DHBW Stuttgart

Java-TX Roadmap 2025ff

Martin Plümicke

1. Pattern Matching
2. Generated Generics
3. Heterogenous Translation
4. Bound generics
5. Java-TX–compiler
6. Java-TX unification as a webservice
7. Real Functionstype in Java-TX
8. Modulsystem for Java-TX

Pattern Matching

```
sealed interface List<T> permits Cons, Empty {}

public record Cons<T>(T a, List<T> l) implements List<T> {}
public record Empty<T>() implements List<T> {}

public class PatternMatchingListAppend {

    public append(Cons(a, b), list2) {
        return new Cons<>(a, append(b, list2));
    }

    public append(Empty(), list2) {
        return list2;
    }
}
```

Result of type inference

```
class TPHsAndGenerics {

  id = x -> x;

  id2(x){
      return id.apply(x);}
  m(a, b){
      return b; }

  m2(a, b) {
      var c = m(a,b);
      return a; }
}
```

Result of type inference

```
class TPHsAndGenerics {

  id = x -> x;

  id2(x){
      return id.apply(x);}
  m(a, b){
      return b; }

  m2(a, b) {
      var c = m(a,b);
      return a; }
}
```

```
class TPHsAndGenerics {
  Fun1$$<UD, ETX>
      id = (DZP x) -> x;

  ETX id2(V x) {
       return id.apply(x);}
  AI m(AM a, AN b){
      return b; }

  AA m2(AB a, AD b){
      AE c = m(a,b);
      return a; }
}
```

Result of type inference

```
class TPHsAndGenerics {

  id = x -> x;

  id2(x){
      return id.apply(x);}
  m(a, b){
      return b; }

  m2(a, b) {
      var c = m(a,b);
      return a; }
}
```

```
class TPHsAndGenerics {
  Fun1$$<UD, ETX>
      id = (DZP x) -> x;

  ETX id2(V x) {
       return id.apply(x);}
  AI m(AM a, AN b){
      return b; }

  AA m2(AB a, AD b){
      AE c = m(a,b);
      return a; }
}
```

$\{ AB \lessdot AA, AD \lessdot AN, AN \lessdot AI, AI \lessdot AE, V \lessdot UD, AB \lessdot AM, DZP \lessdot ETX, UD \lessdot DZP \}$

Family of generated generics of the class and its methods

```
class TPHsAndGenerics
        <UD extends DZP, DZP extends ETX, ETX> {

    Fun1$$<UD, ETX> id = x -> x;

    <V extends UD> ETX id2(V x) {
        return id.apply(x);}

    <AM, AN extends AI, AI> AI m(AM a, AN b){
        return b;}

    <AA,AB extends AA, AD, AE> AA m2(AB a, AD b){
        AE c = m(a,b);
        return a;}
}
```

Complete the family of generated generics

```
class TPHsAndGenerics
        <UD extends DZP, DZP extends ETX, ETX> {

    Fun1$$<UD, ETX> id = x -> x;

    <V extends UD> ETX id2(V x) {
        return id.apply(x);}

    <AM, AN extends AI, AI> AI m(AM a, AN b){
        return b;}

    <AA,AB extends AA, AD extends AE, AE> AA m2(AB a, AD b){
        AE c = m(a,b);
        return a;}
}
```

Eliminating the cycles I

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
```

Eliminating the cycles I

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
...
  m(b1, b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

Eliminating the cycles II

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
...
  <L extends M, M extends L> void m(Box <L> b1, Box<M> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

Eliminating the cycles II

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
...
  <L extends M, M extends L> void m(Box <L> b1, Box<M> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

<L extends M, M extends L> is not correct!!!

Eliminating the cycles II

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
...
  <L extends M, M extends L> void m(Box <L> b1, Box<M> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
```

`<L extends M, M extends L>` is not correct!!!

Solution: Substitute all type variables of the cycle by a fresh type variable.

Subtyping Relation on wildcards

Semantics:

`? extends` θ: *There is a subtype of* θ.

`? super` θ': *There is a supertype of* θ'.

Subtyping Relation on wildcards

Semantics:

$\texttt{?}\ \texttt{extends}\ \theta$: *There is a subtype of* θ.

$\texttt{?}\ \texttt{super}\ \theta'$: *There is a supertype of* θ'.

For $\theta \leq^* \theta'$ holds

$$\theta \leq^* \texttt{?}\ \texttt{super}\ \theta',$$

$$\texttt{?}\ \texttt{extends}\ \theta \leq^* \theta', \text{ and}$$

$$\texttt{?}\ \texttt{extends}\ \theta \leq^* \texttt{?}\ \texttt{super}\ \theta'.$$

Subtyping Relation on wildcards

Semantics:

$\texttt{?}\ \texttt{extends}\ \theta$: *There is a subtype of* θ.

$\texttt{?}\ \texttt{super}\ \theta'$: *There is a supertype of* θ'.

For $\theta \leq^* \theta'$ holds

$$\theta \leq^* \texttt{?}\ \texttt{super}\ \theta',$$

$$\texttt{?}\ \texttt{extends}\ \theta \leq^* \theta', \text{ and}$$

$$\texttt{?}\ \texttt{extends}\ \theta \leq^* \texttt{?}\ \texttt{super}\ \theta'.$$

Especially: $\leq^*$ is not reflexiv on wildcards

Wildcard–Approach

```
class Box<T> { ... }
...
  <L> void m(Box <L> b1, Box<L> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
...
   Box<?> bb1 = ...;
   Box<?> bb2 = ...;
   m(bb1, bb2);
...
```

is not correct!

Wildcard–Approach

```
class Box<T> { ... }
...
  <L> void m(Box <L> b1, Box<L> b2) {
    b1.set(b2.get());
    b2.set(b1.get());
  }
...
   Box<?> bb1 = ...;
   Box<?> bb2 = ...;
   m(bb1, bb2);
...
```

is not correct!

Java-TX Signature: `<L extends L>`

have to be introduced.

One argument I

```
class Box<T> {
  T elem;

  T get() { return elem; }

  <T> void set(T x) { elem = x; }
}
...
  m(b) {
     b.set(b.get());
  }
```

One argument II

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

Java-TX Signature: `<L extends L>`

is not necessary!

Eliminating the infima

```
class Infimum {

    m(a, b, c) {
        b = a;
        c = a;
    }
}
```

Eliminating the infima

```
class Infimum {

    m(a, b, c) {
        b = a;
        c = a;
    }
}
```

```
class Infimum {
    <A extends B, A extends C>
    m(A a, B b, C c) {
        b = a;
        c = a;
    }
}
```

Eliminating the infima

```
class Infimum {

    m(a, b, c) {
        b = a;
        c = a;
    }
}
```

```
class Infimum {
    <A extends B, A extends C>
    m(A a, B b, C c) {
        b = a;
        c = a;
    }
}
```

Multiple inheritance is not allowed!

Algorithm

- Building the family of generated generics of the class and its methods
- Complete the family of generated generics
- Eliminating the cycles (To Do: Java-TX Signature)
- Eliminating the infima (To Do)

Function types

```
class OLFun {
  m(f) {
    var x;
    x = f.apply(x+x);
    return x;
  }
}
```

```
Double m(Fun1$$ <Double ,Double>);
    descriptor: (LFun1$$$_$Double$_$Double$_$;)Double;
Integer m(Fun1$$ <Integer ,Integer>);
    descriptor: (LFun1$$$_$Integer$_$Integer$_$;)Integer;
String m(Fun1$$ <String ,String >);
    descriptor: (LFun1$$$_$String$_$String$_$;)String;
```

Further generic types

```
class VectorAdd {
  m(v1, v2) {
    var ret = new Vector<>();
    while (i < v1.size()) {
      erg.addElement(v1.elementAt(i) + v2.elementAt(i));
      i++;
    }
  }}
```

```
Double m(Vector<Double>, Vector<Double>);
    descriptor:(LVector;LVector;)LVector;
Integer m(Vector<Integer>, Vector<Integer>);
    descriptor:(LVector;LVector;)LVector;
String m(Vector<String>, Vector<String>);
    descriptor:(LVector;LVector;)LVector;
```

Type Inference of Bound Generics I

```
class Pair {
  fst;
  snd;

  Pair(fst, snd) { this.fst=fst; this.snd=snd; }

  getfst() { return this.fst; }

  swap() {
    return new Pair<>(this.snd, this.fst);
  }
}
```

Type Inference of Bound Generics II

```
class Pair<X> {
  X fst;
  X snd;

  Pair(X fst, X snd) { this.fst=fst; this.snd=snd; }

  X getfst() { return this.fst; }

  Pair<X> swap() {
    return new Pair<>(this.snd, this.fst);
  }
}
```

Type Inference of Bound Generics III

```
class Pair {
  fst;
  snd;

  Pair(fst, snd) { this.fst=fst; this.snd=snd; }

  getfst() { return this.fst; }

  swap() {
    return new Pair<>(this.snd, this.fst);
  }

  m() { Number n = getfst(); }
}
```

Type Inference of Bound Generics IV

```
class Pair {
  Number fst;
  Number snd;

  Pair(Number fst, Number snd) {this.fst=fst; this.snd=snd;}

  Number getfst() { return this.fst; }

  Pair swap() {
    return new Pair<>(this.snd, this.fst);
  }

  void m() { Number n = (new Pair<>()).getfst(); }
}
```

Type Inference of Bound Generics V

```
class Pair<X extends Number> {
  X fst;
  X snd;

  Pair(X fst, X snd) { this.fst=fst; this.snd=snd; }

  X getfst() { return this.fst; }

  Pair<X> swap() {
    return new Pair<>(this.snd, this.fst);
  }
  void m() { Number n =  (new Pair<>()).getfst(); }
}
```

Implementation of the theory in *Global Type Inference for Featherweight Generic Java* (ECOOP 22)

Java-TX in Java-TX

- Complete the implementation of the Java-TX–compiler in Java-TX
- Remove type annotations in the Java code
- Infer new types and compare
- First large project in Java-TX
- maven for Java-TX

Java-TX unification as a webservice

- Java-TX unification is NP-hard
- The algorithm is scheduled in parallel
- Implementation on a massiv parallel machine
- Provide as a webservice

Real Functionstype in Java-TX

- Theory is done: Integration of function type and functional interfaces (strawman approach)
  [Reinhold 2009, Plümicke/Stadelmeier 2017]
- Complete implementation and publish

Modulsystem for Java-TX

A new challenge is to transfer ideas from the complex sml modul system to Java-TX.

Summary