DHBW Stuttgart

Completing the functional approach in object-oriented languages

Martin Plümicke

Overview

1. Development of OO programming languages
2. Global type inference
3. Function types
4. Additional features
   - Ad-hoc polymorphism
   - Generalized type variables
5. Pattern matching
6. Summary and Future work

Development of OO programming languages

Many features from functional programming languages are transfered to OO-languages (e.g. PIZZA, Scala, Java, C#, C++).

Main challenges

- States in imperative OO languages
- Subtyping
- Overloading and overriding

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Algebraic data types, pattern–matching: Record–Types and Pattern–Matching (`switch`- and `instanceof`-statement)

No patterns in method headers

Example Java

Parametric Polymorphism: Generics with use-side variance

Very complex feature with wildcards and capture-conversion

Type inference: *Local* type inference

No type inference for method declarations und recursive lamda expressions

Functions as first-class citizens: Lamda-expressions as implementation of functional interfaces

No function types

Algebraic data types, pattern–matching: Record–Types and Pattern–Matching (`switch`- and `instanceof`-statement)

No patterns in method headers

⇒ In Java-TX we address these gaps

Global type inference

First example

```
import java.lang.Integer;

public class Fac {

  getFac(n){
    var res = 1;
    var i = 1;
    while(i<=n) {
      res = res * i;
      i++;
    }
    return res;
  }}
```

Type inference

First example

```
import java.lang.Integer

public class Fac {

 java.lang.Integer getFac(java.lang.Integer n){
    var res = 1;
    var i = 1;
    while(i<=n) {
      res = res * i;
      i++;
    }
    return res;
 }}
```

```
class Matrix extends Vector<Vector<Integer>> {
  mul(m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
  mul(m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Which is the correct type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Matrix m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Matrix m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Is this the *best* type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<Vector<Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<Vector<Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Which is the *most general* type?

```
class Matrix extends Vector<Vector<Integer>> {
 Matrix mul(Vector<? extends Vector<? extends Integer>> m) {
    var ret = new Matrix();
    var i = 0;
    while(i < size()) {
        var v1 = this.elementAt(i);
        var v2 = new Vector<Integer>();
        var j = 0;
        while(j < v1.size()) {
            var erg = 0;
            var k = 0;
            while(k < v1.size()) {
              erg = erg + v1.elementAt(k) * m.elementAt(k).elementAt(j);
              k++; }
            v2.addElement(new Integer(erg));
            j++; }
        ret.addElement(v2);
        i++; }
    return ret; } }
```

Principal type

Principal type for a functional program:

A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it. [Damas, Milner 1982]

Principal type

Principal type for a functional program:

A type-scheme for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a generic instance of it. [Damas, Milner 1982]

Principal type for Java-TX:

An intersection type-scheme with minimal number of elements for a declaration is a principal type-scheme, if any other type-scheme for the declaration is a subtype of generic instance of one element of the intersection type-scheme.

Example Principal Type

Intersection Type of `mul`

```
mul : Matrix → Matrix
    ∧ Vector<Vector<Integer>> → Matrix
    ∧ Vector<Vector<Integer>> → Vector<Vector<Integer>>
    ∧ Vector<Vector<? extends Integer>> → Matrix
    ∧ Vector<Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<Integer>> → Matrix
    ∧ Vector<? extends Vector<Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<? extends Integer>> → Matrix
    ∧ Vector<? extends Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
```

Example Principal Type

Principal Type of `mul`

```
mul : Matrix → Matrix
    ∧ Vector<Vector<Integer>> → Matrix
    ∧ Vector<Vector<Integer>> → Vector<Vector<Integer>>
    ∧ Vector<Vector<? extends Integer>> → Matrix
    ∧ Vector<Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<Integer>> → Matrix
    ∧ Vector<? extends Vector<Integer>>
                                → Vector<Vector<Integer>>
    ∧ Vector<? extends Vector<? extends Integer>> → Matrix
    ∧ Vector<? extends Vector<? extends Integer>>
                                → Vector<Vector<Integer>>
```

Function types

- In Java 8 lambda expressions has fuctional interfaces as target types.
- There are no real function types
- Java-TX has function types as in Scala
- In Java-TX Function types and functional interfaces are integrated

Drawbacks of missing function types

- Missing function types $\Rightarrow$ Introducing `Bi/Function`–Interfaces
- Subtyping problem $\Rightarrow$ Using wildcards
- Impossibility of direct application of lambda expressions
  $\Rightarrow$ Using type-casts

All problems are solvable, but not pretty!!!

$\Rightarrow$ **Introducing real function types with integration into the concept of target types.**

Introduction of Fun*N*$$

```
interface Fun N$$<-T1, ..., -TN,+R> {
    R apply(T1 arg1, ..., TN argN);
}
```

where

- Fun*N*$$<$T'_1, \ldots, T'_N, T_0$> $\leq^*$ Fun*N*$$<$T_1, \ldots, T_N, T'_0$> iff $T_i \leq^* T'_i$
- In Fun*N*$$ no wildcards are allowed.

Introduction of Fun*N*$$

```
interface Fun N$$<-T1, ..., -TN,+R> {
    R apply(T1 arg1, ..., TN argN);
}
```

where

- Fun*N*$$<$T'_1,\ldots,T'_N,T_0$> $\leq^*$ Fun*N*$$<$T_1,\ldots,T_N,T'_0$> iff $T_i \leq^* T'_i$
- In Fun*N*$$ no wildcards are allowed.

Lambda–expressions are explicitly typed by Fun*N*$$–types

Example function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                                      ? extends C>>>


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

Example function types

```
//A -> (B -> (((A, B) -> C) -> C)))
Func.<? super A,
      ? extends Func.<? super B,
                      ? extends Func.<? super BiFunc.<? super A,
                                                      ? super B,
                                                      ? ext. C>>,
                                                      ? extends C»>

Fun1$$<A, Fun1$$<B, Fun2$$<A, B, C>, C>>
g = x -> y -> f -> f.apply(x,y);
```

Ad-hoc polymorphism

Example

```
class OL {
  m(x) { return x + x; }

  m(x) { return x || x; }
}

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

Type of `m`, `m`, and `main`

$$\texttt{m} : \texttt{Integer} \to \texttt{Integer} \\ \wedge\ \texttt{Double} \to \texttt{Double} \\ \wedge\ \texttt{String} \to \texttt{String}$$

$$\texttt{m} : \texttt{Boolean} \to \texttt{Boolean}$$

$$\texttt{main} : \texttt{Integer} \to \texttt{Integer} \\ \wedge\ \texttt{Double} \to \texttt{Double} \\ \wedge\ \texttt{String} \to \texttt{String} \\ \wedge\ \texttt{Boolean} \to \texttt{Boolean}$$

Generalized type variables

```
class Id {
  id(x) { return x; }
}
```

What is the type of `id`?

Generalized type variables

```
class Id {
  id(x) { return x; }
}
```

What is the type of `id`?

Result of the type inference algorithm:

```
class Id { K id(L b)({ return (b)::L; }):: M}
```

$(\{(\texttt{L} < \texttt{M}), (\texttt{M} < \texttt{K})\}, \emptyset)$

Generalized type variables

```
class Id {
  id(x) { return x; }
}
```

What is the type of `id`?

Result of the type inference algorithm:

```
class Id { K id(L b)({ return (b)::L; }):: M}
```

$$(\{(\texttt{L} < \texttt{M}), (\texttt{M} < \texttt{K})\}, \emptyset)$$

Generated class:

```
class Id {
  <L extends K, K> K id (L x) { return x; }
}
```

Second Example

```
class Assign {
  assign (x, y) { y = x; }
}
```

Second Example

```
class Assign {
  assign (x, y) { y = x; }
}
```

Result of the type inference algorithm:

```
class Assign {
  K assign(L x, M y)({ (y)::M = (x)::L; }):: N
}
```

$$(\{(\mathtt{L} < \mathtt{M})\}, \{\{(\mathtt{K} \mapsto \mathtt{void})\}\})$$

Second Example

```
class Assign {
  assign (x, y) { y = x; }
}
```

Result of the type inference algorithm:

```
class Assign {
  K assign(L x, M y)({ (y)::M = (x)::L; }):: N
}
```

$$(\{(\mathtt{L} < \mathtt{M})\}, \{\{(\mathtt{K} \mapsto \mathtt{void})\}\})$$

Generated class:

```
class Assign {
  <L extends M, M> void assign (L x, M  y) { y = x; }
}
```

Pattern matching I

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(A a, Cons(A b, List l))
                    -> System.out.println(a);
    case Cons(A a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching II

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(var a, Cons(var b, var l))
                    -> System.out.println(a);
    case Cons(var a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching in Java-TX I

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(List<A> x) {
  switch(x) {
    case Cons(a, Cons(b, l)) -> System.out.println(a);
    case Cons(a, Nil()) -> System.out.println(a);
    case Nil() -> System.out.println("Nil");
  }
}
```

Pattern matching in Java-TX II

```
sealed interface List<A> permits Cons, Nil {}

record Cons<A>(A a, List<A> l) implements List<A> {}

record Nil<A>()implements List<A>  {}

void m(Cons(a, Cons(b, l))) { System.out.println(a); }
void m(Cons(a, Nil())) { System.out.println(a); }
void m(Nil()) { System.out.println("Nil"); }
```

Summary and Future work

Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Inferred generics

Summary and Future work

Summary

Introducing into Java:

- Global type inference
- Scala function types
- Ad-hoc polymorphism
- Inferred generics

Future work:

- Implementing pattern matching
- Avoiding capture-conversion

Studienarbeit–Ausschreibungen I

Portierung auf die Java–Erweiterung Muli

Zur Realisierung von Backtracking eignen sich besonders logische Programmiersprachen mit freien Variablen. Eine an der Universität Münster entwickelte Java–Erweiterung Muli ergänzt Java um Komponenten von logischen Programmiersprachen.
Ziel der Arbeit ist es die Backtracking–Schritte in der bisherigen Implementierung durch logische Konstrukte in Muli zu ersetzen, die das Backtracking automatisieren und damit ggf. erheblich effizienter machen.

Studienarbeit–Ausschreibungen II

Java–Bytecode Manipulation

Java-TX erlaubt es von anderen JVM-Sprachen erzeugte class-Files zu lesen und zu benutzen. Da die Verarbeitung etwas anders funktioniert müssen die class-Files nachbearbeitet werden.
In der Arbeit soll zunächst ein geeigntes Tool dafür ausgewählt werden. Dann sollen mit diesem Tool die Java-TX-Signaturen in die class-Files eingefügt werden.

Studienarbeit–Ausschreibungen III

Java-TX-Compiler in Java-TX

Im Compilerbau ist ein wichtiger Benchmark, dass ein Compiler in der eigenen Sprache implementiert ist. Dies soll im Rahmen dieser Studienarbeit für Java-TX erfolgen.
Für die Studienarbeit steht der Java-Code des Java-TX-Compilers und die Ergebnisse einer Vorgängerstudienarbeit zur Verfügung. Im Wesentlichen müssen aus dem Code alle Typinformationen entfernt werden und mit dem Java-TX-Compilers compiliert werden.
Sukzessive werden während der Implementierung Fehler gefunden werden. Diese sollten dokumentiert und wenn möglich auch behoben werden.
Darüber hinaus soll *Maven* für Java-TX nutzbar gemacht werden.

Studienarbeit–Ausschreibungen IV

invokedynamic

Der in JDK–7 eingeführte Bytecodebefehl soll in der Arbeit analysiert werden. Dazu soll ein Überblick über den Einsatz in JVM-basierten Sprachen und dem Standard-Java-Compiler gegeben werden. Schließlich soll das Java-TX–Feature *Pattern-Matching in Function Headern* mit Hilfe von `invokedynamic` implememtiert werden.