Java-TX Compiler in Java-TX

Bad Honnef 2024

Julian Schmidt

https://www.dhbw-stuttgart.de

Motivation I

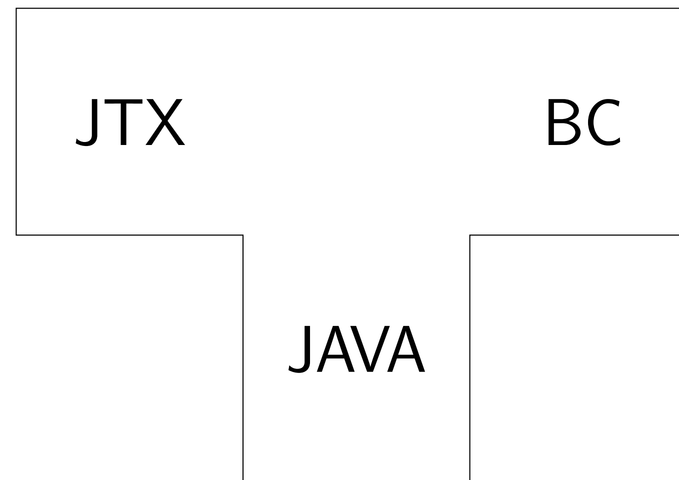
- Welche Features fehlen noch in Java-TX?
- Welche Bugs gibt es?
- Vorteile/Nachteile zu Java in der Praxis
- Wie performant ist Java-TX für „größere“ Projekte?

Motivation I

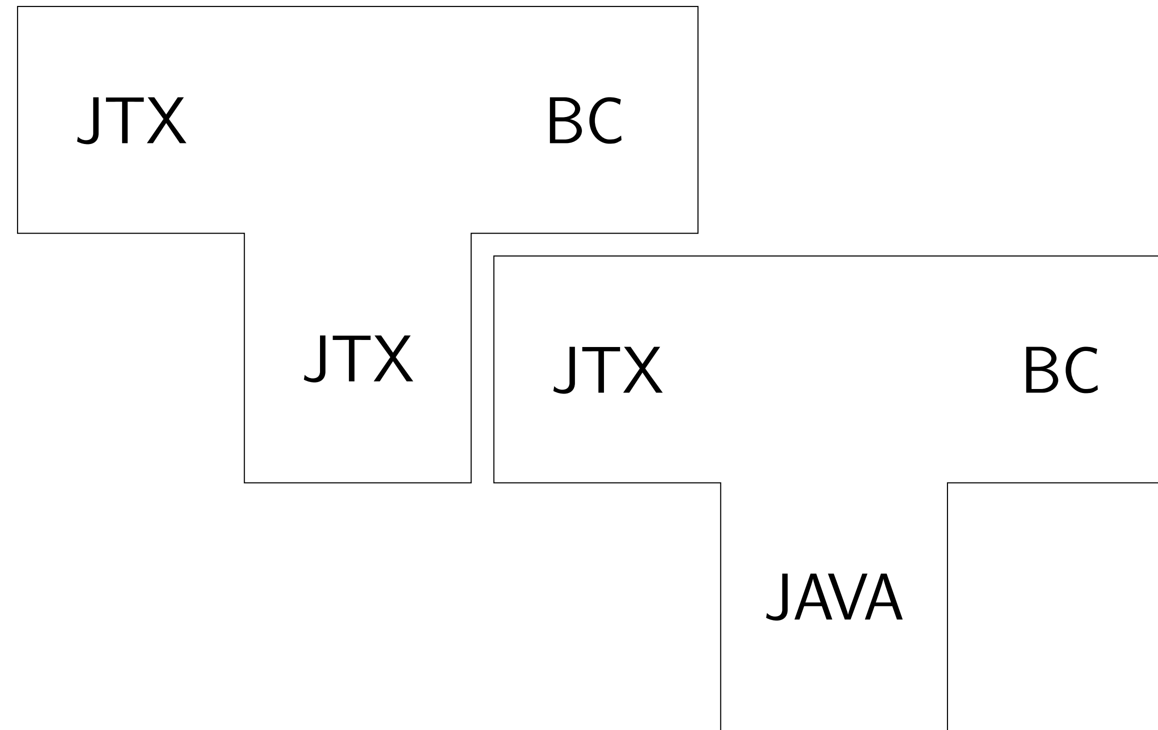
- Welche Features fehlen noch in Java-TX?
- Welche Bugs gibt es?
- Vorteile/Nachteile zu Java in der Praxis
- Wie performant ist Java-TX für „größere" Projekte?

Language	files	code
Java	247	17958
ANTLR Grammar	2	771
SUM	249	18729

Motivation II



Motivation II



JavaTX

- Programmiersprache basierend auf Java 8
- Globale Typinferenz
- Lambda-Ausdrücke sind getypt
- Überladung von Funktionstypen
- Automatisches Überladen von Funktionen
- Automatisches generieren von Generics
- Autoboxing von primitiven Datentypen

Vergleich Sourcecode Java - Java-TX

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
3    createGenerics(
4      List<GenericRefType> funNParams) {
5
6      var generics =
7      new ArrayList<GenericTypeVar>();
8
9      for (GenericRefType param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static
3    createGenerics(
4      funNParams) {
5
6      var generics =
7      new ArrayList<>();
8
9      for (param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

Vergleich Sourcecode Java - Java-TX

```java
public class FunNClass extends ClassOrInterface {
  private static GenericDeclarationList
  createGenerics(
    List<GenericRefType> funNParams) {

    var generics =
    new ArrayList<GenericTypeVar>();

    for (GenericRefType param : funNParams) {
      generics.add(...);
    }
    return new GenericDeclarationList(generics, new NullToken());
  }
}
```

```java
public class FunNClass extends ClassOrInterface {
  private static
  createGenerics(
    funNParams) {

    var generics =
    new ArrayList<>();

    for (param : funNParams) {
      generics.add(...);
    }
    return new GenericDeclarationList(generics, new NullToken());
  }
}
```

Vergleich Sourcecode Java - Java-TX

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
3    createGenerics(
4      List<GenericRefType> funNParams) {
5
6      var generics =
7      new ArrayList<GenericTypeVar>();
8
9      for (GenericRefType param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
3    createGenerics(
4      Iterable<? extends GenericRefType> funNParams) {
5
6      ArrayList<GenericTypeVar> generics =
7      new ArrayList<GenericTypeVar>();
8
9      for (GenericRefType param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

Vergleich Sourcecode Java - Java-TX

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
3    createGenerics(
4      List<GenericRefType> funNParams) {
5
6      var generics =
7      new ArrayList<GenericTypeVar>();
8
9      for (GenericRefType param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

```
1  public class FunNClass extends ClassOrInterface
     {
2    private static GenericDeclarationList
3    createGenerics(
4      Iterable<? extends GenericRefType> funNParams) {
5
6      ArrayList<GenericTypeVar> generics =
7      new ArrayList<GenericTypeVar>();
8
9      for (GenericRefType param : funNParams) {
10       generics.add(...);
11     }
12     return new GenericDeclarationList(generics,
         new NullToken());
13   }
14 }
```

Aufbau der Umgebung I

1. Compiler soll sukzessive in Java-TX umgeschrieben werden
2. Umgebung mit .java und .jav Dateien
3. Ziel: Auf JVM ausführbare .class Dateien

Aufbau der Umgebung II - make

Erster Ansatz mit make:

```
1  # Use find to locate all .java and .jav files recursively
2  JAVASOURCES := $(shell find $(SRCDIR) -name '*.java')
3  JAVSOURCES := $(shell find $(SRCDIR) -name '*.jav')
4
5  # Convert .java/.jav files to .class files with the same directory structure
6  JAVACLASSES := $(patsubst $(SRCDIR)/%.java,$(DESTDIR)/%.class,$(JAVASOURCES))
7  JAVCLASSES := $(patsubst $(SRCDIR)/%.jav,$(DESTDIR)/%.class,$(JAVSOURCES))
8
9  # Rule for compiling .jav files
10 $(DESTDIR)/%.class: $(SRCDIR)/%.jav
11 java -jar $(JTX) -d "$(dir $@)" -cp "$(SRCDIR):$(DESTDIR):target/dependencies/" $<
12 # Rule for compiling .java files
13 $(DESTDIR)/%.class: $(SRCDIR)/%.java
14 $(JC) -nowarn -d $(DESTDIR) -cp "$(SRCDIR):$(DESTDIR):target/dependencies/*" $(JFLAGS) $<
```

Aufbau der Umgebung III

Probleme:

1. javac compiliert und trackt Änderungen von Abhängigkeiten automatisch
2. javac ist sehr langsam wenn für jede Datei einzeln aufgerufen (mehrfache Compilierungen + JVM overhead)

```
javac src/main/java/de/dhbwstuttgart/typedeployment/TypeInsert.java
javac src/main/java/de/dhbwstuttgart/typedeployment/TypeInsertPlacer.java
...
javac src/main/java/Main.java
```

∼ 5min Compilerzeit

```
javac src/main/java/de/dhbwstuttgart/typedeployment/TypeInsert.java src/main/java/de/dhbwstuttgart/typedeployment/TypeInsertPlacer.java ... src/main/java/Main.java
```

∼ 2sec Compilerzeit

Aufbau der Umgebung IV - compile script

1. Lese alle Sourcecode Dateien (*.jav, *.java)
2. Filtere die Dateien, die neu compiliert werden müssen
3. Rufe den Compiler einmal mit allen Sourcefiles als Argumente auf

```
javac -d "$DESTDIR" -cp "$SRCDIR:$DESTDIR:target/dependencies/*" $JAVAC_FLAGS "${JAVA_CHANGED[@]}"
```

Primitive Typen in Java-TX

- Java erlaubt neben Referenztypen auch primitive Datentypen (int, boolean, ...)
- Java-TX erlaubt primitive Datentypen zwar im Quellcode, wandelt diese aber in die korrespondierende Wrapperklasse um (Integer, Boolean, ...)

```
int a = 10;
boolean b = true;
float c = 10.0f;
```

```
Integer var1 = null;
var1 = 10;
Boolean var2 = null;
var2 = true;
Float var3 = null;
var3 = 10.0F;
```

Überschreiben von Methoden II

- In Java: Methoden nicht anhand von Rückgabewert überschreibbar

```
1 public class Bar {
2   int foo(Object obj){return 0;}
3   boolean foo(Object obj){return false;}
4 }
```

```
1 Bar.java:3: Fehler: Methode foo(Object) ist bereits in Klasse Bar definiert
2     boolean foo(Object obj){return false;}
3             ^
```

- Aber: Generell auf JVM lauffähig

Überschreiben von Methoden II

- Überschreiben von Java Methoden mit primitiven Datentypen als Parameter funktionierte nicht

```
public boolean equals(Object obj);
```

```
public class Foo {
   equals(Object o){
        return false;
   }
}
```

```
public class Foo {
    public Foo() {}
    Boolean equals(Object var1) {
        return false;
    }
}
```

Überschreiben von Methoden III

- Lösung: Wenn Methodensignatur eines Supertyps sich nur in primitiven-/Wrapper-Datentypen unterscheidet, werden Typen vom Supertyp in Subtyp substituiert

```
1 public class Foo {
2    equals(Object o){
3         return false;
4    }
5 }
```

```
1 public class Foo {
2     public Foo() {}
3     boolean equals(Object var1) {
4         return false;
5     }
6 }
```

Kompatibilität mit funktionalen Interfaces I

- In Java haben Lambda Ausdrücke als Target Type ein funktionales Interface z.B. java.util.function.Function

```
Function<Integer, Integer> func = x -> x*2;
```

- Java-TX unterstützt echte Funktionstypen

```
var func = x -> x*2;
func: Fun1$$<Integer, Integer>
```

Kompatibilität mit funktionalen Interfaces II

- Java ist nominal typisierte Sprache
- Problem: Lambda Ausdrücke haben in Java-TX einen FunN$$ Typ (für N = #Parameter)
- Aber: Java Bibliotheken wie Stream erwarten verschiedene funktionale Interfaces
- Daher: Lambda Ausdrücke müssen je nach Kontext erwartetes funktionales Interface als Target Typ haben

```
1 public class Main {
2     main() {
3       List<Integer> list = new ArrayList<>(List.of(1,2,3,4,5));
4       return list.stream().map(x -> x*2).toList();
5     }
6 }
```

Kompatibilität mit funktionalen Interfaces III

```
<R> Stream<R> map(Function<? super T, ? extends R> var1);
```

```
public interface Function<T, R> {
   R apply(T var1);
}
```

- Eigentlich würde Java-TX Fun1$$<Integer, Integer> inferieren
- Aber: Stream.map erwartet Function<? super T, ? extends R>

```
...
3: invokedynamic #41,  0               // InvokeDynamic #0:apply:(LFoo;)
   LFun1$$Ljava$lang$Integer$_$Ljava$lang$Integer$_$;
//Invalid Bytecode
5: invokeinterface #45,  2             // InterfaceMethod java/util/stream/Stream.map:(Ljava/util/
   function/Function;)Ljava/util/stream/Stream;

...
```

Kompatibilität mit funktionalen Interfaces IV

```
...
50: invokedynamic #60,  0              // InvokeDynamic #0:apply:(LFoo;)Ljava/util/function/Function;
55: invokeinterface #66,  2            // InterfaceMethod java/util/stream/Stream.map:(Ljava/util/function/Function;)Ljava/util/stream/Stream;
...
```

Fazit

Vorteile:

- Programmierer muss weniger Typen explizit angeben
- Funktionstypen erlauben übersichtlichere Subtypisierung von anonymen Funktionen als Java

Nachteile:

- (Alle verwendeten/berückstichtigten Typen müssen manuell importiert werden)
  $\rightarrow$ Der Programmierer muss schon wissen, welche Typen in Frage kommen
- Es ist möglich, dass ein ungewünschter Typ inferiert wird
- Aktuell begrenzte Sprachfeatures & vermutlich einige Bugs

Fazit

Weg ist noch weit ...

Aktuell $\sim$ 3% der Java Dateien in Java-TX übersetzt