

A Variation on Java Wildcards

Trading Expressiveness for Global Type Inference

Andreas Stadelmeier (DHBW), Martin Plümicke (DHBW),

Peter Thiemann (University of Freiburg)

Research Focus – Programming Languages

<https://www.dhbw.pl>

Java Data Structures Are Mutable

```
class Box<A> {  
    A value; // mutable field  
  
    void set(A in){ this.value = in; }  
  
    A get() { return this.value; }  
}
```



What Is Capture Conversion Good For?



```
Box<String> boxString = ...  
Box<Object> boxObject = ...  
Box<?> randomBox = ...
```

```
randomBox = boxString;    // ✓ correct  
randomBox = boxObject;    // ✓ correct  
randomBox.set(42);        // ERROR! read-only
```

What Is Capture Conversion Good For?

```
<A> void repack(Box<A> b) {
    b.set(b.get());
}
```



```
Box<String> boxString = ...
Box<Object> boxObject = ...
Box<?> randomBox = ...
```

```
randomBox = boxString;    // ✓ correct
randomBox = boxObject;    // ✓ correct
randomBox.set(42);        // ERROR! read-only
```

```
repack(randomBox);    // ✓ correct! Thanks to CC!
```

Introducing Java-TX: Java with Global Type Inference

```
<X> void repack(Box<X> box) {  
    box.set(box.get());  
}
```



```
repack(box) {  
    box.set(box.get());  
}
```

Java-TX

Introducing Java-TX: Java with Global Type Inference

```
<X> void repack(Box<X> box) {  
    box.set(box.get());  
}
```



Our Goal: Global Type Inference

Global means that **ALL** type annotations are optional

```
repack(box) {  
    box.set(box.get());  
}
```

Java-TX

Motivation: Capture Conversion Is Hard for Global Type Inference

- For standard Java (with wildcards), **no sound and complete** global type inference algorithm exists
- **Problem:** Wildcard types and Capture Conversion

Motivation: Capture Conversion Is Hard for Global Type Inference

- For standard Java (with wildcards), **no sound and complete** global type inference algorithm exists
- **Problem:** Wildcard types and Capture Conversion
- **Solution:** Introducing our own language **Java-TX**
 - with a different approach at Wildcard types
 - It's type system is compatible with an existing type inference algorithm from DHBW Stuttgart
 - **but** Wildcards in Java-TX are not as Expressive as in Java

Motivation: Capture Conversion Is Hard for Global Type Inference

- For standard Java (with wildcards), **no sound and complete** global type inference algorithm exists
- **Problem:** Wildcard types and Capture Conversion
- **Solution:** Introducing our own language **Java-TX**
 - with a different approach at Wildcard types
 - It's type system is compatible with an existing type inference algorithm from DHBW Stuttgart
 - **but** Wildcards in Java-TX are not as Expressive as in Java

Trading Expressiveness for Global Type Inference

Introducing Java-TX Wildcards

Java-TX

```
// Wildcards in Java-TX:  
Box<?> randomBox = ...;  
randomBox = new Box<String>();    // ✓ correct  
randomBox = new Box<Integer>();   // ✓ correct
```

Introducing Java-TX Wildcards

Java-TX

```
// Wildcards in Java-TX:  
Box<?> randomBox = ...;  
randomBox = new Box<String>();    // ✓ correct  
randomBox = new Box<Integer>();   // ✓ correct  
  
<A> A unpack(Box<A> b) { return b.get(); }  
  
unpack(randomBox);    // ✓ correct in Java-TX  
<?>unpack(randomBox);
```

Introducing Java-TX Wildcards

Java-TX

```
// Wildcards in Java-TX:  
Box<?> randomBox = ...;  
randomBox = new Box<String>();    // ✓ correct  
randomBox = new Box<Integer>();   // ✓ correct
```

```
<A> A unpack(Box<A> b) { return b.get(); }
```

```
unpack(randomBox);    // ✓ correct in Java-TX  
<?>unpack(randomBox);
```



Wildcards are treated as regular types

Problem: Type Variables Need Reflexivity

```
<X> Pair<X,X> swap(Pair<X,X> in) {
    X temp = in.left;
    in.left = in.right;
    in.right = temp;
}
```



```
Pair<?,?> pair = new Pair<String,Integer>();
swap(pair); // ERROR!
```

Problem: Type Variables Need Reflexivity

Java-TX

```
<X> Pair<X,X> swap(Pair<X,X> in) {  
    X temp = in.left;  
    in.left = in.right;  
    in.right = temp;  
}
```

```
Pair<?,?> pair = new Pair<String,Integer>();  
<?>swap(pair); // ERROR!
```

Problem: Type Variables Need Reflexivity

Java-TX

```
<X> Pair<X,X> swap(Pair<X,X> in) {  
    X temp = in.left;  
    in.left = in.right;  
    in.right = temp;  
}
```

```
Pair<?,?> pair = new Pair<String,Integer>();  
<?>swap(pair); // ERROR!
```

↑
PROBLEM: using ? as a type
parameter for X leads to unsoundness

Problem: Type Variables Need Reflexivity

Java-TX

```
<X> Pair<X,X> swap(Pair<X,X> in) {  
    X temp = in.left;  
    in.left = in.right;  
    in.right = temp;  
}
```

```
Pair<?,?> pair = new Pair<String,Integer>();  
<?>swap(pair); // ERROR!
```

↑
PROBLEM: using ? as a type
parameter for X leads to unsoundness

In Java-TX type parameters are not reflexive per default

Solution: Enforcing Reflexivity with `X extends X`

```
<X extends X> Pair<X,X> swap(Pair<X,X> in) {  
    X temp = in.left;  
    in.left = in.right;  
    in.right = temp;  
}
```

Java-TX

```
Pair<?,?> pair = new Pair<String,Integer>();  
<?>swap(pair); // ERROR! ? does not extend ?
```

Solution: Enforcing Reflexivity with `X extends X`

Java-TX

```
<X extends X> Pair<X,X> swap(Pair<X,X> in) {
    X temp = in.left;
    in.left = in.right;
    in.right = temp;
}
```

```
Pair<?,?> pair = new Pair<String,Integer>();
<?>swap(pair); // ERROR! ? does not extend ?
```

Global Type Inference will add
this annotation based on the method body

Incompatible With Java-TX: Reflexive Methods

Java-TX

```
<X extends X> void repack(Box<X> b) {  
    b.set(b.get());  
}
```

```
Box<?> box = ...;  
repack(box);    // incorrect in Java-TX  
                // but correct in Java (via CC)
```

Can Java-TX Be Used in Practice?

Can Java-TX Be Used in Practice?

Projects analysed

- JDK source (6.5M lines)
- 7 GitHub projects: graphhopper, nacos, commons-collections, maven, ... (700k lines)

Can Java-TX Be Used in Practice?

Projects analysed

- JDK source (6.5M lines)
- 7 GitHub projects: graphhopper, nacos, commons-collections, maven, ... (700k lines)

Results

Project	Method Calls	involving Wildcards	invalid in Java-TX
Java Development Kit	778 339	8 943	2 109 (0.27%)
GitHub Projects	107 899	1 795	903 (0.84%)

The Analysis Is a Pessimistic Estimate

We counted *every* CC on a polymorphic method call as incompatible.

- But Java-TX Wildcards still supports some method calls involving Wildcards:

The Analysis Is a Pessimistic Estimate

We counted *every* CC on a polymorphic method call as incompatible.

- But Java-TX Wildcards still supports some method calls involving Wildcards:

```
<A> A unbox(Box<A> l) { return l.get(0); }
```

Java-TX

```
Box<?> randomBox = ...;  
unbox(randomBox);    // ✓ possible in Java-TX
```

The Analysis Is a Pessimistic Estimate

We counted *every* CC on a polymorphic method call as incompatible.

- But Java-TX Wildcards still supports some method calls involving Wildcards:

```
<A> A unbox(Box<A> l) { return l.get(0); }
```

Java-TX

```
Box<?> randomBox = ...;  
unbox(randomBox);    // ✓ possible in Java-TX
```

- We suspect the real incompatibility rate is much lower

Case Study: `collect` and `toList()`

A recurring incompatible pattern:

```
<A, B extends B, C> void collect(Collector<A,B,C> f) {...}
```

Java-TX

```
list.stream().collect(Collectors.toList());  
// Collectors.toList() returns Collector<..., ?, ...>
```

Case Study: `collect` and `toList()`

A recurring incompatible pattern:

```
<A, B extends B, C> void collect(Collector<A,B,C> f) {...}
```

Java-TX

```
list.stream().collect(Collectors.toList());  
// Collectors.toList() returns Collector<..., ?, ...>
```

Error in Java-TX

Case Study: `collect` and `toList()`

A recurring incompatible pattern:

```
<A, B extends B, C> void collect(Collector<A,B,C> f) {...}
```

Java-TX

```
list.stream().collect(Collectors.toList());  
// Collectors.toList() returns Collector<..., ?, ...>
```

Error in Java-TX

Nearly **30 %** of all invalid method calls in the 7 open-source projects stem from the Java Streams library.

Fix: Remove the Wildcard Annotation in `toList()`

Original (incompatible):

```
public static <T>  
    Collector<T, ?, List<T>>   
    toList() {  
        return new CollectorImpl<>(  
            ArrayList::new, ...);  
    }
```

Fix: Remove the Wildcard Annotation in `toList()`

Original (incompatible):

```
public static <T>  
    Collector<T, ?, List<T>>  
    toList() {  
        return new CollectorImpl<>(  
            ArrayList::new, ...);  
    }
```



Fixed (compatible):

```
public static <T>  
    Collector<T, Object, List<T>>  
    toList() {  
        return new CollectorImpl<>(  
            ArrayList::new, ...);  
    }
```

Java-TX

Conclusion

Contributions

- FJ-TX: Featherweight Java with restricted wildcards
- No capture conversion—supports global type inference
- Type soundness proof (Preservation + Progress)

Conclusion

Contributions

- FJ-TX: Featherweight Java with restricted wildcards
- No capture conversion—supports global type inference
- Type soundness proof (Preservation + Progress)

Results

- $\approx 95\%$ of Java classes are already compatible
- Global Type Inference can increase that number

Questions?

A Variation on Java Wildcards

Trading Expressiveness for Global Type Inference

Stadelmeier • Plümicke • Thiemann

ECOOP 2026

A Genuinely Incompatible Example

```
class NotJavaTXCompatible {
    Box<?> randomBox;    // mutable field

    <X extends X> void repack(Box<X> b) { ... }

    void test() {
        repack(this.randomBox);    // not possible in Java-TX
                                   // possible in Java via CC
    }
}
```



Backup: addAll — Wildcard vs. Generic

With wildcard:

```
class List<A> {  
    void addAll(  
        List<? extends A> l){...}  
}
```



Could also be typed as:

```
class List<A> {  
    <B extends A>  
    void addAll(List<B> l){...}  
}
```

Java-TX

Both forms are semantically equivalent. The wildcard form is incompatible with Java-TX; the generic form is compatible and can be inferred by GTI.