : \mathtt{\overline{T}}, \mathtt{this} : \mathtt{C<\overline{X}>} \vdash \mathtt{e_0} : \mathtt{S}$ $\mathtt{class\ C<\overline{X} \triangleleft \overline{U'}> \triangleleft D<\overline{E'}>\{\ldots\}}$, $\vdash \mathtt{e} : \mathtt{E}$, $\vdash \mathtt{E} <: \mathtt{C<\overline{G}>}$, $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}](\overline{\mathtt{X}} <: \overline{\mathtt{U'}})$, *and* $\vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$ *with* $\vdash \overline{\mathtt{F}} <: [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ *and* $\vdash \overline{\mathtt{E}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$, *then* $\vdash [\mathtt{e'}/\mathtt{this}][\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e} : \mathtt{S}$ *with* $\vdash \mathtt{S} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$

Proof. By induction over the *mtype* function.

Case M-Class: By premise of M-Class the method `m` is in a class $\mathtt{C<\overline{X} \triangleleft \overline{N}>}$ and by T-Method we have $\mathtt{\overline{X} <: \overline{N}}, \mathtt{\overline{Y} <: \overline{U}}; \mathtt{\overline{x}} : \mathtt{\overline{T}}, \mathtt{this} : \mathtt{C<\overline{X}>} \vdash \mathtt{e} : \mathtt{E}$, $\mathtt{\overline{X} <: \overline{N}}, \mathtt{\overline{Y} <: \overline{U}} \vdash \mathtt{E} <: \mathtt{T}$.
By lemma 8 we get $\emptyset; \overline{\mathtt{x}} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}, \mathtt{this} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{C<\overline{X}>} \vdash \mathtt{e} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{E}$. Then by lemma 9 we get $\vdash [\mathtt{C<\overline{G}>}/\mathtt{this}][\overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e} : [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{E}$. $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$ by lemma 7 finishing the case.

Case M-Super $\mathit{mtype}(\mathtt{m}, \mathtt{C<\overline{G}>}) = \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{D<\overline{E'}>})$. $\mathtt{class\ D<\overline{X'} \triangleleft \overline{S'}> \triangleleft N}$ by T-Class. By S-Class $\mathtt{C<\overline{G}>} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{D<\overline{E}>}$ and $\mathtt{E} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\mathtt{D<\overline{E}>}$ by S-Trans. $\vdash [\overline{\mathtt{G}}/\overline{\mathtt{X}}][\overline{\mathtt{E'}}/\overline{\mathtt{X'}}](\overline{\mathtt{X'}} <: \overline{\mathtt{S'}})$ by lemma 7 using $\overline{\mathtt{X}} <: \overline{\mathtt{S}} \vdash [\overline{\mathtt{E'}}/\overline{\mathtt{X'}}](\overline{\mathtt{X'}} <: \overline{\mathtt{S'}})$ by T-Class. Then induction hypothesis finishes the case. ◀

Subtyping is not reflexive on type variables in our calculus, but it is on named types $\Delta \vdash \mathtt{N} <: \mathtt{N}$. This is because wildcard types are represented as interval types that can stand for any type inside of their bounds. When applying a computation step to an expression `e` the resulting expression $\mathtt{e'}$ does not necessarily have a subtype of the previous expression's type. For example a method call can have a wildcard type as its return type and an intermediate

state during computation might look like this: $\{\,\mathtt{x} : \mathtt{List<?}^{\mathtt{Object}}_{\perp}\mathtt{>}\,\} \vdash \mathtt{x.get}() : \mathtt{?}^{\mathtt{Object}}_{\perp}$. After a term substitution we get $\vdash \mathtt{new\ List<String>.get}() : \mathtt{?}^{\mathtt{Object}}_{\perp}$, but the return type remains the wildcard $\mathtt{?}^{\mathtt{Object}}_{\perp}$, which is not a subtype of itself. Therefore, the preservation theorem says that the result either has the same type or a subtype of the previous type.

▶ **Theorem 12** (Preservation). *If* $\vdash \mathtt{e} : \mathtt{E}$ *and* $\mathtt{e} \longrightarrow \mathtt{e'}$ *then there exists some* $\mathtt{E'}$ *with* $\vdash \mathtt{e'} : \mathtt{E'}$ *where either* $\mathtt{E'} = \mathtt{E}$ *or* $\vdash \mathtt{E'} <: \mathtt{E}$.

Proof. By structural induction over the $\mathtt{e} \longrightarrow \mathtt{e'}$ relation.

Case R-Field. $\mathtt{e} = (\mathtt{new\ C<}\overline{\mathtt{S}}\mathtt{>}(\overline{\mathtt{e}})).f_i$, $\mathtt{e'} = \mathtt{e}_i$ and $\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$. We get by T-New $\vdash \mathtt{new\ C<}\overline{\mathtt{S}}\mathtt{>}(\overline{\mathtt{e}}) : \mathtt{N} \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$ and $\vdash \overline{\mathtt{E}} <: \overline{\mathtt{T}}$, $\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T'}}\mathtt{>}$, $\vdash \overline{\mathtt{S}} <: [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{T'}}$.
We get by T-Access $\vdash \mathtt{N} <: \mathtt{N'}$ and $\mathit{fields}(\mathtt{N'}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}$ and $\mathtt{E} = \mathtt{U_i}$. By lemma 10 either $\vdash \mathtt{T_i} <: \mathtt{U_i}$ or $\mathtt{T_i} = \mathtt{U_i}$ and in both cases $\vdash \mathtt{E_i} <: \mathtt{U_i}$ (using S-Trans and $\vdash \overline{\mathtt{E}} <: \overline{\mathtt{T}}$). Letting $\mathtt{E'} = \mathtt{E_i}$ finishes the case.

Case C-Field: Given $\mathtt{e} = \mathtt{e}_0\mathtt{.f} \longrightarrow \mathtt{e}'_0\mathtt{.f}$ we get by T-Field $\vdash \mathtt{e} : \mathtt{E}$, $\vdash \mathtt{E} <: \mathtt{N}$ and $\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$. By hypothesis and S-Trans we get $\vdash \mathtt{e'} : \mathtt{T'}$ with $\vdash \mathtt{T'} <: \mathtt{N}$. Then letting $\mathtt{E'} = \mathtt{T_i}$ by lemma 10 finishing the case.

Case R-Invk: $\mathtt{e} = (\mathtt{new\ C<}\overline{\mathtt{G}}\mathtt{>}(\overline{\mathtt{d}}))\mathtt{.m<}\overline{\mathtt{F}}\mathtt{>}(\overline{\mathtt{e}})$ reduces to $[\mathtt{new\ C<}\overline{\mathtt{G}}\mathtt{>}(\overline{\mathtt{e}})/\mathtt{this}, \overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0$ with $\mathit{mbody}(\mathtt{m<}\overline{\mathtt{G}}\mathtt{>}, N) = \overline{\mathtt{x}}\mathtt{.e}$. We have by T-Invk and T-New: $\vdash \mathtt{new\ C<}\overline{\mathtt{G}}\mathtt{>}(\overline{\mathtt{e}}) : \mathtt{N}$, $\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}\{\ldots\}$, $\vdash \overline{\mathtt{G}} <: [\overline{\mathtt{G}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, $\vdash \mathtt{N} <: \mathtt{N'}$, $\mathit{mtype}(m, \mathtt{N'}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$, $\vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$, $\vdash \overline{\mathtt{E}} <: [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$ and $\vdash \mathtt{e} : [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$. Additionally $\mathit{mbody}(\mathtt{C<}\overline{\mathtt{G}}\mathtt{>}, m\mathtt{<}\overline{\mathtt{E}}\mathtt{>}) = \overline{\mathtt{x}}.\mathtt{e}_0$ by T-Method and *override*. By lemma 11 we get $\vdash [\mathtt{new\ C<}\overline{\mathtt{G}}\mathtt{>}(\overline{\mathtt{e}})/\mathtt{this}, \overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathtt{S}$ with $\vdash \mathtt{S} <: [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$ as desired.

Case C-Invk: $\mathtt{e.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{e}})$ reduces to $\mathtt{e'.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{e}})$, where $\mathtt{e} \longrightarrow \mathtt{e'}$. $\vdash \mathtt{e.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{e}}) : [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$ with $\mathit{mtype}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{U}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$, $\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{T'}}\mathtt{>}\{\ldots\}$, $\vdash \mathtt{e} : \mathtt{E}_0$, $\vdash \mathtt{E}_0 <: \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}$, $\vdash \overline{\mathtt{S}} <: [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{T'}}$, $\vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$, $\vdash \overline{\mathtt{E}} <: [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$, $\vdash \overline{\mathtt{F}} <: [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$ by T-Invk. $\vdash \mathtt{e'} : \mathtt{E}'_0$ with $\vdash \mathtt{E}'_0 <: \mathtt{E}_0$ by induction hypothesis and $\vdash \mathtt{E}'_0 <: \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}$ by S-Trans. By T-Invk we get $\vdash \mathtt{e'.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{e}}) : [\overline{\mathtt{F}}/\overline{\mathtt{Y}}]\mathtt{T}$ as desired.

Case C-Invk-Param $\mathtt{v.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{v}}, \mathtt{e}, \overline{\mathtt{e}}) \longrightarrow \mathtt{v.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{v}}, \mathtt{e'}, \overline{\mathtt{e}})$. This case is similar to C-Invk: We have $\mathtt{v.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{v}}, \mathtt{e}, \overline{\mathtt{e}})$ by T-Invk and $\vdash \mathtt{e'} : \mathtt{E}'_0$ with $\vdash \mathtt{E}'_0 <: \mathtt{E}_0$ or $\vdash \mathtt{E}'_0 = \mathtt{E}_0$ by induction hypothesis. In both cases $\mathtt{v.<}\overline{\mathtt{F}}\mathtt{>m}(\overline{\mathtt{v}}, \mathtt{e'}, \overline{\mathtt{e}})$ by T-Invk finishing the case.

Case C-New: Analogous to C-Invk-Param.

Case R-Null-Field. No reduction.

Case R-Null-Invk. No reduction.

Case C-Cast $\mathtt{(T)e} \longrightarrow \mathtt{(T)e'}$. $\vdash \mathtt{e} : \mathtt{E}$ by T-AnyCast. We know $\mathtt{e'} : \mathtt{E'}$ by i.h. where either $\mathtt{E'} = \mathtt{E}$ or $\mathtt{E'} <: \mathtt{E}$. In both cases $\vdash \mathtt{(T)e'} : \mathtt{T}$ by T-AnyCast finishing the case.

Case R-Upcast $\mathtt{(T)(new\ N}(\overline{\mathtt{v}})) \longrightarrow (\mathtt{new\ N}(\overline{\mathtt{v}}))$ with $\vdash \mathtt{(T)(new\ N}(\overline{\mathtt{v}})) : \mathtt{T}$. We know $\vdash \mathtt{new\ N}(\overline{\mathtt{v}}) : \mathtt{N}$ by T-New and we get $\vdash \mathtt{N} <: \mathtt{T}$ by the premise of R-Upcast finishing the case.

Case R-Cast-Exception. No reduction.

Case Err-Field, Err-New, Err-Invk, Err-Invk-Param, Err-Cast. No reduction. ◀

▶ **Theorem 13** (Progress). *If* $\vdash \mathtt{e} : \mathtt{E}$ *then either* $\mathtt{e}$ *is a value or* $\mathtt{e} \longrightarrow \mathtt{e'}$*, for some* $\mathtt{e'}$*, or* $\mathtt{e}$ `err`.

Proof. By induction over the typing derivation.

Case T-Var cannot occur ($\Gamma = \emptyset$)

Case T-New: By premise of T-New we have $\vdash \overline{\mathtt{e}} : \overline{\mathtt{E}}$ and either every $\mathtt{e}$ in $\overline{\mathtt{e}}$ is a value or we can apply C-New or Err-New by induction hypothesis.

Case T-Field $\mathtt{e} = \mathtt{e}_0\mathtt{.f}_i$: Either $\mathtt{e}_0$ is a value $\mathtt{e}_0 = \mathtt{new\ N'}(\overline{\mathtt{v}})$ with $\vdash \mathtt{N'} <: \mathtt{N}$ and $\mathit{fields}(\mathtt{N'}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{T'}}\ \overline{\mathtt{g}}$ by lemma 10 then we can apply R-Field. Otherwise we have $\vdash \mathtt{e}_0 : \mathtt{E}$ by premise of T-Field and therefore either a reduction $\mathtt{e}_0 \longrightarrow \mathtt{e}'_0$ or we can apply Err-Field and get $\mathtt{e}_0$ `err` by induction hypothesis.

```
<X> List<List<X>> make2DList(List<X> from){ ... }
<X> List<List<X>> shuffle2DList(List<List<X>> l){ ... }

List<?> input = ...;
shuffle2DList(make2DList(input));
```

Figure 11 Code snippet incompatible with Java-TX, but accepted by Java.

```
class NestedFields<X extends X> {
   Box<Box<X>> f; // X nested two levels deep -> X must be reflexive
}

NestedFields<?> nf = ...; // Not possible in Java-TX
```

Figure 12 Illicit type annotation, due to nested generics inside class body.

Case T-Invk $\texttt{e} = \texttt{e}_0.\texttt{<}\overline{\texttt{F}}\texttt{>m}(\overline{\texttt{v}}, \texttt{e}_\texttt{p}, \overline{\texttt{e}})$: Either $\texttt{e}_0 \longrightarrow \texttt{e}_0'$ because of $\vdash \texttt{e}_0 : \texttt{E}$ and induction hypothesis and we can apply C-Invk. Or the same for $\texttt{e}_\texttt{p} \longrightarrow \texttt{e}_\texttt{p}'$ and we can apply C-Invk-Param. If $\texttt{e}_0 = \texttt{e}_0'\ \texttt{err}$ we can apply Err-Invk and if $\texttt{e}_\texttt{p} = \texttt{e}_\texttt{p}'\ \texttt{err}$ we can apply Err-Invk-Param. Otherwise $\texttt{e} = (\texttt{new N}(\overline{\texttt{e}'})).\texttt{<}\overline{\texttt{F}}\texttt{>m}(\overline{\texttt{v}})$: By T-Invk we get $\mathit{mtype}(\texttt{m}, \texttt{N}') = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{U}'}\texttt{>}\overline{\texttt{T}} \to \texttt{T}$ with $\vdash \texttt{N <: N}'$. By definition of *override* $\mathit{mtype}(\texttt{m}, \texttt{N}) = \mathit{mtype}(\texttt{m}, \texttt{N}')$ and therefore $\mathit{mbody}(\texttt{m<}\overline{\texttt{F}}\texttt{>}, \texttt{N}) = \overline{\texttt{x}}.\overline{\texttt{e}'}$ where $\overline{\texttt{x}}$ has the same length as $\overline{\texttt{e}}$ and we can apply R-Invk.

Case T-UCast $\vdash (\texttt{T})\texttt{e} : \texttt{T}$. Either we have $\vdash \texttt{e} : \texttt{E}$ then C-Cast can be applied or we have $\vdash \texttt{v} : \texttt{E}$ and we can apply R-Upcast, because we have $\texttt{E <: T}$ by T-UCast. For the error case $\texttt{e} = \texttt{e}'\ \texttt{err}$ we can apply Err-Cast.

Case T-AnyCast $\vdash (\texttt{T})\texttt{e} : \texttt{T}$. Same as for T-UCast except that for the case that we have $\vdash \texttt{v} : \texttt{E}$ with $\vdash \texttt{E} \not<: \texttt{T}$ we have to apply R-Cast-Exception and end up with $\texttt{v err}$.

Case R-Null-Field, R-Null-Invk, R-Cast-Exception, Err-Field, Err-New, Err-Invk, Err-Invk-Param, Err-Cast all lead to $\texttt{e err}$. ◀

5 Practical Evaluation

As discussed earlier, the Java-TX type system deliberately supports only a subset of Java, because certain method calls can only be type-checked using Java's specific implementation of wildcards with capture conversion. Thus we consider the following research question for our evaluation: if we take an existing, well-typed Java program and remove all type annotations, can the Java-TX global type inference algorithm find a typing derivation? And if not, how often does it fail in real-world projects?

As no full-fledged compiler for Java-TX is available, yet, we cannot hope for a precise answer, but perform an experiment that provides an approximate answer. To this end, we compare Java-TX to existing formal models of Java with wildcards (e.g., [3, 4, 18]) and identify two principal points of divergence: (1) generic method invocations involving wildcard types, and (2) field declarations that include doubly nested generic type variables (like `Box<Box<X>>` in Figure 12). We will illustrate both points with concrete examples:

1. Figure 11 shows a method call which is rejected by Java-TX. In Java, the method call to `shuffle2DList(make2DList(input))` is made possible by utilizing Capture Conversion. Although the variable `input` has type `List<?>`, the Java compiler implicitly introduces

a fresh type variable `Cap#1` to capture the wildcard at the call site of `make2DList`. This process replaces the wildcard type `List<?>` with the concrete but locally bound type `List<Cap#1>`. The method `make2DList` then returns a value of type `List<List<Cap#1>>`. When this return value is passed as an argument to `shuffle2DList`, the type parameter `X` of the method can be instantiated with the captured type variable `Cap#1`, rendering the composed call typeable.

2. Java-TX's type system requires some class parameters to be reflexive as discussed in Section 3.3 example 5. For instance, the field declaration `f` in the class `NestedFields`, shown in Figure 12, demands a reflexive type parameter `X`. In consequence, a type annotation like `NestedFields<?>` is invalid in Java-TX.

To estimate how applicable Java-TX would be in practice, we investigate how often these two situations occur in existing Java code. For this purpose, we conduct an empirical study using a modified OpenJDK compiler that logs 1. capture-conversion events and 2. nested field types as explained. To obtain a safe approximation, we assume that every method call involving capture conversion cannot be handled by Java-TX and we assume the same for every occurrence of nested fields. In consequence, our measurements may overestimate the number of cases that actually cause problems for Java-TX. The details of this approximation are described in Section 5.1, the instrumentation and experimental setup are outlined in Section 5.2, and the quantitative findings are presented in Section 5.3.

5.1 The Good, the Bad, and the Ugly

The Java projects we studied contain three types of wildcard use cases: those that are compatible with the Java-TX type system, those that are certainly not, and those we cannot classify.

Calls to methods that are not generic, such as the one in Figure 13, are fully compatible with Java-TX. On the other hand, method calls like the one in Figure 14 are not possible in Java-TX, because the generic type variable `X` is used in a reflexive way within the method body. The third variant is the call shown in Figure 15, which is in fact valid in Java-TX, since the method's type parameter `X` is not reflexive. Unfortunately, our evaluation cannot distinguish between the second and the third case, which is why we conservatively treat all uncertain cases as incompatible with Java-TX. E.g. both method calls shown in Figure 14 and Figure 15 are counted as invalid even though the last one is supported by Java-TX.

Determining whether a generic type variable in a given Java method must be reflexive would require a full implementation of the Java-TX type checker for the entire Java language. This implementation is not yet available, hence we approximate this behavior by logging all method invocations that use capture conversion. We then conservatively classify only method calls to non-generic methods as valid and all others as invalid.

```
List<?> m(List<?> l){ // non-generic method
    return l;
}

List<?> l = ...;
m(l); // accepted in Java and Java-TX
```

Figure 13 Method call accepted by Java and Java-TX.

```
// <X extends X> would be required by Java-TX:
<X> List<X> sort(List<X> l){
    return l.add(l.get(0));
}

List<?> l = ...;
sort(l); //type-error!
```

Figure 14 Method call rejected by Java-TX, but accepted by Java's type system.

```
1 <X> // X is not reflexive
2 List<X> sort(List<X> l){
3     return l;
4 }
5
6 List<?> l = ...;
7 sort(l); // accepted!
```

Figure 15 Method call accepted by Java-TX, but counted as invalid.

Similar to the approximation for method calls, we also approximate Java-TX's restrictions on field declarations. We check class definitions for nested type variables and mark those fields as incompatible with Java-TX. Every type variable that appears anywhere other than in the topmost parameter list of a field type is counted as unsupported. Thus, a field like `f` in Figure 12 would be considered illegal by our evaluation. This provides a pessimistic, upper-bound approximation of the T-Field rule, formalized by Lemma 14, which states that every field type with no doubly nested generic type parameters poses no problem for the Java-TX type system.

▶ **Lemma 14.** *A field with type* $\mathtt{C}\langle\overline{\mathtt{E}}\rangle$ *containing type variables only in* $\overline{\mathtt{E}}$ *but not in any nested type parameter lists is a valid field type according to the rule* T-Field.
If $\mathtt{D}\langle\overline{\mathtt{E}}\rangle$ *where* $\forall \mathtt{E} \in \overline{\mathtt{E}} : \mathtt{E}$ *either is* $\mathtt{X}$ *or a wildcard* $\mathtt{W}$ *or a type* $\mathtt{N}$ *with* $fv(\mathtt{W}) = \emptyset$, $fv(\mathtt{N}) = \emptyset$
Then $\forall\overline{\mathtt{E_1}}\ \forall\overline{\mathtt{E_2}} \in \{\,\overline{\mathtt{E_1}} \mid \emptyset \vdash \mathtt{C}\langle\overline{\mathtt{E_1}}\rangle <: \mathtt{C}\langle\overline{\mathtt{E_2}}\rangle\,\} : [\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{E}}\rangle <: [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{E}}\rangle$

Proof. We will show $\forall\overline{\mathtt{E_1}}\ \forall\overline{\mathtt{E_2}} \in \{\,\overline{\mathtt{E_1}} \mid \emptyset \vdash \mathtt{C}\langle\overline{\mathtt{E_1}}\rangle <: \mathtt{C}\langle\overline{\mathtt{E_2}}\rangle\,\} : \forall \mathtt{E} \in \overline{\mathtt{E}} : \emptyset \vdash [\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]\mathtt{E} \lessdot_? [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]\mathtt{E}$ which implies $\forall\overline{\mathtt{E_1}}\ \forall\overline{\mathtt{E_2}} \in \{\,\overline{\mathtt{E_1}} \mid \emptyset \vdash \mathtt{C}\langle\overline{\mathtt{E_1}}\rangle <: \mathtt{C}\langle\overline{\mathtt{E_2}}\rangle\,\} : [\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]\mathtt{N} <: [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]\mathtt{N}$ by S-WC-Class.

Case $\mathtt{E} = \mathtt{X}$ then $[\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]\mathtt{X} \lessdot_? [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]\mathtt{X}$ is the same as $\emptyset \vdash \mathtt{E}_1 \lessdot_? \mathtt{E}_2$. We have $\emptyset \vdash \mathtt{C}\langle\overline{\mathtt{E_1}}\rangle <: \mathtt{C}\langle\overline{\mathtt{E_2}}\rangle$ by hypothesis which requires $\emptyset \vdash \overline{\mathtt{E_1}} \lessdot_? \overline{\mathtt{E_2}}$ by S-WC-Class finishing the case.

Case $\mathtt{E} = ?^{\mathtt{U}}_{\mathtt{L}}$ and $[\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]?^{\mathtt{U}}_{\mathtt{L}} = [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]?^{\mathtt{U}}_{\mathtt{L}}$, because $fv(?^{\mathtt{U}}_{\mathtt{L}}) = \emptyset$. USV-Extends and S-Refl finishing the case.

Case $\mathtt{E} = \mathtt{D}'\langle\overline{\mathtt{E}'}\rangle$ and $[\overline{\mathtt{E_1}}/\overline{\mathtt{X}}]\mathtt{D}'\langle\overline{\mathtt{E}'}\rangle = [\overline{\mathtt{E_2}}/\overline{\mathtt{X}}]\mathtt{D}'\langle\overline{\mathtt{E}'}\rangle$, because $fv(\mathtt{D}'\langle\overline{\mathtt{E}'}\rangle) = \emptyset$. USV-Equals and S-Refl finishing the case. ◀

5.2 Method

To perform this evaluation, we modify the Java Compiler to output information about every capture conversion that occurs during compilation. We take the compiler from the current Java implementation (JDK 24) of the OpenJDK project (`https://openjdk.org/`) and use it to compile open-source software from GitHub (`https://github.com/`).

Table 1 Method calls involving capture conversion.

Project	Method Calls	with CC	invalid in Java-TX
Java Development Kit	778339	8943	2109 (0.27%)
GitHub Projects	107899	1795	903 (0.84%)

Table 2 Classes that may not type check in Java-TX.

Project	Classes	invalid in Java-TX
Java Development Kit	31831	542 (1.70%)
GitHub Projects	5147	328 (6.37%)

We applied the modified compiler to the JDK itself and to a selection of trending Java projects on GitHub[2] in the time period from September to December 2024. The main criterion for choosing a project is that it compiles with the modified version of the OpenJDK, which means, the project has to be compatible with Java version 24. With this criterion, we ended up with seven open source Java projects compatible with our test setup:[3] graphhopper, nacos, commons-collections, DependencyCheck, maven, Algorithms-Java, commons-lang.

The biggest code base was OpenJDK's *Java Development Kit*[4] with over six million lines of Java code. The seven open source projects gathered from GitHub amount to a total of around seven hundred thousand lines of Java source code (measured with `cloc`[5]).

5.3 Result

Table 1 shows the number of method calls and the number of calls involving capture conversion in the code base. The *invalid* column shows the number of method calls that involve capture conversion and call a generic method.

Table 2 shows the breakdown in terms of classes: A class that contains at least one (potentially) invalid method call or field declaration is counted as invalid. The idea is that those classes may not be accepted by a type inference algorithm based on Java-TX.

Additionally we looked for field declarations that contain a type variable nested inside multiple layers. For example, the type `Box<X>` contains a type variable `X` nested inside `Box`. The *nested generics* column in Table 3 counts field declarations that contain type variables nested at least two levels deep (e.g. `Box<Box<X>>`). This is used as the approximation for the T-Field rule (see lemma 14). The *affected classes* are classes that contain at least one such field declaration.

5.4 Evaluation

The results support our claim that giving up capture conversion for our approach has only a small impact on existing Java code. In the real world projects we considered, around six percent of all classes may be not be compatible with Java-TX. However, we determined an upper limit of method calls which may pose problems for our calculus. Even if a class is counted as invalid, the Java-TX type inference algorithm may still find a typing! Thus,

[2] `https://github.com/trending/java`
[3] Each project name is linked to its GitHub page.
[4] `https://github.com/openjdk/jdk`
[5] `https://github.com/AlDanial/cloc`

Table 3 Classes with deeply nested field types.

Project	Fields	nested generics	affected classes
Java Development Kit	116717	81 (0.07%)	57 (0.18%)
GitHub Projects	13891	45 (0.32%)	28 (0.54%)

invalid just means that Java-TX may not find the exact same type annotations as the Java compiler. An alternative typing may not involve wildcards or the target method may not use its generic type variables in a reflexive way (cf. Fig. 15), in which case there is no problem.

We also scrutinized the unsupported method calls and found that many of those calls are targeting methods with isolated type variables. The next section 5.6 discusses them further and proposes a practical solution for those methods.

Our evaluation shows that Java-TX type inference is useful in real world source code. In fact, it could be used as an IDE plugin [10] enabling automated type completions and improvements, enabling the programmer to concentrate on providing explicit types for difficult cases involving capture conversion.

5.5 Directions for a More Thorough Evaluation

There are two principal ways to extend our evaluation.

Wider coverage. Our current approach requires manually running a modified `javac` compiler on each project. The advantage is that we count actual capture conversion events during the type-checking phase, which gives a precise (if conservative) measure. The downside is that it does not scale easily to a large number of projects. One alternative is to use the Boa infrastructure [5], which provides a query language over the ASTs of millions of open-source Java projects. While Boa cannot count capture conversions directly, it can count user-defined wildcard type annotations and methods with isolated type variables – the latter being a class of method calls that Java-TX can handle as described in Section 5.6. Another alternative is to adapt existing empirical studies on Java wildcards and generics to our use case. For instance, Altidor et al. [1] find that 32% of generic classes have single-variance type parameters and that 37% of existing wildcard uses are rendered unnecessary by their inference algorithm. Single-variance type parameters are compatible with Java-TX, since they do not require reflexivity and can therefore be instantiated with a wildcard. The problematic cases are invariant wildcards, which require use-site variance and depend on capture conversion – precisely the feature that Java-TX does not support.

Finer-grained classification. Our current approximation counts all method calls involving capture conversion on a generic method as incompatible with Java-TX, without distinguishing where the wildcard types and the called methods originate. Section 5.6 already shows one refinement: pre-typed library methods with isolated type variables can be handled by Java-TX and should not be counted as incompatible. A further refinement is to distinguish between user-defined and library methods, and between wildcard types originating in user code and those returned by library APIs. This distinction matters because the typical use case for Java-TX global type inference is to infer types for *user-written* code; library code is already fully type-annotated. The incompatibility between capture conversion and Java-TX type inference only arises when neither the types of the method parameters nor the types of the arguments are known to the inference algorithm. Concretely, three categories of call sites are worth distinguishing:

1. Calls where neither the called method nor the argument types are known – these are the genuinely problematic cases for Java-TX type inference.
2. Calls where all types are already known (e.g., both method and arguments are from library code) – here the Java type checker resolves the call directly and no type inference is needed.
3. Calls to user-defined methods where the argument type originates from a library function returning a wildcard – here it may be possible to support the call by applying capture conversion selectively at the boundary between library and user code.

5.6 Calls to Pre-typed Methods with Isolated Type Variables

A type inference algorithm can leverage the fact that external Java code included in a project (e.g., the Java standard library) is already type checked. Generally those methods could be converted to Java-TX by making their type parameters reflexive: `<X extends Object>` becomes `<X extends Object, X extends X>`.

```
// Pre-Typed Java Libraray
<X> void shuffle(List<X> list){ ... }
```

```
// Typeless Java-TX code:
List<?> anyList = ...;
shuffle(anyList); // accepted
```

Figure 16 Pre-Typed Method with Isolated Type Variable `X`, called from Java-TX program.

If a type parameter appears only once in a method header, this conversion can be safely omitted without compromising soundness. For instance, the `shuffle` method shown in Figure 16 has a type parameter `X` that occurs only once in the method header, namely in the parameter type `List<X>`. In Java this method can be invoked with an argument of type `List<?>`. Therefore we can treat `X` as a non-reflexive type argument in our Java-TX type system while still preserving soundness.

Furthermore we now can look for methods in the Java standard library with a similar pattern and treat their isolated type arguments also as non-reflexive. This allows our approximation to count more method calls as possible in Java-TX. One prominent example is the `collect` method of the Java Stream API:

```
class Stream<T> { <R,A> R collect(Collector<?  super T,A,R> collector) }
```

The type parameter `A` is used only once in the method header. therefore fits our requirements to be eligible for wildcards. The calls to `collect` using `Collectors.toList()` as a parameter instantiate `A` with a wildcard type and were counted as invalid by our evaluation as explained in Section 5.1. But these calls would all be sound under Java-TX's interpretation of wildcards, due the fact that those calls are invoking a pre-typed method with an isolated type variable.

In total (including JDK source code) we found 10785 uses of capture conversion in method calls of which 3012 are used to call a polymorphic method. So over 70% of method calls utilizing capture conversion are compatible with Java-TX anyway. Using the method proposed in this section we could additionally support 948 of those 3012 incompatible method calls pushing this number to 80% support for method calls using capture conversion.

6 Related Work

Igarashi et al [8] define Featherweight Java and its generic sibling, Featherweight Generic Java. This language is a functional core calculus that embodies the essential ingredients of Java. They develop the full metatheory for FJ and FGJ and study the type erasure transformation used by the Java compiler.

Wildcards are first described in a research paper by Torgersen et al [19]. Subsequently, they propose Wild FJ as an extension of FGJ with wildcards [18], but without any proofs of their formal system. The Java Language Specification [6] refers to Wild FJ for the introduction of wildcards. Cameron et al [4] propose a refined formal model of wildcards based on explicit existential types. They give a soundness proof and a translation of a subset of Java to the formal model. Bierhoff [3] gives a subtly different core calculus with a soundness proof. The motivation for this paper is to show that the unsoundness of Java which was discovered by Amin and Tate [2] is avoidable, even in the absence of a nullness-aware type system.

Altidor et al [1] present a framework which combines use-site variance (wildcards as in Java) and definition-site variance (as in Scala). For instance, it can be used to add use-site variance to Scala and extend the Java type system to infer the definition-site variance.

All these approaches rely on capture conversion to process arguments of wildcard type in calls to generic methods. As in our proposal, subtyping is *not* reflexive on wildcards in the systems of Cameron et al [4] and Bierhoff [3]. As our calculus instantiates type variables with wildcards, our subtyping relation is not reflexive on type variables. This restriction can be amended with reflexivity constraints, which in turn rule out instantiation with wildcards to retain soundness.

The type inference algorithm of Java-TX [13] treats wildcards similar to our approach. In particular, the subtyping relation is similar to ours. Their main contribution is the algorithm for type inference along with its soundness and completeness proof. Furthermore, Plümicke [15] proposes a similar calculus than considered here, but its soundness is not considered.

7 Summary and Outlook

This paper introduced the calculus FJ-TX, a core calculus for the language Java-TX. The calculus is based on a novel approach to deal with wildcards. Contrary to the mainstream approach, which is implemented in the Java compiler, our calculus avoids the notion of capture conversion. This design choice enables us to treat wildcards like any other type, instantiate type variables with wildcards, and thus support global type inference. Our approach is implemented in the Java-TX compiler.

Both approaches guarantee soundness of using wildcards in combination with generic types. As we have shown in Section 2.3, our approach accepts some programs that are rejected by capture conversion and vice versa. Further research and experimentation is needed to investigate which approach is more manageable and intuitive in practice. For some programmers the restriction that subtyping is not reflexive on type variables might not be intuitive. However, type inference of Java-TX will silently insert the additional reflexivity constraints without programmer interaction. For others the concept of capture conversion with its complex error messages may be confusing.

References

1 John Altidor, Shan Shan Huang, and Yannis Smaragdakis. Taming the wildcards: combining definition- and use-site variance. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 602–613. ACM, 2011. doi:10.1145/1993498.1993569.

2 Nada Amin and Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In Eelco Visser and Yannis Smaragdakis, editors, *OOPSLA 2016*, pages 838–848. ACM, 2016. doi:10.1145/2983990.2984004.

3 Kevin Bierhoff. Wildcards need witness protection. *Proc. ACM Program. Lang.*, 6(OOPSLA2), 2022. doi:10.1145/3563301.

4 Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for Java with wildcards. *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, 5142:2–26, 2008. doi:10.1007/978-3-540-70592-5_2.

5 Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, Hyderabad, India, May 31 – June 7, 2014*, pages 779–790. ACM, 2014. doi:10.1145/2568225.2568295.

6 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java® Language Specification*. Addison-Wesley, Java SE 21 edition, 2023. URL: https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf.

7 Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Proceedings of the ACM SIGPLAN Conference OOPSLA*, 1999.

8 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

9 Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding type changes in Java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 629–641, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3409725.

10 Ruben Kraft and Martin Plümicke. Ein Language-Server für Java-TX. In Stefan Brunthaler, editor, *23. Kolloquium Programmiersprachen und Grundlagen der Programmierung – Vorläufiger Tagungsband*. Universität der Bundeswehr München, September 2025. (in german).

11 Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 803–813, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2568225.2568317.

12 Martin Pluemicke. Completing the functional approach in object-oriented languages. In *A Second Soul: Celebrating the Many Languages of Programming - Festschrift in Honor of Peter Thiemann's Sixtieth Birthday,* Freiburg, Germany, 30th August 2024, volume 413 of *Electronic Proceedings in Theoretical Computer Science*, pages 43–56. Open Publishing Association, 2024. doi:10.4204/EPTCS.413.4.

13 Martin Plümicke. Java type unification with wildcards. In *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2007. doi:10.1007/978-3-642-00675-3_15.

14 Martin Plümicke. Optimization of the Java Type Unification. In Sibylle Schwarz and Mario Wenzel, editors, *Proceedings of the 37th Workshop on (Constraint) Logic Programming (WLP 2023)*, 2023. URL: https://dbs.informatik.uni-halle.de/wlp2023/WLP2023_Pl%C3%BCmicke_Optimization%20of%20the%20Java%20Type%20Unification.pdf.

15 Martin Plümicke. Featherweight-Java-TX: A Minimal Core Calculus for Java-TX (FJ-TX). In Daniel Holle, Jens Knoop, Martin Plümicke, Peter Thiemann, and Baltasar Trancón y Widemann, editors, *40. Workshop der GI-Fachgruppe "Programmiersprachen und Rechenkonzepte"*, number 02/2024 in INSIGHTS – Schriftenreihe der Fakultät Technik, pages 93–101, Bad Honnef, Germany, April 2024. URL: `https://www.dhbw-stuttgart.de/forschung-transfer/technik/schriftenreihe-insights`.

16 Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 23–34, New York, NY, USA, 2017. ACM. `doi:10.1145/3132190.3132203`.

17 Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. Global Type Inference for Featherweight Generic Java. *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, 222:28:1–28:27, 2022. `doi:10.4230/LIPIcs.ECOOP.2022.28`.

18 Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In Philip Wadler, editor, *Proceedings of FOOL 12*, Long Beach, California, USA, 2012. ACM, School of Informatics, University of Edinburgh. URL: `http://homepages.inf.ed.ac.uk/wadler/fool/`.

19 Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, December 2004. `doi:10.5381/jot.2004.3.11.a5`.