

Contents

1	Introduction	3
2	Generic Java	5
2.1	Introduction	5
2.1.1	From PIZZA via GJ to Java 5.0	5
2.1.2	Generics	5
2.1.3	F-bounded parameters	7
2.1.4	Boxed and unboxed types	8
2.1.5	Type Inference in Java 5.0	9
2.2	Generics and type inference as features of Java 5.0	9
2.2.1	Benefits of generics in Java 5.0	10
2.2.2	Convenience by using type inference	14
3	Java 5.0 Types	17
3.1	Introduction	17
3.2	Inheritance Hierarchy	17
4	Type Unification	31
4.1	Overview	31
4.2	Type unification algorithm	33
4.2.1	Type Unification without Wildcards	34
4.2.2	Type Unification with Wildcards	47
4.2.3	Unified Minimal Upper Bound	63
4.3	Implementation	64
5	Type Inference	67
5.1	Introduction to Type Inference	67
5.1.1	Cecil	71
5.2	Type inference in object oriented languages	71
5.2.1	Flow analysis approaches and constraint solving	72
5.2.2	Converting raw types to parametrized types	74
5.2.3	Milner-like approaches	75
5.2.4	ImplicitPoly-Tiger	77
5.3	Non-explicitly typed Generic Java Programs	78

5.4	The Type System of Java 5.0	78
5.4.1	Type inference rule for one class	81
5.4.2	Type inference example	92
5.4.3	Principal type property	100
5.5	Type Reconstruction	103
5.5.1	Type Reconstruction Algorithm	103
5.5.2	ANDERER SECTION NAME Type reconstruction example	134
5.6	Implementation	143
5.7	Comparison of our approach to other existing approaches	143
5.7.1	Flow analysis approaches	143
5.7.2	Converting raw types to parametrized types	144
5.7.3	Cecil	145
5.7.4	Milner-like approaches	145
5.7.5	OCAML	148
5.7.6	ImplicitPoly-Tiger	151
A	Abstract syntax	153
B	Type reconstruction example	155
B.1	Type reconstruction of the middle while-loop	160
C	Type reconstruction example factorial	171

Chapter 1

Introduction

Chapter 2

Generic Java

2.1 Introduction

In this introduction we will describe the new features of Java 1.5 [BCK⁺01, Aus04]. In the following we call Java's version 1.5 **Java 5.0**. First we give a short historical outline, then we consider explicitly the generics in **Java 5.0**. In a third part we introduce **??F-bounded???** class parameters [CCH⁺89, CHC90]. Finally, we consider boxed and unboxed types.

2.1.1 From PIZZA via GJ to Java 5.0

[OW97] [ORW00] [BOSW98b] [BOSW98a] [BOSW98c] [BCK⁺01]

2.1.2 Generics

The essential extensions of **Java 5.0** are the generic types (generics). This means classes can have parameters. The type system is then extended by parameterized types, type variables, and type terms. An earlier implementation of these features have been done by the **Generic-Java** compiler [BOSW98b], [BOSW98a].

Examples for parameterized types are

```
Seq<a>
Vector<a>
Pair<a,b>
```

The parameters in the classes can be instantiated by each type. For example a declaration of a vector of **Integers** is done by

```
Vector<Integer> v = new Vector<Integer>();
```

In comparison to the declaration `Vector v = new Vector();` in older java versions now the vector can only contain instances of the class **Integer**. In most cases datatypes like **Vector** are used only for instances of one class. Nevertheless, if the vector is needed for instances of different classes the vector instances can be declared as **Vector<Object>**.

The main advantage of declarations like this is that many type casts are unnecessary. For example, if we want to get the sixth object of an **Integer** vector **v**, in traditional **java** (Version ≤ 1.4) we must write

```
Integer i = (Integer)v.get(6);
```

whereas in **Java 5.0** satisfies

```
Integer i = v.get(6);
```

In parameterized classes the parameters substitute the previous type **Object**, as shown in the following cutout of the class **Vector**:

```
class Vector<a> {
    void add (a elem) { ...}

    a get (int i) { ...}
}
```

This means that the declaration `Vector<Integer> v = new Vector<Integer>();` instantiated the type variable **a** by **Integer**. Therefore the type cast shown above is unnecessary. Furthermore some run time errors become compiling errors. In traditional **java** statements like

```
Vector v = new Vector();
v.add(new Integer(9));
((String)v.get(5)) + "hallo"
```

are type correct, as the result type of **get** is **Object** and the type cast from **Object** to **String** is correct. It leads to an runtime error. In **Java 5.0** the compilation of the analogous lines

```
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(9));
((String)v.get(5)) ++ "hallo"
```

causes a type error, as a type cast from **Integer**, the instantiated result type of **get**, to **String** is incorrect.

Now, we give an example for a class with instantiated parameters.

```
class Matrix extends Vector<Vector<Integer>> {

    Matrix mul(Matrix m) {
        Matrix ret = new Matrix ();
        for(int i = 0; i < size(); i++) {
            Vector<Integer> v1 = this.elementAt(i);
```

```

        Vector<Integer> v2 = new Vector<Integer> ();
        for (int j = 0; j < v1.size(); j++) {
            int erg = 0;
            for (int k = 0; k < v1.size(); k++) {
                erg = erg + v1.elementAt(k).intValue()
                    * (m.elementAt(k)).elementAt(j).intValue();
            }
            v2.addElement(new Integer(erg));
        }
        ret.addElement(v2);
    }
    return ret;
}

```

The class `Matrix` is an extension of `Vector<Vector<Integer>>`, where the method `mul` is added.

2.1.3 ???F-bounded??? parameters

Furthermore in [BCK⁺01] ???F-bounded??? parameters are introduced, which means that parameters are constrained by interfaces. in the next example, which corresponds to example 2 from [BCK⁺01],
BEISPIEL NOCHMALs TESTEN

```

class ReprChange<a implements ConvertibleTo<b>,
                b implements ConvertibleTo<a>> {
    a element;

    void set(b x) { element = x.convert(); }
    b get() { return element.convert(); }
}

```

with the interface

```

interface ConvertibleTo<a> {
    a convert();
}

```

the type variable `a` can only be instantiated by types which corresponding objects implement the interface `ConvertibleTo`.

The following classes `INT` and `FLOAT` is an example for the application of `ReprChange`.

```

class INT implements ConvertibleTo<FLOAT> {

    int i;

    INT(int i) {

```

```

        this.i = i;
    }

    public FLOAT convert() {
        return new FLOAT((new Float((float)i)).floatValue());
    }
}

class FLOAT implements ConvertibleTo<INT> {

    float i;

    FLOAT(float i) {
        this.i = i;
    }

    public INT convert() {
        return new INT((new Integer((int)i)).intValue());
    }
}

```

In the actual beta-version of Java 5.0 ???F-bounded??? parameters are not implemented.

2.1.4 Boxed and unboxed types

In Java 5.0 there is a further extension. The base types like `boolean`, `integer`, or `char` and the respective corresponding classes can be used equivalently. This means that if a method has the result type `int`, another method with parameter type `Integer` is applicable to the result.

The following small examples present the possibilities of the feature:

Transformation base type in a objekte: `int` to `Integer`

so far:

```

int i = 16;
Integer I = new Integer(i)
I.equals(new Integer(17));

```

new: without explicit transformation

```

int i = 16;
Integer I = new Integer(i)
I.equals(17);

```

Transformation Objekt to base type: `Integer` to `int`

so far:

```
Integer I = new Integer(22);
int erg = I.intValue() + 21;
```

new: without explicit transformation

```
Integer I = new Integer(22);
int erg = I + 21;
```

Unfortunately, there is one restriction. It is not possible to apply a method on base constant as `1.toString()`.

2.1.5 Type Inference in Java 5.0

In Java 5.0 type inference is implemented for type instantiations of formal type parameters (type variables) in polymorphic types. Odersky describes the implementation in [Ode02]. The instantiations use only local knowledge and is a variant of the local type inference [PT00] (cp. section 5.1).

As an example we consider again the class `Vector<a>`:

```
class Vector<a> {
    void add (a elem) { ...}

    a get (int i) { ...}
}
```

An application of the method `add` to an `Integer`

```
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(1));
```

infers the type `Integer` for the type variable `a`. In Java 5.0 it is impossible to declare this instantiation explicitly, such as `v.add:(Integer -> void) (new Integer(1))`.

2.2 Generics and type inference as features of Java 5.0

In this section we will discuss the introduction of generics in Java. From this following we will show, why it make sense to extends the generics by a type inference system.

2.2.1 Benefits of generics in Java 5.0

Now we compare Java's type system without parametrized classes respectively types with parametrized types in Java 5.0. We will present two main advantages of the generics. The first one is that less type casts are necessary, which reduces errors, which cannot be detected during compile time. The second one is the necessity to use much more detailed type annotations. As type annotations can be viewed as an abstract interpretation of the program's semantic, more details type systems detect more semantical error during program construction time [ZITAT [BOSW98c] ???].

Type casts

Let us consider again the Java 5.0 declaration

```
Vector<Integer> v = new Vector<Integer>();
```

The main advantage of declarations like this are that already during compile time errors are detected, if a method with an instantiated parameter type is applied to an object of a wrong type, which would cause no error if the parameter type would be `Object`. For example

```
v.add(new Integer(27));
((String)v.get(0)) ++ "Hallo";
```

would cause an error during compilation time. The result type of `v.get(0)` is `Integer` and there is no possibility to cast an `Integer` to a `String`. There a type error provoked, although the operator `++` connecting two `Strings` is correct.

If we would implement the same program in a lower Java version it would look like

```
Vector v = new Vector();
v.add(new Integer(27));
((String)v.get(0)) ++ "Hallo";
```

Then, `v.get(0)` has the type `Object`. But this means that the type cast from `Object` to `String` is possible. Therefore the type cast would not provoke a type error. As the operator `++` is correct for two `Strings` no error during compilation time would arise.

This example shows that type casts often disguise real errors, as only for technical reasons they are inserted, although a type change is not needed really. This means that type casts should be avoided as often as possible. This is supported by the generics in Java as shown above.

Generics considered as more detailed type annotations

Now, we will consider the possibilities and the necessities as more detailed type annotations for fields, variables, and methods. As type annotations of programs can be considered as abstract interpretations of the programs, more detailed type annotations means that the semantics of the program is also more detailed approached by the types. But

this means then, that programs written in languages with more detailed type systems, are rather semantically correct if they are syntactically correct. An *old SML* [Mil97] programmer said *"If my programs have the right types, they are nearly already work also correct"*. In section ?? we will compare the Hindley/Milner type system [DM82], which is the base of the SML type system and is also a detailed type system, with the Java 5.0 type system.

We give a further example which should confirm the thesis. The example implements a cut out of a statistics of a town. The correlation between children and adults in some districts are considered. We give this implementation again in the lower Java versions and in Java 5.0.

For the presentation of the correlation we need a class `Pair`. In traditional Java `Pair` is implemented as

```
class Pair {
    Object x;
    Object y;

    Pair(Object x, Object y) {
        this.x = x;
        this.y = y;
    }

    Object fst() {
        return x;
    }

    Object snd() {
        return y;
    }
}
```

The next class we consider the population development over nine years. The field `year_district_children_adults` describes the correlation between children and adults in different districts in different years.

```
class population_development {
    Vector year_district_children_adults;

    population_development() {
        year_district_children_adults = new Vector();
        for (int i = 0; i < 9; i++) { // year 0 to year 9
            Vector district_children_adults = new Vector();
            for (int j = 0; j < 4; j++) { // district 0 bis district 4
                long children = select from Database
                long adults = select from Database
                Pair children_adults
                    = new Pair(new Long(children), new Long(adults));
                district_children_adults.addElement(children_adults);
            }
        }
    }
}
```

```
        year_district_children_adults.addElement(district_children_adults);
    }
}

public static void main(String[] args) {
    population_development be = new population_development();
    for (int i = 0; i < 9; i++) { // year 0 to year 9
        for (int j = 0; j < 4; j++) { // district 0 bis district 4
            System.out.print("year " + i + ": district " + j + ": ");
            System.out.print("children: " +
                ((Pair)((Vector)be.year_district_children_adults.get(i)).get(j)).fst() + " ");
            System.out.println("adults: " +
                ((Pair)((Vector)be.year_district_children_adults.get(i)).get(j)).snd());
        }
    }
}
```

The imprecise type declarations caused by the loss of generics in traditional Java are italic printed. The variables `year_district_children_adults` and `district_children_adults` both are typed by `Vector`. But it is not determined the type of the vector's elements. In contrast in Java 5.0 these variables are typed more precise.

```
class Pair<a,b> {
    a x;
    b y;

    Pair(a x, b y) {
        this.x = x;
        this.y = y;
    }

    a fst() {
        return x;
    }

    b snd() {
        return y;
    }
}

class population_development {
    Vector<Vector<Pair<Long,Long>>> year_district_children_adults;

    population_development() {

```

```

    year_district_children_adults = new Vector<Vector<Pair<Long,Long>>>();
    for (int i = 0; i < 9; i++) { // year 0 to year 9
        Vector<Pair<Long,Long>> district_children_adults
            = new Vector<Pair<Long,Long>>();
        for (int j = 0; j < 4; j++) { //district 0 to district 4
            long children = select from Database
            long adults = select from Database
            Pair<Long,Long> children_adults
                = new Pair<Long,Long>(children, adults);
            district_children_adults.addElement(children_adults);
        }
        year_district_children_adults.addElement(district_children_adults);
    }
}

public static void main(String[] args) {
    population_development be = new population_development();
    for (int i = 0; i < 9; i++) { // year 0 to year 9
        for (int j = 0; j < 4; j++) { //district 0 to district 4
            System.out.print("year " + i + ": district " + j + ": ");
            System.out.print("children: " +
                be.year_district_children_adults.get(i).get(j).fst() + " ");
            System.out.println("adults: " +
                be.year_district_children_adults.get(i).get(j).snd());
        }
    }
}
}

```

The field `year_district_children_adults` has the type `Vector<Vector<Pair<Long,Long>>>` and the local variable `district_children_adults` has the type `Vector<Pair<Long,Long>>`. In this example substantial reflections have to be done to declare the type of the field `year_district_children_adults`. In contrast in traditional Java 5.0 it is obvious that the type is `Vector`. A possible semantical defect could be caused in the fact that the implementer ignores during implementing that the town is divided in districts. This would lead to the leave out of the inner loops and a declarations:

```

long children = select from Database
long adults = select from Database
Pair<Long,Long> children_adults = new Pair<Long,Long>(children, adults);
year_district_children_adults.addElement(children_adults);

```

In Java 5.0 the statement `year_district_children_adults.addElement(children_adults);` would bring the type error

```

population_development.java:30:
addElement(java.util.Vector<Pair<java.lang.Long, java.lang.Long>>)

```

```

in java.util.Vector<java.util.Vector<Pair<java.lang.Long, java.lang.Long>>>
cannot be applied to (Pair<java.lang.Long, java.lang.Long>)
    year_district_children_adults.addElement(children_adults);
                                ^

```

1 error

In contrast in traditional Java no error would be brought. This is a semantical error which is transformed to a syntactical error by a type declaration using generics.

If we estimate this error, we recognize that the real cause of the not recognition of the error by the compiler is, that the field `year_district_children_adults` is not precise type declared. Types of fields are determined during the design phase of the software development process. This means on the one hand that the given scenario is absolutely realistic, as the type declaration and the coding is done in different points in time. On the other hand it shows that error avoiding is possible by doing more reflections in designing more detailed types.

2.2.2 Convenience by using type inference

In last two section, we show why it makes sense to use generics in Java. On the one hand type casts can be avoided, which is comfortable and allows detecting errors in a earlier phase of the software development process. On the other hand generics can avoid some semantical errors as typing is a kind of abstract interpretation of the program's semantics by transforming it to a type error.

The main problems to argue the programmers to use generics are that they must learn the theoretical base of the type system and that it is unconvience to write these syntactically long type terms. The first problem is indeed only solvable by learning the ideas of the type system. But the second problem can be solved by a type inference system. Type inference systems are well-known from functional programming languages (e.g. [DM82, Mil97, eAB⁺02]) (NEUESTE VERSION HASKELE-REPORT). The idea in Java 5.0 is that the fields are typed by the programmer and that the types of the methods and local variables are determined by the system. If we consider the two example from the above paragraphs again, they would look like the following.

In the matrix example the variables are printed italic which types must be determined.

```

class Matrix extends Vector<Vector<Integer>> {

    mul(m) {
        ret = new Matrix ();
        i = 0;
        while(i < size()) {
            v1 = this.elementAt(i);
            v2 = new Vector<Integer> ();
            j = 0;
            while (j < v1.size()) {
                erg = 0;

```

```

        k = 0;
        while (k < v1.size()) {
            erg = erg + v1.elementAt(k)
                * (m.elementAt(k)).elementAt(j);
            k++; }
        v2.addElement(erg);
        j++; }
    ret.addElement(v2);
    i++; }
    return ret; }

```

Additionally, the return type of `mul` must be determined. Here, no field is declared. In contrast in the town statistics there is a field. The field is explicitly typed.

```

class population_development {
    Vector<Vector<Pair<Long,Long>>> year_district_children_adults;

    population_development() {
        year_district_children_adults = new Vector<Vector<Pair<Long,Long>>>();
        for (i = 0; i < 9; i++) { // year 0 to year 9
            district_children_adults = new Vector<Pair<Long,Long>>();
            for (j = 0; j < 4; j++) { // district 0 to district 4
                children = select from Database
                adults = select from Database
                children_adults = new Pair<Long,Long>(children, adults);
                district_children_adults.addElement(children_adults);
            }
            year_district_children_adults.addElement(district_children_adults);
        }
    }
}

```

The types of the local variables (italic printed) and the return type of the method should be determined by the system.

As shown by these examples it is convenient for the programmer to omit the type declarations. Coeval the static detailed type system and its benefits is not lost. Rather type determination can be considered as a check if the types of a method defined during the analysis and the design phase agree with the by the system determined types of the coded method.

Chapter 3

Java 5.0 Types

3.1 Introduction

In this chapter we describe the type system of Java 5.0. The base of the type system is given in [GJSB05]. In Java 5.0 types are given as type terms, like `Vector<Vector<Integer>>` or `? extends List<Object>`. The type terms are constructed over the class/interface names. The given class/interface names form a finite rank alphabet, where the number of parameters is mapped to the corresponding class/interface names. The parameters can be bounded by further type terms. This restrict the type term construction, such that as arguments only types are allowed, which fulfils the bounds. Furthermore the inheritance relation forms a subtyping ordering. This means that the inheritance hierarchy can be considered as polymorphic order-sorted types. Polymorphic order-sorted types are a part of the theory of polymorphic order-sorted algebras.

The theory of polymorphic order-sorted algebras is a generalization of the concepts of order-sorted algebras (e.g. [GM89]) and of polymorphic many-sorted algebras [Thi94]. Polymorphic order-sorted algebras describe in [Smo89, Smo88a] the semantics of a typed logical programming language, while in [Thi93] a first approach to describe a functional programming language by polymorphic order-sorted algebras is given. In [Plü99a] we extended the approach of [Thi93].

From consideration the Java 5.0 types as polymorphic order-sorted types follows that the methods typed by the type terms could be considered as polymorphic order-sorted signature and the semantics of Java 5.0 class could be considered as a polymorphic order-sorted algebra. We will not do this. We will only consider the Java 5.0 types as polymorphic order-sorted types, as this is the base for the type unification (chapter 4) and the type inference (chapter 5).

3.2 Polymorphic Order-sorted Types as Inheritance Hierarchy

In this section, we define a set of types, an ordering on this set of types, and the translation from the inheritance hierarchy of Java 5.0 classes to the type ordering.

The base of the types are elements of the set of terms $T_\Theta(TV)$, which are given as a set of terms over a finite rank alphabet Θ of class names and a set of type variables TV . Because of this we will speak of *type terms* instead of types.

Example 3.1 *Let the following Java 5.0 program be given:*

```
class A<a> implements I<a> { ...}

class B<a> extends A<a> { ...}

class C<a extends I<b>,b> { ...}

interface I<a> { ...}
```

The rank alphabet $\Theta = \Theta_{n \in \mathbb{N}}^{(n)}$ is determined by

$$\Theta^{(1)} = \{A, B, I\} \text{ and } \Theta^{(2)} = \{C\}.$$

For example `A<Integer>`, `A<B<Boolean>>`, and `C<A<Object>, Object>` are type terms.

The problem is that the bound of the parameter `b` in the class `C` is not considered. This leads to the problem that type terms like `C<C<a>>` are allowed, although they are not correct in Java 5.0.

The solution of the problem is that we extend the rank alphabet Θ to a type signature, where the arity of the type constructors is indexed by bounded type variables. This leads to a restriction in the type term construction, such that the correct set of type terms is a subset of $T_\Theta(TV)$. Additionally the set of correct type terms is added by some wildcard constructions. We call the set of correct types *set of simple types* $S\text{Type}_{TS}(BTV)$ (definition 3.8). Unfortunately, the definitions of the type signature and the simple types are mutually dependent. This is caused by the fact, that on the one hand the set of simple types is the term set over the type signature. On the other hand the types of the type signature are also the simple types. This means that we must first assume a given set of simple types, without knowing, how the set of simple types is exactly defined. As an approximation the reader must first assume the set of simple types as the term set over the class names.

Definition 3.2 (Bounded type variables) *Let $S\text{Type}_{TS}(BTV)$ be a set of simple types. Then, the set of bounded type variables is a family $BTV = (BTV^{(ty)})_{ty \in I(S\text{Type}_{TS}(BTV))}$, where each type variable is assigned to an intersection of simple types. $I(S\text{Type}_{TS}(BTV))$ denotes the set of intersections over simple types (cp. def. 3.8).*

In the following we will write a type variable a bounded by the type ty as $a|_{ty}$.

Remark Type variables which are not bounded can be considered as bounded type variables by themselves. For example a not bounded type variable a can be considered as $a|_a$.

BOUNDED TYPE
VARIABLE
UEBERALL MIT
& SCHREIBEN

Definition 3.3 (Bounded type variables in a declared Java 5.0 class) *Let the following Java 5.0 class be given:*

```
class C<A1 extends A1_bnd, ..., An extends An_bnd> {
    ...
    <T1 extends T1_bnd_1 & ... & T1_bnd_n1
    , ...,
    Tm extends Tm_bnd_1 & ... & Tm_bnd_nm > rty methodname ( ...) { ...}
    ...
}
```

The set of bounded type variables BTV of a method $methodname$ is given by set smallest set with the properties $A_i \in BTV^{(A_i.bnd)}$ and $T_i \in BTV^{(T1.bnd_i)}$ for $1 \leq j \leq bnd.ni$.

Example 3.4 *Let the following Java 5.0 class be given.*

```
class BoundedTypeVars<A extends Number> {
    ...
    <T extends Vector & J<A> & I,
    R extends Number> void m ( ...) { ...}
}
```

The set of bounded type variable BTV of the method m is given as $BTV^{(Number)} = \{A, R\}$ and $BTV^{(Vector \& J<A> \& I)} = \{T\}$.

Definition 3.5 (Type signature, type constructor) *Let $S_{Type_{TS}}(BTV)$ be a set of simple types. A type signature TS is a pair $(S_{Type_{TS}}(BTV), TC)$ where BTV is a family of bounded type variables and TC is a $(BTV)^*$ -indexed family of type constructors (class names).*

Definition 3.6 (Type signature of declared Java 5.0 classes) *Let JC be a set of declared Java 5.0 classes. The type signature $TS = (S_{Type_{TS}}(BTV), TC)$ of JC is given as: For each class declaration*

$$\text{class } C\langle expr_1, \dots, expr_n \rangle \dots \in JC$$

respectively for each interface declaration

$$\text{interface } C\langle expr_1, \dots, expr_n \rangle \dots \in JC$$

holds

$$C \in TC^{(tv_1, \dots, tv_n)}$$

where for $1 \leq i \leq n$ holds:

$$tv_i = \begin{cases} a_i|_{a_i} & \text{if } expr_i = "a_i" \\ a_i|_{\overline{ty_i}} & \text{if } expr_i = "a_i \text{ extends } \overline{ty_i}" \end{cases}$$

Example 3.7 *Let the following from example 3.1 extended Java 5.0 program be given:*

```
class A<a> implements I<a> { ... }

class B<a> extends A<a> { ... }

class C<a extends I<b>, b> { ... }

interface I<a> { ... }

interface J<a> { ... }

class D<a extends B<a> & J<b>, b> { ... }
```

Then, the corresponding type signature $(S_{Type_{TS}}(BTV), TC)$ is given by $TC^{(a|_a)} = \{A, B, I, J\}$, $TC^{(a|_{I \ } b|_b)} = \{C\}$, and $TC^{(a|_{B \<a> \& J \ } b|_b)} = \{D\}$.

Now, we have given all premises, such we can define the set of Java 5.0 simple types.

Definition 3.8 (Simple types) *The set of simple types $S_{Type_{TS}}(BTV)$ for a given type signature $(S_{Type_{TS}}(BTV), TC)$ is defined as the smallest set satisfying the following conditions:*

- For each intersection type ty : $BTV^{(ty)} \subseteq S_{Type_{TS}}(BTV)$
- $TC^0 \subseteq S_{Type_{TS}}(BTV)$
- For $\overline{ty_i} \in S_{Type_{TS}}(BTV)$

$$\begin{aligned} & \cup \{ ? \} \\ & \cup \{ ? \text{ extends } \tau \mid \tau \in S_{Type_{TS}}(BTV) \} \\ & \cup \{ ? \text{ super } \tau \mid \tau \in S_{Type_{TS}}(BTV) \} \end{aligned}$$
and $C \in TC^{(a_1|_{ty_1} \dots a_n|_{ty_n})}$ it holds

$$C\langle \overline{ty_1}, \dots, \overline{ty_n} \rangle \in S_{Type_{TS}}(BTV)$$

if there is a substitution $\sigma = [a_i \mapsto \overline{ty_i} \mid 1 \leq i \leq n]$ such that for all $ty_i = \theta_{i,1} \& \dots \& \theta_{i,n_i}$ either

- *it holds $\overline{ty_i} \leq^* \sigma(\theta_{i,j})$ for all $1 \leq j \leq n_i$ or*
- *for $\overline{ty_i} \in BTV^{(\tau_1 \& \dots \& \tau_m)}$ for each τ_j there is a $\theta_{i,k}$ with $\tau_j \leq^* \sigma(\theta_{i,k})$ or*
- *it holds $\overline{ty_i} = ?$*

or there is a substitution $\sigma = [a_i \mapsto \overline{ty_i'} \mid 1 \leq i \leq n]$ such that for all $ty_i = \theta_{i,1} \& \dots \& \theta_{i,n_i}$

- *it holds $\overline{ty_i} = "? \text{ extends } \overline{ty_i}'"$ and $\overline{ty_i'} \leq^* \sigma(\theta_{i,j})$ for all $1 \leq j \leq n_i$*

where \leq^* is a subtyping ordering (def. 3.10).

The set of intersection types over a set of $\text{SType}_{TS}(BTV)$ is denoted by:

$$I(\text{SType}_{TS}(BTV)) = \{ \theta_1 \& \dots \& \theta_n \mid \theta_i \in \text{SType}_{TS}(BTV), n \in \mathbb{N} \}.$$

$\text{TVar}(ty)$ determines the type variables of an intersection type ty .

? is a wildcard type. Wildcard types are a restricted form of existential types. A formal account of wildcard types is given in [TEPH05].

After the following definitions of the *extends* relation and the *subtyping ordering* we will give some examples to explain the definition of simple types.

Now we define the inheritance hierarchy of Java 5.0 classes. The inheritance hierarchy consists of two different relations: The “*extends* relation” is explicitly defined in Java 5.0 programs by the *extends* respectively the *implements* declarations. The “subtyping relation” is then built as the reflexive, transitive, and instantiating closure of the “*extends* relation”.

Definition 3.9 (Extends relation \leq) Let JC be a set of declared Java 5.0 classes and $TS = (\text{SType}_{TS}(BTV), TC)$ its type signature. For each Java 5.0 class or interface declaration in JC the *extends* relation \leq is defined follows.
BEDINGUNG FUER PARAMETER, MUESSEN IN DEN BOUND DER PARAMETER IHRER SUPERKLASSEN LIEGEN.

1. `class/interface C<expr1, ..., exprn> extends τ'` defines

$$C\langle a_1, \dots, a_n \rangle \leq \tau',$$

where for $1 \leq i \leq n$ holds

$$\text{expr}_i = \text{“}a_i\text{”}$$

or

$$\text{expr}_i = \text{“}a_i \text{ extends } ty_i\text{”}$$

and for $1 \leq i \leq n$: $\text{TVar}(ty_i) \subseteq \{a_1, \dots, a_n\}$.

2. `class C<expr1, ..., exprn> implements τ'_1, \dots, τ'_p` defines

$$C\langle a_1, \dots, a_n \rangle \leq \tau'_1, \dots, C\langle a_1, \dots, a_n \rangle \leq \tau'_p$$

where for $1 \leq i \leq n$ holds

$$\text{expr}_i = \text{“}a_i\text{”}$$

or

$$\text{expr}_i = \text{“}a_i \text{ extends } ty_i\text{”}$$

and for $1 \leq i \leq n$: $\text{TVar}(ty_i) \subseteq \{a_1, \dots, a_n\}$.

Was passiert
mir class CJB
implements XJ
extends CJSuperJB
implements XJ
06-09-13 erledigt
ist schon durch
die type signature
definition
ausgeschlossen.

Now, we define the subtyping relation by building the reflexive, transitive, and instantiating closure of the *extends* relation.

Definition 3.10 (Subtyping relation \leq^*) Let JC be a set of declared Java 5.0 classes, $TS = (\text{SType}_{TS}(BTV), TC)$ its type signature, and \leq the corresponding *extends* relation. The subtyping relation \leq^* is given as the reflexive and transitive closure of the smallest relation satisfying the following conditions:

- if $\theta \leq \theta'$ then $\theta \leq^* \theta'$.
- if $\theta_1 \leq^* \theta_2$ then $\sigma_1(\theta_1) \leq^* \sigma_2(\theta_2)$ for all substitutions σ_1, σ_2 , which satisfy for each type variable a of θ_2 one of the following conditions:
 - $\sigma_1(a) = \sigma_2(a)$ (soundness condition)
 - $\sigma_1(a) = \theta$ and $\sigma_2(a) = ?$
 - $\sigma_1(a) = \theta$ and $\sigma_2(a) = ?$ extends θ' and $\theta \leq^* \theta'$
 - $\sigma_1(a) = ?$ extends θ and $\sigma_2(a) = ?$ extends θ' and $\theta \leq^* \theta'$
 - $\sigma_1(a) = \theta'$ and $\sigma_2(a) = ?$ super θ and $\theta \leq^* \theta'$
 - $\sigma_1(a) = ?$ super θ' and $\sigma_2(a) = ?$ super θ and $\theta \leq^* \theta'$
- $a \leq^* \theta'$ for $a \in BTV^{(\theta_1 \& \dots \& \theta_n)}$ where $\exists \theta_i : \theta_i \leq^* \theta'$.

Example 3.11 1. We consider again the type signature $(\text{SType}_{TS}(BTV), TC)$ from example 3.7. For $C \in TC^{(a|_{\text{I} \mapsto \text{b}} |_{\text{b}})}$ it holds $C\langle B\langle a \rangle, a \rangle \in \text{SType}_{TS}(BTV)$, as for $\sigma = [a \mapsto B\langle a \rangle, b \mapsto a]$ it holds $\sigma(a) = B\langle a \rangle \leq^* I\langle a \rangle = \sigma(I\langle b \rangle)$ and $a \leq^* \sigma(b)$.
Otherwise for example $C\langle C\langle A\langle a \rangle, a \rangle, A\langle a \rangle \rangle$ is no element of $\text{SType}_{TS}(BTV)$, as $C\langle A\langle a \rangle, a \rangle \not\leq^* I\langle A\langle a \rangle \rangle$.

2. If we extend the Java 5.0 program by the class declaration

```
class BJ<c> extends B<BJ<c>> implements J<c> { ... }
```

then the type $D\langle BJ\langle Integer \rangle, Integer \rangle$ is correct, as for $D \in TC^{(a|_{\text{B} \mapsto \text{J} \mapsto \text{b}} |_{\text{b}})}$ there is a substitution $\sigma = [a \mapsto BJ\langle Integer \rangle, b \mapsto Integer]$ with

$$BJ\langle Integer \rangle \leq^* B\langle BJ\langle Integer \rangle \rangle (= \sigma(B\langle a \rangle))$$

and

$$BJ\langle Integer \rangle \leq^* J\langle Integer \rangle (= \sigma(J\langle b \rangle)).$$

3. The type $C\langle ? \text{ extends } A\langle a \rangle, a \rangle$ is also a simple type. As it holds $A\langle a \rangle \leq^* I\langle a \rangle$, it is also allowed to use a wildcard bounded by $A\langle a \rangle$.

Remark The the fourth condition in definition 3.8 for simple type construction with wildcards is more restrictive than in the original Java 5.0 compiler implementation (jdk 1.5.0.08).

The less restrictive original condition allows confusing correct simple types. Let us consider the following example. Again, let be given the Java 5.0 program from example 3.7. A variable declaration

```
D<? extends B<a>, a> v;
```

is correct. But an allocation statement

```
D<? extends B<a>, a> v = new D<B<a>,a> ();
```

is not correct, as $D<B<a>,a>$ is no correct simple type. From our point of view it is not obvious, if this approach make any sense.

Remark It is surprising that in the subtyping definition (def. 3.10) the soundness condition for σ_1 and σ_2 in the first subitem is not $\sigma_1(a) \leq^* \sigma_2(a)$, but $\sigma_1(a) = \sigma_2(a)$. This is necessary to get a sound type system, because of the contravariance problem in object oriented languages.

Let the following Java 5.0 classes be given.

```
class Super { ...}
class Sub extends Super { ...}

class Application {
  public static void main(String[] args) {
    Vector<Super> v = new Vector<Sub> ();
    v.add(new Super()); }
}
```

An element of the type $\text{Vector}<\text{Sub}>$ is assigned to the variable v of the type $\text{Vector}<\text{Super}>$. This is no problem as all elements which have the type Sub have also the type Super . Then a new element of the type Super is added to the vector which is assigned to the variable v . Now we have the problem, that elements of this vector have the type Sub and Super is no subtype of Sub .

As in expression assignments, like $\text{Vector}<\text{Super}> v = \text{new Vector}<\text{Sub}>();$ the type of the right hand side must be a subtype of the left hand side's type, the class Application is not type correct, caused by this substitution's restriction.

But, sometimes assignments like

```
Vector<Super> v = new Vector<Sub> ();
```

would although be desirable. Therefore the wildcards are introduced. It is allowed:

```
Vector<? extends Super> v = new Vector<Sub> ();
```

verglichen ob das
irgendwas mit dem
nicht soundness
von oo-Sprachen
zu tun hat. vgl.
Hinweis in ECCOP
Paper.

Now, $\text{Vector}<\text{Sub}>$ is a subtype of $\text{Vector}<? \text{ extends Super}>$, which means the assignment ist type correct. As “ $? \text{ extends Super}$ ” is not a simple type in the Java 5.0 type system, in this case $v.\text{add}(\text{new Super}());$ is prohibited. There is no method add with the argument type “ $? \text{ extends Super}$ ”, which would be necessary. This means, that problems like in the class Application cannot arise.

Remark In the theory of polymorphic order-sorted algebras [Smo88a, Smo89, Thi93, Plü99a] partial ordered types, like the simple types are called polymorphic order-sorted types.

In the following we will declare an additional ordering on the set of simple types, which we will call the finite closure of the *extends* relation. This ordering is very important to calculate the finite number of general type unifiers by our type unification algorithm (section 4.2).

Definition 3.12 (Finite closure of \leq) Let \leq be an *extends* relation and \leq^* its subtyping relation. Then, the finite closure $\mathbf{FC}(\leq)$ is defined as the reflexive and transitive closure of fc_{base} with

$$fc_{base} = \bigcup_{\theta < \theta'} (\{(\bar{\theta}, \theta') \mid \bar{\theta} \leq^* \theta'\} \cup \{(\theta, \bar{\theta}') \mid \theta \leq^* \bar{\theta}'\}).$$

Remark The finite closure $\mathbf{FC}(\leq)$ is a finite subset of \leq^* .

Now we give some examples to illustrate the abstract definitions.

Example 3.13 The simple types $\text{SType}_{TS}(BTV)$ of the well-known traditional Java inheritance hierarchy (jdk ≤ 1.4) is defined on a type signature $TS = (\text{SType}_{TS}(BTV), TC)$, where all type constructors are contained in $TC^{(0)}$.

```
class Object { ...}

class Boolean extends Object { ...}

interface Comparable { ...}

class Character extends Object implements Comparable { ...}

class Number extends Object { ...}

class Integer extends Number implements Comparable { ...}

class String extends Object implements Comparable { ...}
```

suchen nach dem
Begriff partitions
muss gestrichen
werden
Skeleton durch fite
closure ersetzen

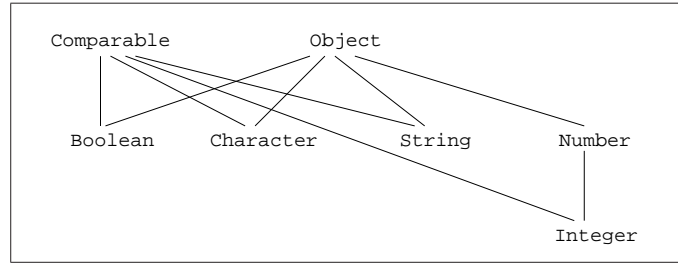


Figure 3.1: Inheritance hierarchy of traditional java

From this cutout of the traditional inheritance hierarchy the subtyping relation is given in fig. 3.1 as an Hasse-diagram.

The second example shows how the subtyping ordering of parameterized classes is arranged.

Example 3.14 Let the following Java 5.0 declarations be given:

```

interface List<a> { ...}

abstract class AbstractList<a> implements List<a> { ...}

class Vector<a> extends AbstractList<a> { ...}

class Stack<a> extends Vector<a> { ...}

```

The extends relation \leq is shown in fig. 3.2. The corresponding subtyping ordering \leq^* comprises for example

$$\text{Stack<Integer>} \leq^* \text{Vector<Integer>},$$

$$\text{Stack<Vector<Integer>>} \leq^* \text{Vector<Vector<Integer>>},$$

and

$$\text{Vector<List<Integer>>} \leq^* \text{List<List<Integer>>}.$$

Following the soundness condition (def. 3.10)

$$\text{Vector<Vector<Integer>>} \not\leq^* \text{List<List<Integer>>}$$

but

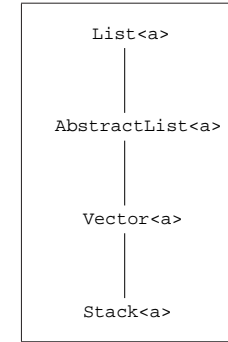
$$\text{Vector<Vector<Integer>>} \leq^* \text{List<? extends List<Integer>>}.$$


Figure 3.2: Type term ordering of a parameterized vector

Furthermore for example the following simple types with wildcards are contained in \leq^* :

$$\text{Vector<? extends Vector<Integer>>} \leq^* \text{List<? extends List<Integer>>}$$

$$\text{Vector<List<Integer>>} \leq^* \text{List<? super Vector<Integer>>}$$

$$\text{Vector<? super List<Integer>>} \leq^* \text{List<? super Vector<Integer>>}$$

If we add the default declaration $\text{Vector<a>} < \text{Object}$ to the extends relation, we get in \leq^* an infinite chain:

$$\begin{aligned}
 & \vdots \\
 & \leq^* \text{Vector<Vector<Vector<? extends Object>>>} \\
 & \leq^* \text{Vector<Vector<? extends Object>>} \\
 & \leq^* \text{Vector<? extends Object>} \\
 & \leq^* \text{Object}
 \end{aligned}$$

Example 3.15 Now we change the Java 5.0 declaration of the last example:

```

abstract class AbstractList<a> implements List<a> { ...}

class Vector<a> extends AbstractList<a> { ...}

class Matrix<a> extends Vector<Vector<a>> { ...}

class ExtMatrix<a> extends Matrix<a> { ...}

```

We consider now the finite closure $\mathbf{FC}(\leq)$. First we determine f_{base} :

$$\begin{aligned}
f_{C_{base}} &= \bigcup_{\theta < \theta'} (\{(\bar{\theta}, \theta') \mid \bar{\theta} \leq^* \theta'\} \cup \{(\theta, \bar{\theta}') \mid \theta \leq^* \bar{\theta}'\}) \\
&= \{(\bar{\theta}, \text{List}\langle a \rangle) \mid \bar{\theta} \leq^* \text{List}\langle a \rangle\} \cup \\
&\quad \{(\text{AbstractList}\langle a \rangle, \bar{\theta}') \mid \text{AbstractList}\langle a \rangle \leq^* \bar{\theta}'\} \quad (1) \\
&\cup \{(\bar{\theta}, \text{AbstractList}\langle a \rangle) \mid \bar{\theta} \leq^* \text{AbstractList}\langle a \rangle\} \cup \\
&\quad \{(\text{Vector}\langle a \rangle, \bar{\theta}') \mid \text{Vector}\langle a \rangle \leq^* \bar{\theta}'\} \quad (2) \\
&\cup \{(\bar{\theta}, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle) \mid \bar{\theta} \leq^* \text{Vector}\langle \text{Vector}\langle a \rangle \rangle\} \cup \\
&\quad \{(\text{Matrix}\langle a \rangle, \bar{\theta}') \mid \text{Matrix}\langle a \rangle \leq^* \bar{\theta}'\} \quad (3) \\
&\cup \{(\bar{\theta}, \text{Matrix}\langle a \rangle) \mid \bar{\theta} \leq^* \text{Matrix}\langle a \rangle\} \cup \\
&\quad \{(\text{ExtMatrix}\langle a \rangle, \bar{\theta}') \mid \text{ExtMatrix}\langle a \rangle \leq^* \bar{\theta}'\} \quad (4)
\end{aligned}$$

Now we consider the sets (1), (2), (3), (4), separately:

$$\begin{aligned}
(1) &= \{(\text{List}\langle a \rangle, \text{List}\langle a \rangle), \\
&\quad (\text{AbstractList}\langle a \rangle, \text{List}\langle a \rangle), \\
&\quad (\text{Vector}\langle a \rangle, \text{List}\langle a \rangle)\} \\
&\cup \{(\text{AbstractList}\langle a \rangle, \text{AbstractList}\langle a \rangle), \\
&\quad (\text{AbstractList}\langle a \rangle, \text{List}\langle a \rangle)\} \\
(2) &= \{(\text{AbstractList}\langle a \rangle, \text{AbstractList}\langle a \rangle), \\
&\quad (\text{Vector}\langle a \rangle, \text{AbstractList}\langle a \rangle)\} \\
&\cup \{(\text{Vector}\langle a \rangle, \text{Vector}\langle a \rangle), \\
&\quad (\text{Vector}\langle a \rangle, \text{AbstractList}\langle a \rangle), \\
&\quad (\text{Vector}\langle a \rangle, \text{List}\langle a \rangle)\} \\
(3) &= \{(\text{Vector}\langle \text{Vector}\langle a \rangle \rangle, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{Matrix}\langle a \rangle, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle)\} \\
&\cup \{(\text{Matrix}\langle a \rangle, \text{Matrix}\langle a \rangle), \\
&\quad (\text{Matrix}\langle a \rangle, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{Matrix}\langle a \rangle, \text{AbstractList}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{Matrix}\langle a \rangle, \text{List}\langle \text{Vector}\langle a \rangle \rangle)\} \\
(4) &= \{(\text{Matrix}\langle a \rangle, \text{Matrix}\langle a \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{Matrix}\langle a \rangle)\} \\
&\cup \{(\text{ExtMatrix}\langle a \rangle, \text{ExtMatrix}\langle a \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{Matrix}\langle a \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{Vector}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{AbstractList}\langle \text{Vector}\langle a \rangle \rangle), \\
&\quad (\text{ExtMatrix}\langle a \rangle, \text{List}\langle \text{Vector}\langle a \rangle \rangle)\}
\end{aligned}$$

Then, the finite closure $\mathbf{FC}(\leq)$ is built by the **transitive and reflexive closure** of the union $(1) \cup (2) \cup (3) \cup (4)$. It is shown in fig. 3.3.

Example 3.16 The last example which we give in this section is an example for the possibility that there are multiple maximal lower bounds in a subtyping ordering. The cause for this is the possibility that one interface can be implemented by different classes.

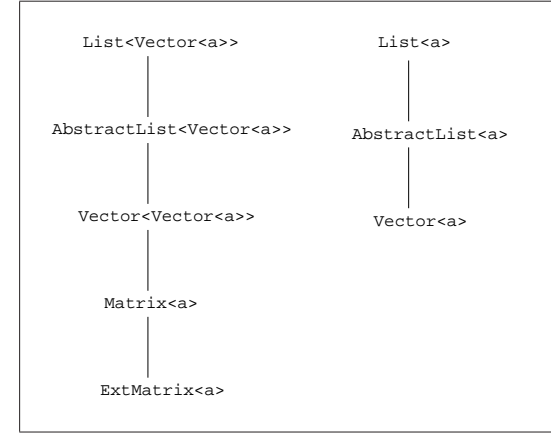


Figure 3.3: Finite closure of an extends relation

```

interface Collection<a> { ... }

interface Serializable { ... }

class Vector<a> extends AbstractList<a>
    implements Collection<a>, Serializable { ... }

class PriorityQueue<a> extends AbstractQueue<a>
    implements Collection<a>, Serializable { ... }

```

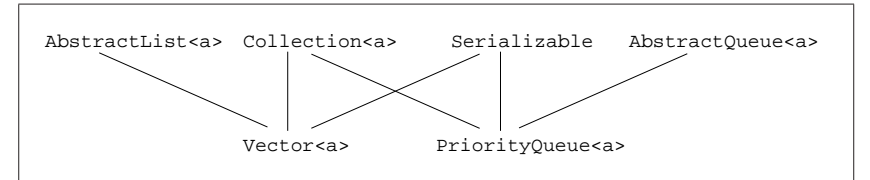


Figure 3.4: Multiple maximal lower bounds

In this example `Vector<a>` as well as `PriorityQueue<a>` implements both interfaces `Collection<a>` and `Serializable`. This leads to two maximal lower bounds of `Collection<a>` and `Serializable`, which means that \leq^* forms no semi-lattice. Furthermore, this is an example that the subtype has more type variables than its supertype.

We close the section with a lemma, which describe a very important property for the type unification (section 4.2) of the finite closure.

Lemma 3.17 *Let $\bar{\theta} = A(\bar{\theta}_1, \dots, \bar{\theta}_n)$ and $\bar{\theta}' = B(\bar{\theta}'_1, \dots, \bar{\theta}'_n)$ be two simple types with $\bar{\theta} \leq^* \bar{\theta}'$. Then, there is a pair $(\theta, \theta') \in \mathbf{FC}(\leq)$ and a substitution σ with $\sigma(\theta) = \bar{\theta}$ and $\sigma(\theta') = \bar{\theta}'$.*

Proof: In the proof two cases must be considered.

Either from definition 3.10 follows that there is a simple types $B(\theta'_1, \dots, \theta'_m)$ and a substitution σ with $(A(a_1, \dots, a_n), B(\theta'_1, \dots, \theta'_m))$ is an element of the reflexive and transitive closure of \leq with $\sigma(A(a_1, \dots, a_n)) = \bar{\theta}$, and $\sigma(B(\theta'_1, \dots, \theta'_m)) = \bar{\theta}'$. As the reflexive and transitive closure of \leq is a subset of $\mathbf{FC}(\leq)$, it holds $(A(a_1, \dots, a_n), B(\theta'_1, \dots, \theta'_m)) \in \mathbf{FC}(\leq)$. But this means that $\theta = A(a_1, \dots, a_n)$ and $\theta' = B(\theta'_1, \dots, \theta'_m)$.

Otherwise, there is no $B(\theta'_1, \dots, \theta'_m)$ with $(A(a_1, \dots, a_n), B(\theta'_1, \dots, \theta'_m))$ in the reflexive and transitive closure of \leq , where there is a substitution σ with $\sigma(A(a_1, \dots, a_n)) = \bar{\theta}$ and $\sigma(B(\theta'_1, \dots, \theta'_m)) = \bar{\theta}'$. In this case follows from definition 3.10, that there are simple types τ_1, \dots, τ_o and a substitution σ with

$$A(a_1, \dots, a_n) \leq \tau_1 \leq^* \dots \leq^* \tau_p \leq^* B(\theta'_1, \dots, \theta'_m)$$

and $\sigma(A(a_1, \dots, a_n)) = \bar{\theta}$, and $\sigma(B(\theta'_1, \dots, \theta'_m)) = \bar{\theta}'$. From this follows by definition 3.12 that $(A(a_1, \dots, a_n), B(\theta'_1, \dots, \theta'_m)) \in \mathbf{FC}(\leq)$. But this means that $\theta = A(a_1, \dots, a_n)$ and $\theta' = B(\theta'_1, \dots, \theta'_m)$. ■

Chapter 4

Type Unification

In this chapter we consider the type unification problem of **Java 5.0** type terms. The type unification problem is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

The algorithm solving the type unification problem is an important base of the type inference algorithm which we will give in the next chapter.

In this chapter we will first give a small overview of similar type unification problems. In the second section we will give an algorithm for the solution of the type unification problem. The section is divided in two parts. In the first part we give a solution for type terms without wildcards. In the second part we give a solution for the whole set of simple types (including type terms with wildcards).

4.1 Overview

In the last chapter we assert, that the type system of **Java 5.0** can be considered as polymorphic order-sorted types. Besides our functional language **OBJ-P** [Plü99a, Plü99b] in logical languages **TEL** [Smo88b, Smo88a, Smo89], **PROTOS-L** [BBM94, BM94, Bei95], and the typed logic programs of [HT92] polymorphic order-sorted types are used. The logical language **TEL** allows subtype relationships between polymorphic types having different arities (e.g. $\text{List}(\mathbf{a}) \leq \text{myLi}(\mathbf{a}, \mathbf{b})$), which means that the subtyping relation contains infinite chains. In the typed logic programs of [HT92] subtype relationships between polymorphic types are allowed only between type constructors of the same arity. The type system of **PROTOS-L** was derived from **TEL** by disallowing any explicit subtype relationship between polymorphic type constructors.

The given type inference algorithms and type unification algorithms, respectively, in [Smo88a, Smo89] and [HT92] are incomplete. They are even incomplete for the restricted type system of [Bei95]. In [Bei95] a complete type unification algorithm is given, which can be extended to the type system of [HT92]. They solved the problem, that there is no most general unifier, by the calculation of a set of equivalent unifiers, respective a representant of the set of most general unifiers. This approach is very similar to our approach

for the type system of **OBJ-P** [Plü99a].

In [Smo88a, Smo89] the type unification problem of **TEL** without restrictions on the polymorphic type constructors is mentioned as an open problem.

If we compare the type system of **TEL** to the **Java 5.0** type system, we assert that these type systems are very similar. The **Java 5.0** type system restricted to simple types without parameter bounds (but including the wildcard constructions) has the same properties wrt. type unification. The only difference is, that in **TEL** the number arguments in a subtype relationship of a supertype type can be greater, whereas in **Java 5.0** the number of arguments of a subtype can be greater. This means that in **TEL** infinite chains has a lower bound and in **Java 5.0** an upper bound. Let us consider the following example: In **TEL** for $\text{List}(\mathbf{a}) \leq \text{myLi}(\mathbf{a}, \mathbf{b})$ is holds:

$$\text{List}(\mathbf{a}) \leq \text{myLi}(\mathbf{a}, \text{List}(\mathbf{a})) \leq \text{myLi}(\mathbf{a}, \text{myLi}(\mathbf{a}, \text{List}(\mathbf{a}))) \leq \dots$$

In contrast in **Java 5.0** for $\text{myLi}(\mathbf{b}, \mathbf{a}) < \text{List}(\mathbf{a})$ it holds:

$$\dots \leq^* \text{myLi}(\text{myLi}(\text{? extends List}(\mathbf{a}), \mathbf{a}), \mathbf{a}) \leq^* \text{myLi}(\text{? extends List}(\mathbf{a}), \mathbf{a}) \leq^* \text{List}(\mathbf{a})$$

The open type unification problem of [Smo88a, Smo89] is caused by the infinite chains. We will give a solution for the open problem. First, we restrict in section 4.2.1 the simple types to type terms without wildcards. This means that there are no infinite chain, following from the soundness condition of definition 3.8. We give a type unification algorithm for the restricted set of simple types. In this approach we do not consider equivalence classes of unifiers, as in [Bei95] and [Plü99a]. This means that there is sometimes no most general unifier. There are more than one general unifier. The type unification problem is not longer unitary, but finitary. Then, in section 4.2.2 we extend this algorithm to simple types with wildcards. This means that we will solve the open problem of [Smo88a, Smo89]. We will prove that the type unification problem for simple types with wildcards is also finitary.

Our type unification algorithm bases on the algorithm of Herbrand [Her30] and A. Martelli and U. Montanari [MM76, MM82]¹ solving the original untyped unification problem. There is another unification algorithm in [Rob65] solving also the untyped unification problem. This algorithm is the base of the **ml** type inference algorithm [Mil78, DM82]. In [BS01] the unification theory is presented comprehensively.

In general unification algorithms are the base of logical programming languages. The original unification is the base of untyped logical programming languages as **PROLOG** [?]. For logical languages with an (non-polymorphic) order-sorted type system the order-sorted unification [SS89, MGS89, SNGM89, Wal90] is the base. A further extended polymorphic order-sorted type system has the polymorphic order-sorted unification as its base. The polymorphic order-sorted unification algorithms needs a type unification algorithm ??? for the type inference of logical variables ??? instead of a simple infimum function as in the non-polymorphic case. Therefore the type unification problem is considered in [Smo88a, Smo89], [HT92], and [Bei95].

¹In the unification overview article [Sie89] Siekmann said that Martelli and Montanari rediscovered the original unification algorithm of Herbrand.

4.2 Type unification algorithm

We give an algorithm which determines a substitution σ for two type terms θ_1, θ_2 with

$$\sigma(\theta_1) \leq^* \sigma(\theta_2).$$

We call this the *type unification problem*.

We divide the consideration of the type unification in two parts. First we consider only simple types without wildcards. Then also simple types with wildcards are considered. The following definition gives the subset of simple types without wildcards.

Definition 4.1 (Type terms without wildcards) Let $TS = (\text{SType}_{TS}(BTV), TC)$ be a type signature. The set of type terms without wildcards $T_{TC}(BTV)$ over TS is given as the smallest set with the following conditions:

1. For each type ty $BTV^{(ty)} \subseteq T_{TC}(BTV)$
2. $TC^0 \subseteq T_{TC}(BTV)$
3. For $C \in TC^{(a_1|_{ty_1} \dots a_n|_{ty_n})}$ it holds

$$C \langle \sigma(a_1), \dots, \sigma(a_n) \rangle \in T_{TC}(BTV)$$

if for the substitution σ holds $\sigma(a_i) \leq^* \sigma(ty_i)$ for all $1 \leq i \leq n$.

In the following, $T_{TC}(BTV)$ denotes a set of type terms without wildcards, $\text{SType}_{TS}(BTV)$ denotes a set of simple types (definition 3.8), \leq denotes an *extends* relation (definition 3.9), \leq^* the corresponding subtyping relation (definition 3.10), and $\mathbf{FC}(\leq)$ its finite closure (definition 3.12).

Definition 4.2 (Type unifier) Given type terms $\theta_1, \theta_2 \in \text{SType}_{TS}(BTV)$ and a substitution σ with $\sigma(\theta_1) \leq^* \sigma(\theta_2)$, σ is called a *type unifier*.

Definition 4.3 (General type unifier) Given type terms $\theta_1, \theta_2 \in \text{SType}_{TS}(BTV)$ and a type unifier σ . σ is called *general type unifier* if there is no type unifier σ' and no substitution σ'' such that $\sigma = \sigma'' \circ \sigma'$.

In the following we present algorithms, which compute general type unifiers. The algorithms unify a set of *equations* $Eq = \{\theta_1 \leq \theta'_1, \dots, \theta_n \leq \theta'_n\}$ where θ_i, θ'_j are simple types, and $\theta \leq \theta'$ means, that the two type terms should be unified, such that $\sigma(\theta) \leq^* \sigma(\theta')$. During the unification algorithms \leq is replaced by \doteq , where $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

Our type unification algorithms are based on the unification algorithm of Martelli and Montanari [MM76, MM82]. The main difference is, that in the original unification a unifier is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$. This means that a pair $a \doteq \theta$ determines that the unifier substitutes a by the term θ . In contrast a pair $a \leq \theta$ respectively $\theta \leq a$ leads

to multiple correct substitutions. All types smaller than θ and greater than θ respectively are correct substitutions for a . This means that there are multiple unifiers.

The next definition gives a form of the set of equations for the end configuration of the algorithms.

Definition 4.4 (Solved form) A set of equations Eq is in *solved form*, if

$$Eq = \{a_1 \doteq \theta_1, \dots, a_n \doteq \theta_n\}$$

where $a_1, \dots, a_n \in TV$ are pairwise different variables, $\theta_1, \dots, \theta_n \in \text{SType}_{TS}(BTV)$, and for all $i, j \in \{1, \dots, n\}$ $a_i \notin \text{TVar}(\theta_j)$ holds.

4.2.1 Type Unification without Wildcards

Now, we give the type unification algorithm for type terms without wildcards. This algorithm is extended in the next section to the type unification algorithm, which computes type unifiers for all simple types (including simple types with wildcards).

Algorithm 4.5 The type unification algorithm \mathbf{TUnify}_{\leq^*} is given as follows

Input: Set of equations $Eq = \{\theta_1 \leq \theta'_1, \dots, \theta_n \leq \theta'_n\}$

Precondition: $\theta_i, \theta'_i \in T_{TC}(BTV)$ for $1 \leq i \leq n$.

Output: Set of all general type unifiers $Uni = \{\sigma_1, \dots, \sigma_m\}$

Postcondition: For all $1 \leq i \leq n$ and for all $1 \leq j \leq m$ holds $(\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i)) \in T_{TC}(BTV) \times T_{TC}(BTV)$

The algorithm itself is given in seven steps:

$$1. Eq_1 = \{\theta \leq \theta' \mid (\theta \leq \theta') \in Eq \wedge ((\theta, \theta' \notin TV) \vee (\theta, \theta' \in TV))\}$$

$$2. Eq_2 = Eq \setminus Eq_1$$

$$3. Eq_{set} =$$

$$\{Eq_1\} \times \left(\bigotimes_{(a \leq \theta') \in Eq_2} \{a \doteq \sigma(\theta) \mid (\theta \leq^* \theta') \in \mathbf{FC}(\leq), \sigma = \text{Unify}(\theta', \theta')\} \right) \quad (4.1)$$

$$\times \left(\bigotimes_{(\theta \leq a) \in Eq_2} \{a \doteq \theta' \mid \theta \leq^* \theta'\} \right) \quad (4.2)$$

4. Repeated application of the rules *reduce1*, *reduce2*, *erase*, *swap*, and *adapt* of figure 4.1 to all elements of Eq_{set} . The end configuration Eq'_{set} is reached if for each element no rule is applicable.

NOCHMAL'S UEBERPRUEFEN
FC INSBESONDERE VARIABLENBEZEICHNUNGEN
BY
AUCH
UEBERALL

$$\begin{aligned}
(\text{reduce1}) \quad & \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \leq D \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \doteq \theta'_1, \dots, \theta_{\pi(n)} \doteq \theta'_n \}} \\
& \text{where} \\
& \bullet C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& \bullet \{ a_1, \dots, a_n \} \subseteq TV \\
& \bullet \pi \text{ is a permutation} \\
(\text{reduce2}) \quad & \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \doteq C \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_1 \doteq \theta'_1, \dots, \theta_n \doteq \theta'_n \}} \\
(\text{erase}) \quad & \frac{Eq \cup \{ \theta \doteq \theta' \}}{Eq} \quad \theta = \theta' \\
(\text{swap}) \quad & \frac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}} \quad \theta \notin TV, a \in TV \\
(\text{adapt}) \quad & \frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto \theta_i \mid 1 \leq i \leq n] \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}
\end{aligned}$$

where there are $\bar{\theta}'_1, \dots, \bar{\theta}'_m$ with

- $(D \langle a_1, \dots, a_n \rangle \leq^* D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle) \in \mathbf{FC}(\leq)$

Figure 4.1: Java 5.0 type unification

5. Application of the following *subst* rule

$$\frac{Eq \cup \{ a \doteq \theta \}}{Eq[a \mapsto \theta] \cup \{ a \doteq \theta \}} \quad a \text{ occurs in } Eq \text{ but not in } \theta$$

to each element of $Eq' \in Eq'_{set}$ as often as possible. The result is Eq''_{set} .

6. (a) Foreach $Eq'' \in Eq''_{set}$ which has changed in the last step start again with the first step.
- (b) Build the union Eq'''_{set} of all results of (a) and $Eq'' \in Eq''_{set}$ which has not changed in the last step.
7. $Uni = \{ Eq''' \mid Eq''' \in Eq'''_{set} \text{ is in solved form} \}$

The result is then a set of unifiers

$$\{ \sigma_1, \dots, \sigma_n \},$$

where for each $Eq_i \in Uni$ holds $\sigma_i = \{ a \mapsto \theta \mid a \doteq \theta \}$.

In the following we explain the algorithm. The algorithm takes a set of type term pairs and delivers a set of general type unifiers. The precondition and the postcondition assures that the constraints which are given by bounded class parameters are maintained. The explanation of the algorithm itself we do by the explanation of the changes in comparison to the original algorithm [MM76, MM82]. The main difference is that our type unification problem is not unitary, which means that there are more than one general unifier. This follows from pairs $a \leq \theta$ respectively $\theta \leq a$, where a is a type variable. The existence of these pairs means that a can be substituted by all type terms smaller respectively greater than θ . Each of these types generates an own general unifier. These generations are done in the steps one, two, and three of the algorithm. After that the known rules of the original unification algorithm are applied. These rules are changed in a few positions:

reduce1 rule: The *reduce1* rule follows from the construction of \leq^* where from

$$C(a_1, \dots, a_n) \leq D(a_{\pi(1)}, \dots, a_{\pi(n)})$$

follows $C(\theta_1, \dots, \theta_n) \leq^* D(\theta'_1, \dots, \theta'_n)$ only if $\theta_{\pi(i)} = \theta'_i$ for $1 \leq i \leq n$. The *reduce1* rule replaces the demands that type terms are smaller (\leq) to demands that they are equal (\doteq).

reduce2 rule: The *reduce2* rule follows directly from the original algorithm. This rule is applied to the subterms after an application of the *reduce1* rule.

erase rule: The *erase* rule is unchanged.

swap rule: The *swap* rule is unchanged.

adapt rule: The *adapt* rule is necessary to handle extends relationships defined by

$$\text{class } C \langle a_1, \dots, a_n \rangle \text{ extends } D \langle D_1 \langle \dots \rangle, \dots, D_m \langle \dots \rangle \rangle.$$

The rule *adapt* reduces type terms which are built by class declarations like this. This reduction is done in two steps. First a pattern in the finite closure is searched, which matches the given pair. From lemma 3.17 follows, that such a pattern exists, if the unification will not fail. Then, the smaller type of the given pair is replaced by a type term, where in the greater type of the pattern, the arguments of the smaller type are instantiated. Finally, the reduction is done by the *reduce2* rule.

The algorithm which is given if we disclaim the *adapt* rule solves the type unification problem for extends relations \leq which satisfy the following condition: If $\lambda_1 \leq \lambda_2$ then the type term λ_1 is of the form $C_1(a_1, \dots, a_n)$ and the type term λ_2 of the form $C_2(a_{\pi(1)}, \dots, a_{\pi(n)})$ for type variables $\{ a_1, \dots, a_n \}$. $\pi : \{ 1, \dots, n \} \rightarrow \{ 1, \dots, n \}$ is a permutation.

subst rule: The *subst* rule is separated from step four of the original algorithm. This is done, as there are pairs of type terms $a \leq^* b$, where a and b are type variables. If a or b is substituted, step one respectively step two and three must be applied, which is done by step six (a).

Step six (b) summarizes the different sets of type term pairs. Step seven finally filters the type term pairs, which are solved form. This means, that they build type unifiers.

Now, we give five examples for the algorithm. The first one shows how more than one unifier is generated.

Example 4.6 We extend the type signature and the subtyping relation of example 3.14, such that for $TS = (\text{SType}_{TS}(\text{BTV}), TC)$ holds:

$$TC^{()} = \{ \text{Integer}, \text{Object} \}, TC^{(a|_a)} = \{ \text{Vector}, \text{Stack}, \text{AbstractList}, \text{List} \}, \\ TC^{(a|_a \ b|_b)} = \{ \text{Pair} \},$$

$$\text{Integer} \leq \text{Object},$$

and

$$\text{Stack}\langle a \rangle \leq \text{Vector}\langle a \rangle \leq \text{AbstractList}\langle a \rangle \leq \text{List}\langle a \rangle.$$

Now, we apply the unification algorithm to

$$Eq = \{ a < \text{Vector}\langle \text{Integer} \rangle, \\ \text{AbstractList}\langle \text{Object} \rangle < b, \\ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \}$$

In the first two steps the set of type term pairs are separated. The result of the first step are type type term pairs, where either both type terms are type variables or both type terms are no type variables.

$$Eq_1 = \{ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \}$$

The result of the second step is:

$$Eq_2 = \{ a < \text{Vector}\langle \text{Integer} \rangle, \\ \text{AbstractList}\langle \text{Object} \rangle < b \}$$

In the third step all types which can be mapped to these type variables are determined and the other pairs are added to each of these elements:

$$Eq_{set} = \{ \{ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \} \\ \cup \{ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \} \\ \cup \{ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \} \\ \cup \{ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \}, \\ \{ \text{AbstractList}\langle \text{Pair}\langle \text{Stack}\langle \text{Integer} \rangle, d \rangle \rangle \\ < \text{List}\langle \text{Pair}\langle c, \text{List}\langle \text{Integer} \rangle \rangle \rangle, \\ c < d \} \\ \cup \{ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \} \}$$

In the fourth step the rules of fig. 4.1 are applied to the elements of Eq_{set} . The result is

$$Eq'_{set} = \{ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, c < d, \\ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, c < d, \\ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, c < d, \\ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, c < d, \\ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \} \}$$

In the fifth step the *subst* rule is applied:

$$Eq''_{set} = \{ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, \\ \text{Stack}\langle \text{Integer} \rangle < \text{List}\langle \text{Integer} \rangle, \\ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, \\ \text{Stack}\langle \text{Integer} \rangle < \text{List}\langle \text{Integer} \rangle, \\ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Vector}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, \\ \text{Stack}\langle \text{Integer} \rangle < \text{List}\langle \text{Integer} \rangle, \\ b \doteq \text{AbstractList}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \}, \\ \{ c \doteq \text{Stack}\langle \text{Integer} \rangle, d \doteq \text{List}\langle \text{Integer} \rangle, \\ \text{Stack}\langle \text{Integer} \rangle < \text{List}\langle \text{Integer} \rangle, \\ b \doteq \text{List}\langle \text{Object} \rangle, a \doteq \text{Stack}\langle \text{Integer} \rangle \} \}$$

The sixth step is divided in (a) and (b):

(a): The algorithm is applied again to the elements of Eq''_{set} . In the first three steps nothing is changed, as Eq_2 is empty.

In step four $Stack<Integer> \leq List<Integer>$ is reduced and finally erased.

(b): This leads to the result

$$Eq'''_{set} = \{ \{ c \doteq Stack<Integer>, d \doteq List<Integer>, \\ b \doteq AbstractList<Object>, a \doteq Vector<Integer> \}, \\ \{ c \doteq Stack<Integer>, d \doteq List<Integer>, \\ b \doteq List<Object>, a \doteq Vector<Integer> \}, \\ \{ c \doteq Stack<Integer>, d \doteq List<Integer>, \\ b \doteq AbstractList<Object>, a \doteq Stack<Integer> \}, \\ \{ c \doteq Stack<Integer>, d \doteq List<Integer>, \\ b \doteq List<Object>, a \doteq Stack<Integer> \} \}$$

In the seventh step nothing is changed, such that the result are four general unifiers:

$$\{ \{ a \mapsto Vector<Integer>, b \mapsto AbstractList<Object>, \\ c \mapsto Stack<Integer>, d \mapsto List<Integer> \}, \\ \{ a \mapsto Vector<Integer>, b \mapsto List<Object>, \\ c \mapsto Stack<Integer>, d \mapsto List<Integer> \}, \\ \{ a \mapsto Stack<Integer>, b \mapsto AbstractList<Object>, \\ c \mapsto Stack<Integer>, d \mapsto List<Integer> \}, \\ \{ a \mapsto Stack<Integer>, b \mapsto List<Object>, \\ c \mapsto Stack<Integer>, d \mapsto List<Integer> \} \}$$

In the following the present an example, where the recursive application of the algorithm generates new general unifiers.

Example 4.7 Let the subtyping ordering be the same as in example 4.6. Only the set of equations is changed:

$$Eq = \{ a \leq Vector<Integer>, \\ List<Object> \leq b, \\ List<Pair<AbstractList<Integer>, d>> \leq List<Pair<c, List<Integer>>>, \\ c \leq e \}$$

As in $List<Object> \leq b$ the $AbstractList$ is changed to $List$ in comparison to the above example, Eq_{set} is a set of two pairs:

$$Eq_{set} = \{ \{ List<Pair<AbstractList<Integer>, d>> \leq List<Pair<c, List<Integer>>>, \\ c \leq e \} \\ \cup \{ b \doteq List<Object>, a \doteq Stack<Integer> \}, \\ \{ List<Pair<AbstractList<Integer>, d>> \leq List<Pair<c, List<Integer>>>, \\ c \leq e \} \\ \cup \{ b \doteq List<Object>, a \doteq Vector<Integer> \} \}$$

Then in the fourth step the application of the rules of fig. 4.1 leads to:

$$Eq'_{set} = \{ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, c \leq e, \\ b \doteq List<Object>, a \doteq Stack<Integer> \}, \\ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, c \leq e, \\ b \doteq List<Object>, a \doteq Vector<Integer> \} \}$$

In the fifth step the subst rule is applied. Now the variable e is not substituted.

$$Eq''_{set} = \{ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ AbstractList<Integer> \leq e, \\ b \doteq List<Object>, a \doteq Stack<Integer> \}, \\ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ AbstractList<Integer> \leq e, \\ b \doteq List<Object>, a \doteq Vector<Integer> \} \}$$

Then in part (a) of the sixth step the pair

$$AbstractList<Integer> \leq e$$

generates for each element of Eq''_{set} two new elements.

This means that in step (b) these four elements are united to the set of type term pairs Eq'''_{set} :

$$Eq'''_{set} = \{ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ e \doteq AbstractList<Integer>, \\ b \doteq List<Object>, a \doteq Stack<Integer> \}, \\ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ e \doteq List<Integer>, \\ b \doteq List<Object>, a \doteq Stack<Integer> \}, \\ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ e \doteq AbstractList<Integer>, \\ b \doteq List<Object>, a \doteq Vector<Integer> \}, \\ \{ c \doteq AbstractList<Integer>, d \doteq List<Integer>, \\ e \doteq List<Integer>, \\ b \doteq List<Object>, a \doteq Vector<Integer> \} \}$$

The result is then

```
{ { a ↦ Stack<Integer>, b ↦ List<Object>,
    c ↦ AbstractList<Integer>,
    d ↦ List<Integer>, e ↦ AbstractList<Integer> },
  { a ↦ Stack<Integer>, b ↦ List<Object>,
    c ↦ AbstractList<Integer>,
    d ↦ List<Integer>, e ↦ List<Integer> },
  { a ↦ Vector<Integer>, b ↦ List<Object>,
    c ↦ AbstractList<Integer>,
    d ↦ List<Integer>, e ↦ AbstractList<Integer> },
  { a ↦ Vector<Integer>, b ↦ List<Object>,
    c ↦ AbstractList<Integer>,
    d ↦ List<Integer>, e ↦ List<Integer> } }
```

Both examples do not use the *adapt* rule. We give another examples, which show, how the *adapt* rule works.

Example 4.8 Now, we consider again a part of the type signature from example 3.15. Let the $TS = (S\text{Type}_{TS}(BTV), TC)$ be given as

$$TC^{()} = \{ \text{Object} \}, TC^{(a|a)} = \{ \text{Vector}, \text{Matrix}, \text{List} \}$$

and

$$\text{Matrix}\langle a \rangle \leq \text{Vector}\langle \text{Vector}\langle a \rangle \rangle.$$

Now, we apply the unification algorithm to

$$Eq = \{ \text{Matrix}\langle \text{List}\langle b \rangle \rangle \leq \text{Vector}\langle \text{Vector}\langle \text{List}\langle \text{Object} \rangle \rangle \}$$

In the first three steps nothing happens. If we consider in the fourth step the rules of fig. 4.1, we see that neither the erase, nor the swap rule is applicable.

The *reduce1* rule is also not applicable, as there is no relationship $\text{Matrix}\langle a \rangle \leq \text{Vector}\langle a \rangle$. Now the *adapt* rule is necessary.

As

$$(\text{Matrix}\langle a \rangle \leq^* \text{Vector}\langle \text{Vector}\langle a \rangle \rangle) \in \mathbf{FC}(\leq)$$

from

$$Eq = \{ \text{Matrix}\langle \text{List}\langle b \rangle \rangle \leq \text{Vector}\langle \text{Vector}\langle \text{List}\langle \text{Object} \rangle \rangle \}$$

follows by the *adapt* rule:

$$Eq_{\text{set}} = \{ \{ \text{Vector}\langle \text{Vector}\langle \text{List}\langle b \rangle \rangle \doteq \text{Vector}\langle \text{Vector}\langle \text{List}\langle \text{Object} \rangle \rangle \} \}.$$

Then the *reduce2* rule leads finally to the result:

$$\{ \{ b \mapsto \text{Object} \} \}$$

A further example is an example for type terms with F-bounded parameters.

Example 4.9 For this example we refine the example from section 2.1.3. The class *ReprChange* is divided in class where only the *element* is set and in a subclass, where additionally the *element* can be got in the other representation.

```
interface ConvertibleTo<a> {
    a convert();
}

class ReprChange<a extends ConvertibleTo<b>,
    b extends ConvertibleTo<a>> {

    a element;

    void set(b x) { element = x.convert(); }
}

class ReprChangeBack<a extends ConvertibleTo<b>,
    b extends ConvertibleTo<a>> extends ReprChange<a, b> {

    b get() { return element.convert(); }
}

class INT implements ConvertibleTo<FLOAT> { ...}

class FLOAT implements ConvertibleTo<INT> { ...}
```

WEITERES BEISPIEL IN (Types_GJAVA/translation.tex am Schluss).

The type signature $TS = (S\text{Type}_{TS}(BTV), TC)$ is then given as

$$\begin{aligned} TC^{()} &= \{ \text{Object}, \text{INT}, \text{FLOAT} \} \\ TC^{(a|a)} &= \{ \text{ConvertibleTo} \} \\ TC^{(a|\text{ConvertibleTo}\langle b \rangle \quad b|\text{ConvertibleTo}\langle a \rangle)} &= \{ \text{ReprChange}, \text{ReprChangeBack} \} \end{aligned}$$

and

$$\begin{aligned} \text{INT} &\leq \text{ConvertibleTo}\langle \text{FLOAT} \rangle, \\ \text{FLOAT} &\leq \text{ConvertibleTo}\langle \text{INT} \rangle, \\ \text{ReprChangeBack}\langle a, b \rangle &\leq \text{ReprChange}\langle a, b \rangle \end{aligned}$$

Now, we apply the unification algorithm to

$$Eq = \{ \text{ReprChangeBack}\langle \text{INT}, b \rangle \leq \text{ReprChange}\langle c, \text{FLOAT} \rangle \}$$

The precondition is maintained as both type terms are elements of $T_{TC}(BTV)$.

In the first three steps nothing happens. In the fourth step the *reduce1* rule is applied.

With the swap rule follows then

ACHTUNG: Bei beschränkten Parametern darf keine reflexive Huelle gebildet werden

$$Eq'_{set} = \{ \{ c \doteq \text{INT}, b \doteq \text{FLOAT} \} \}$$

In the fifth and sixth step nothing happens, this means $Eq'''_{set} = Eq'_{set}$. The only element of Eq'''_{set} is in solved form and the postcondition is maintained, as

$$(\text{ReprChangeBack}\langle \text{INT}, \text{FLOAT} \rangle \leq^* \text{ReprChange}\langle \text{INT}, \text{FLOAT} \rangle) \in T_{TC}(BTV) \times T_{TC}(BTV).$$

This means that it holds $\text{Uni} = Eq'''_{set}$ and the result is:

$$\{ \{ c \mapsto \text{INT}, b \mapsto \text{FLOAT} \} \}$$

The last example, which we will present is an example, where interfaces are used.

Example 4.10 We define a simple abstract subtyping ordering by the following program:

```
interface A { ...}

interface B { ...}

class C implements A, B { ...}

class D implements A, B { ...}
```

The corresponding type signature $TS = (\text{SType}_{TS}(BTV), TC)$ is given as:

$$TC^0 = \{ A, B, C, D \}$$

and the extends relation given as:

$$C \leq A, D \leq A, C \leq B, D \leq B$$

In figure 4.2 the subtyping ordering is shown grafically.

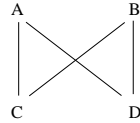


Figure 4.2: Interface subtyping ordering

Now, we apply the unification algorithm to

$$Eq = \{ a \triangleleft A, a \triangleleft B \}.$$

In the first step nothing happens. The result of the second step is

$$Eq_2 = \{ a \triangleleft A, a \triangleleft B \}$$

In the third step all types which can be mapped to these type variables are determined:

$$Eq_{set} = \{ \{ a \doteq A, a \doteq B \}, \\ \{ a \doteq C, a \doteq B \}, \\ \{ a \doteq D, a \doteq B \}, \\ \{ a \doteq A, a \doteq C \}, \\ \{ a \doteq C, a \doteq C \}, \\ \{ a \doteq D, a \doteq C \}, \\ \{ a \doteq A, a \doteq D \}, \\ \{ a \doteq C, a \doteq D \}, \\ \{ a \doteq D, a \doteq D \} \}$$

In the fourth step nothing happens. In the fifth step the subst is applied multiply:

$$Eq''_{set} = \{ \{ a \doteq A, A \doteq B \}, \\ \{ a \doteq C, C \doteq B \}, \\ \{ a \doteq D, D \doteq B \}, \\ \{ a \doteq A, A \doteq C \}, \\ \{ a \doteq C, C \doteq C \}, \\ \{ a \doteq D, D \doteq C \}, \\ \{ a \doteq A, A \doteq D \}, \\ \{ a \doteq C, C \doteq D \}, \\ \{ a \doteq D, D \doteq D \} \}$$

In the sixth step the algorithm is applied again to each element of Eq''_{set} . Only $\{ a \doteq C, C \doteq C \}$ and $\{ a \doteq D, D \doteq D \}$ are changed. The erase rule erases in step four $C \doteq C$ and $D \doteq D$, respectively.

In the seventh step all sets, which are in solved form, are filtered. Only $\{ a \doteq C \}$ and $\{ a \doteq D \}$ are in solved form, such that the result is:

$$\{ \{ a \mapsto C \}, \{ a \mapsto D \} \}.$$

Theorem 4.11 The type unification algorithm determines all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.

The type unification problem is solved by algorithm 4.5. This means that the algorithm 4.5 is sound and complete.

Proof: AUF F-BOUNDED PARAMETERS UND INTERFACES NOCHMALS CHECKEN
We do the proof in two steps: First we prove the *soundness* and then the *completeness*.

Soundness

We take a substitution which is the result of the algorithm. Then we show that this substitution is a unifier of the algorithm input. We do this by proofs for each transformation, that if a substitution is a unifier of the result of the transformation then the substitution is also a unifier of the input of the transformation.

Let

$$\sigma = \{ a_1 \mapsto \theta_1, \dots, a_n \mapsto \theta_n \}$$

be a result of the algorithm. Then

$$Eq = \{ a_1 \doteq \theta_1, \dots, a_n \doteq \theta_n \}$$

is one element of the result of step seven. It is obvious, that σ is a unifier of *elem*.

Now we prove the soundness for each transformation starting by step seven.

Step seven: As there are only the pairs of type terms, which are in solved form, filtered, the set of pairs of type terms Eq , which generates the unifier, is unchanged. But this means that σ is NARH WIE VOR a general unifier.

Step six: In step six only different sets of pairs of type terms are united, which are generated by the steps four and five. This means also that the unifier is unchanged.

Step five: In step five a is substituted by θ if $(a \doteq \theta) \in Eq$. From this follows, that we have to prove: If σ is a general unifier of $Eq[a \mapsto \theta]$ then σ is also general unifier of Eq . As $a \mapsto \sigma(\theta)$ is an element of σ , σ is also an unifier of Eq .

Step four: The soundness of the *reduce2*, *erase*, and *swap* rules are obvious.

reduce1 rule: If σ is a general unifier of $\{ \theta_i \doteq \theta'_{\pi(i)} \mid 1 \leq i \leq n \}$ then it is to prove that σ is also a general unifier of $\{ C(\theta_1, \dots, \theta_n) \leq D(\theta'_{\pi(1)}, \dots, \theta'_{\pi(n)}) \}$, if $C(a_1, \dots, a_n) \leq^* D(a_{\pi(1)}, \dots, a_{\pi(n)})$, where a_1, \dots, a_n are type variables. This follows direct from the construction of type terms (def. 3.10).

adapt rule: If σ is a general unifier of

$$D'(\bar{\theta}'_1, \dots, \bar{\theta}'_m)[a_i \mapsto \theta_i \mid 1 \leq i \leq n] \doteq D'(\theta'_1, \dots, \theta'_m),$$

where for type variables a_1, \dots, a_n it holds $(D(a_1, \dots, a_n) \leq^* D'(\bar{\theta}'_1, \dots, \bar{\theta}'_m))$, it is to prove that σ is also a general unifier of

$$D(\theta_1, \dots, \theta_n) \leq D'(\theta'_1, \dots, \theta'_m).$$

This follows also direct from the construction of type terms (def. 3.10).

Step three: It is to prove, if σ is a general unifier of $\{ a \doteq \theta \}$ then σ is also a general unifier of $\{ a \leq \theta' \}$ and of $\{ \bar{\theta} \leq a \}$ for $\theta \leq^* \theta'$ and $\bar{\theta} \leq^* \theta$, respectively.

As it holds $\sigma(\theta) \leq^* \sigma(\theta')$ and $\sigma(\bar{\theta}) \leq^* \sigma(\theta)$, σ is also a general unifier of $\{ a \leq \theta' \}$ and of $\{ \bar{\theta} \leq a \}$, respectively.

Step one and two: In the first two steps the pairs are only filtered. This means that the pairs are unchanged. From this follows that the general unifier is also unchanged.

We have proved for all transformations that if a substitution is a general unifier of the transformations's result the same substitution is also a general unifier of the transformations's input. This means that the algorithm is sound.

Completeness

For the completeness we have to prove that if there is a unifier of a set of type term pairs then this unifier is also determined by the algorithm. We prove this, by showing for each transformation of the algorithm, that if a substitution is a general unifier of a set of type terms before the transformation is done, then the substitution is also a general unifier of at least one set of type term pairs after the transformation.

Step one, two, and three: The pairs of type terms, which are filtered in step one are unchanged in step three, such these pairs are not needed to be considered.

We consider the type term pairs, which are filtered in step 2:

Let σ be a general unifier of $\{ a \leq D\langle \theta_1, \dots, \theta_n \rangle \}$. It is to prove, that there is one set $Eq \in \{ \{ a \doteq \theta \mid \theta \leq^* D\langle \theta_1, \dots, \theta_n \rangle \} \}$, where σ is a general unifier.

From definition 3.10 follows, that there is a type term $C\langle \bar{\theta}_1, \dots, \bar{\theta}_m \rangle$ with

$$C\langle \bar{\theta}_1, \dots, \bar{\theta}_m \rangle \leq^* D\langle \theta_1, \dots, \theta_n \rangle \text{ and } \sigma(a) = C\langle \sigma(\bar{\theta}_1), \dots, \sigma(\bar{\theta}_m) \rangle,$$

where $\{ C\langle \bar{\theta}_1, \dots, \bar{\theta}_m \rangle \} \in Eq$ and σ is its general unifier.

Step four: It is obvious that the application of the *erase* and the *swap* rule obtain all unifier.

As it holds $\sigma(D\langle \theta_1, \dots, \theta_n \rangle) = D\langle \sigma(\theta_1), \dots, \sigma(\theta_n) \rangle$ the transformations by the rules *reduce1*, *reduce2*, and *adapt* obtain also all unifiers.

Step five: If σ is general unifier of $\{ a \doteq \theta \} \cup Eq$, it is to prove that σ is also a general unifier of $Eq[a \mapsto \theta]$.

From the property that $[a \mapsto \sigma(\theta)]$ is an element of the general unifier σ follows that for any type term $\bar{\theta}$ it holds $\sigma(\bar{\theta}) = \sigma(\bar{\theta}[a \mapsto \theta])$. This means that σ is also a general unifier of $Eq[a \mapsto \theta]$.

Step six: As no pair of type terms is transformed, nothing is to be considered.

Step seven: In step seven the pairs of type terms which are in solved form are filtered. All other sets of pairs of type terms have no unifier, thus all general unifiers are obtained.

We have proved for all transformations of the algorithm that a general unifier of the transformation's input is also a general unifier of the transformation's output.

As for a general unifier $\sigma = [a_i \mapsto \theta_i \mid 1 \leq i \leq n]$ of a set of type term pairs Eq , which are in solved form, it holds $Eq = \{ a_i \doteq \theta_i \mid 1 \leq i \leq n \}$, the general unifier σ is determined by the algorithm.

Summerized, this means, that the algorithm is complete. ■

4.2.2 Type Unification with Wildcards

Now we will extend our algorithm to simple types with wildcards. The main problem is, that there are infinite chains in the subtyping orderings. Therefore it is not obvious, if the type unification problem is solvable respectively, if it is finitary. Finitary means that the number of general unifiers for two given simple types is finite. As said before in [Smo88a, Smo89] this is denoted as an open problem.

We will solve this problem and will show that all elements of the infinite chains are instances of one element. This means that the unifier, which map this a element to a type variable is a general unifier.

In the following we will use $\gamma\theta$ as an abbreviation for the type term “ γ extends θ ”.

Before presenting the algorithm we have to give some auxiliary definitions.

Auxiliary definitions

The first function **smallerFC** determines all smaller types, which can be derived from the finite closure. This means that infinite chains (cp. example 3.15) are not built. Additionally a substitution is determined. This is necessary as class declarations of the form `class E<a,b> extends D<B<a>,b>` are allowed. In example 4.14 we will see, how the substitution is used.

Definition 4.12 (smallerFC) Let $\theta' \in \text{SType}_{TS}(BTV)$ be a simpletype. Then

$$\text{smallerFC}(\theta') = \{ (\sigma(\text{fresh}(\theta)), \sigma) \mid (\theta \leq^* \vec{\theta}') \in \text{FC}(\leq), \sigma \in \text{smallerSubst}(\text{Unify}(\theta', \text{fresh}(\vec{\theta}')))) \}$$

where

$$\begin{aligned} \text{smallerSubst}(\sigma) = & \bigotimes_{(a \mapsto \gamma\theta) \in \sigma} \{ ([a \mapsto \gamma\theta] \cup \sigma'), ([a \mapsto \bar{\theta}] \cup \sigma') \mid (\bar{\theta}, \sigma') \in \text{smallerFC}(\theta) \} \\ & \times \bigotimes_{(a \mapsto \theta) \in \sigma} \{ [a \mapsto \theta] \}, \end{aligned}$$

Unify is the usual unification of terms, and *fresh*(θ) determines fresh type variables in the type term θ .

Remark For each sub-term θ'_i of a type term θ' the function **smallerFC**(θ') determines all elements $\bar{\theta}_i$ of the finite closure, such that for all subtypes θ_i of θ'_i there is a substitution σ with $\sigma(\theta_i) = \bar{\theta}_i$ (cp. corollary ??).

The next function **smallerFC_?** is the corresponding function to **smallerFC** for the sub-terms of a type term. In this case, only if the sub-terms starts with “ γ extends”, subtypes exists (cp. the subtyping definition 3.10)

Definition 4.13 (smallerFC_?) Let $\theta' \in \text{SType}_{TS}(BTV)$ be a simpletype. Then

$$\text{smallerFC}_\gamma(\theta') = \begin{cases} \{ (\bar{\theta}, \sigma), (\gamma\bar{\theta}, \sigma) \mid (\bar{\theta}, \sigma) \in \text{smallerFC}(\theta) \} & \text{if } \theta' = \gamma\theta \\ \{ (\sigma(A\langle a_1, \dots, a_n \rangle), \sigma) \mid \sigma \in \text{smallerSubst}([a_1 \mapsto \theta_1, \dots, a_n \mapsto \theta_n]) \} & \text{if } \theta = A\langle \theta_1, \dots, \theta_n \rangle \\ (\theta', []) & \text{if } \theta' = \text{const} \end{cases}$$

The following example explains the abstract definitions.

Example 4.14 Let the following Java 5.0 program be given:

```
class A { ... }

class B<a> extends A { ... }

class D<a,b> { ... }

class E<a,b> extends D<B<a>,b> { ... }
```

This means that there are infinite chains in \leq^* as

$$\dots \leq^* B < B < \gamma A > > \leq^* B < \gamma A > \leq^* B < \gamma A >.$$

Now we build **smallerFC**($D < a, \gamma A >$). As “ γA ” is a sub-term there is also an infinite chain $\dots \leq^* \text{smallerFC}(D < a, \gamma A >)$

smallerFC($D < a, \gamma A >$):

- $un1 = \text{Unify}(D < a, \gamma A >, D < aa, bb >) = \{ aa \mapsto a, bb \mapsto \gamma A \}$
- **smallerSubst**($un1$):
 - **smallerFC**(A) = $\{ (A, []), (B < aa' >, []) \}$
 - **smallerFC**(a) = $\{ (a, []) \}$

Result of **smallerSubst**($un1$):

- $\{ [aa \mapsto a, bb \mapsto \gamma A] := \sigma_1, [aa \mapsto a, bb \mapsto A] := \sigma_2, [aa \mapsto a, bb \mapsto \gamma B < aa' >] := \sigma_3, [aa \mapsto a, bb \mapsto B < aa' >] := \sigma_4 \}$
- $un2 = \text{Unify}(D < a, \gamma A >, D < B < a''' >, b' >) = \{ a \mapsto B < a''' >, b' \mapsto \gamma A \}$
- **smallerSubst**($un2$):
 - **smallerFC**($B < a''' >$) = $\{ (B < a''' >, []) \}$
 - **smallerFC**(A) = $\{ (A, []), (B < a''' >, []) \}$

Result of **smallerSubst**($un2$):

$$\{ [a \mapsto B < a''' >, b' \mapsto \gamma A] := \sigma_5, [a \mapsto B < a''' >, b' \mapsto A] := \sigma_6, [a \mapsto B < a''' >, b' \mapsto \gamma B < a''' >] := \sigma_7, [a \mapsto B < a''' >, b' \mapsto B < a''' >] := \sigma_8 \}$$

Result of **smallerFC**($D\langle a, ?A \rangle$):

$$\{ (D\langle a, ?A \rangle, \sigma_1), (D\langle a, A \rangle, \sigma_2), (D\langle a, ?B\langle aa' \rangle \rangle, \sigma_3), (D\langle a, B\langle aa' \rangle \rangle, \sigma_4), \\ (\underline{E\langle a''', ?A \rangle}, \sigma_5), (\underline{E\langle a''', A \rangle}, \sigma_6), (\underline{E\langle a''', ?B\langle a'''' \rangle \rangle}, \sigma_7), (\underline{E\langle a''', B\langle a'''' \rangle \rangle}, \sigma_8) \}$$

The result contains not all smaller elements of $D\langle a, ?A \rangle$. There are infinite chains of smaller elements, which are smaller than the underlined elements. They are not determined by **smallerFC**.

If we consider the second line of results, we see that the variable a is disappeared. During the unification we need variables like this. Therefore, we introduced the substitution σ as a further result of **smallerFC**. In σ the substitution of a is contained.

The next definition $\leq_?$ is the corresponding definition to \leq^* for sub-terms of type terms.

Definition 4.15 ($\leq_?$) Let \leq^* be a subtyping relation. Then $\leq_?$ is defined as follows

$$\begin{aligned} X < \theta_1, \dots, \theta_n &\leq_?^* Y < \theta'_1, \dots, \theta'_m && \text{if } X < \theta_1, \dots, \theta_n \leq^* Y < \theta'_1, \dots, \theta'_m \\ ?X < \theta_1, \dots, \theta_n &\leq_?^* ?Y < \theta'_1, \dots, \theta'_m && \text{if } X < \theta_1, \dots, \theta_n \leq^* Y < \theta'_1, \dots, \theta'_m \\ X < \theta_1, \dots, \theta_n &\leq_?^* ?Y < \theta'_1, \dots, \theta'_m && \text{if } Y < \theta'_1, \dots, \theta'_m \leq^* X < \theta_1, \dots, \theta_n \\ ?X < \theta_1, \dots, \theta_n &\leq_?^* ?Y < \theta'_1, \dots, \theta'_m && \text{if } Y < \theta'_1, \dots, \theta'_m \leq^* X < \theta_1, \dots, \theta_n \end{aligned}$$

The type unification algorithm

In this algorithm we unify a set of equations $Eq = \{\theta_1 < \theta'_1, \dots, \theta_n < \theta'_n\}$ where $\theta_i, \theta'_j \in \text{SType}_{TS}(BTV)$, and $\theta < \theta'$ means, that the two type terms should be unified, such that $\sigma(\theta) \leq^* \sigma(\theta')$.

During the unification algorithm $<$ is replaced by $<_?$ and \doteq , respectively, where $\theta <_? \theta'$ means that the two sub-terms of type terms should be unified, such that $\sigma(\theta) \leq_?^* \sigma(\theta')$ and $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

The base of our algorithm is the unification algorithm of Herbrand [Her30] and A. Martelli and U. Montanari [MM76, MM82]. The main extension is, that in the original unification a unifier is demanded, such that $\sigma(\theta_1) = \sigma(\theta_2)$. This means that a pair $a \doteq \theta$ determines that the unifier substitutes a by the term θ . In contrast a pair $a < \theta$ respectively $\theta < a$ leads to multiple correct substitutions. All type terms smaller than θ and greater than θ , respectively, are correct substitutions for a . This means that there are multiple unifiers and the unification is not unitary.

The type unification succeeds if the result is in solved form.

Definition 4.16 (Solved form) A set of equations Eq is in *solved form*, if

$$Eq = \{a_1 \doteq \theta_1, \dots, a_n \doteq \theta_n\}$$

where $a_1, \dots, a_n \in TV$ are pairwise different variables, $\theta_1, \dots, \theta_n \in \text{SType}_{TS}(BTV)$, and for all $i, j \in \{1, \dots, n\}$ $a_i \notin \text{TVar}(\theta_j)$ holds.

The type unification algorithm is given as follows

Input: Set of equations $Eq = \{\theta_1 < \theta'_1, \dots, \theta_n < \theta'_n\}$

Precondition: $\theta_i, \theta'_i \in \text{SType}_{TS}(BTV)$ for $1 \leq i \leq n$.

Output: Set of all general type unifiers $Uni = \{\sigma_1, \dots, \sigma_m\}$

Postcondition: For all $1 \leq i \leq n$ and for all $1 \leq j \leq m$ holds $(\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i))$.

The algorithm itself is given in seven steps:

1. Repeated application of the rules *reduce1*, *reduce2*, *reduce3*, *erase1*, *erase2*, *erase3*, *swap*, and *adapt* of figure 4.4 to all elements of Eq . The end configuration Eq' is reached if for each element no rule is applicable.
2. $Eq_1 = \text{Subset of pairs, which consists only of type variables.}$
3. $Eq_2 = Eq' \setminus Eq_1$
4. $Eq_{set} = \{Eq_1\} \times \left(\bigotimes_{(a <_? \theta') \in Eq_2} \{([a \doteq \theta] \cup \sigma) \mid (\theta, \sigma) \in \text{smallerFC}(\theta')\} \right) \\ \times \left(\bigotimes_{(a <_? \theta') \in Eq_2} \{([a \doteq \theta] \cup \sigma) \mid (\theta, \sigma) \in \text{smallerFC}_?(\theta')\} \right) \\ \times \left(\bigotimes_{(\theta < a) \in Eq_2} \{[a \doteq \theta'] \mid \theta \leq^* \theta'\} \right) \\ \times \left(\bigotimes_{(\theta <_? a) \in Eq_2} \{[a \doteq \theta'] \mid \theta \leq_?^* \theta'\} \right)$
5. Application of the following *subst* rule

$$(\text{subst}) \quad \frac{Eq \cup \{a \doteq \theta\}}{Eq[a \mapsto \theta] \cup \{a \doteq \theta\}} \quad a \text{ occurs in } Eq \text{ but not in } \theta$$

for each $a \doteq \theta$ in each element of $Eq \in Eq_{set}$. The result is Eq'_{set} .

6. (a) Foreach $Eq' \in Eq'_{set}$ which has changed in the last step start again with the first step.
(b) Build the union Eq''_{set} of all results of (a) and $Eq' \in Eq'_{set}$ which has not changed in the last step.
7. $Uni = \{Eq'' \mid Eq'' \in Eq''_{set} \text{ is in solved form}\}$
The result is then a set of unifiers

$$\{\sigma_1, \dots, \sigma_n\},$$

where for each $Eq_i \in Uni$ holds $\sigma_i = \{a \mapsto \theta \mid a \doteq \theta\}$.

In the following we explain the algorithm. The main idea of the algorithm is, that for each general unifier there is one set of type term pairs. A set of type term pairs is multiplied in step four, as for each pair $a < \theta'$ respectively $\theta < a$, for each θ with $(\theta, \sigma) \in \text{smallerFC}(\theta')$ and for each θ' with $\theta \leq^* \theta'$ a new set of type term pairs is generated.

The rules of figure 4.4 are applied to each set of type term pairs. These rules are derived from the original unification algorithm.

reduce1 rule: The *reduce1* rule follows from the construction of \leq^* where from

$$C(a_1, \dots, a_n) \leq D(a_{\pi(1)}, \dots, a_{\pi(n)})$$

follows $C(\theta_1, \dots, \theta_n) \leq^* D(\theta'_1, \dots, \theta'_n)$ if and only if $\theta_{\pi(i)} \leq^* \theta'_i$ for $1 \leq i \leq n$. This is the reason, why the condition \leq of the whole type term is changed to the condition $\leq_?$ of the sub-terms.

reduce2 rule: The *reduce2* rule is the corresponding rule to the *reduce1* rule for sub-terms of type terms.

reduce3 rule: The *reduce3* rule follows directly from the original algorithm. This rule is applied to type term pairs, which are built in step 4 and 5 of the algorithm.

erase rules: The *erase* rules erase type term pairs, which are in the respective relationship.

swap rule: The *swap* rule corresponds to the original *swap* rule.

adapt rule: The *adapt* rule adapts type term pairs, which are built by class declarations like

$$\text{class } C\langle a_1, \dots, a_n \rangle \text{ extends } D\langle D_1\langle \dots \rangle, \dots, D_m\langle \dots \rangle \rangle.$$

The smaller type is replaced by a type term, which has the same outermost type name as the greater type. Its sub-term are determined by the finite closure. The instantiations are maintained.

The *subst* rule is not applied in the first step. The reason is, that before the *subst* rule can be applied, pairs like $a \doteq \theta$ must be generated. This is done not until step four.

In step six (a) the algorithm is applied again to the changed sets of type term pairs.

Step six (b) summarizes the different sets of type term pairs.

Step seven, finally, filters the type term pairs which are unifiers. This means, that have to be in solved form.

Now, we give two examples for the algorithm. The first one shows how more than one unifier is generated.

Example 4.17 In this example we use the standard Java 5.0 types like *Number*, *Integer*, *Vector*, and *Stack*. It holds $\text{Integer} < \text{Number}$ and $\text{Stack}\langle a \rangle < \text{Vector}\langle a \rangle$.

As a start configuration we use

$$Eq = \{ \text{Stack}\langle a \rangle < \text{Vector}\langle ?\text{Number} \rangle, \text{AbstractList}\langle \text{Integer} \rangle < \text{List}\langle a \rangle \}.$$

In the first step the *reduce1* rule is applied:

$$Eq' = \{ a <_? ?\text{Number}, \text{Integer} <_? a \}$$

In the second step nothing happens:

$$Eq_1 = \emptyset$$

The result of the third step is:

$$Eq_2 = Eq'$$

In the fourth step we receive:

$$Eq_{set} = \{ \{ a \doteq ?\text{Number}, a \doteq \text{Integer} \}, \\ \{ a \doteq ?\text{Number}, a \doteq ?\text{Number} \}, \\ \{ a \doteq \text{Number}, a \doteq \text{Integer} \}, \\ \{ a \doteq \text{Number}, a \doteq ?\text{Number} \}, \\ \{ a \doteq ?\text{Integer}, a \doteq \text{Integer} \}, \\ \{ a \doteq ?\text{Integer}, a \doteq ?\text{Number} \}, \\ \{ a \doteq \text{Integer}, a \doteq \text{Integer} \}, \\ \{ a \doteq \text{Integer}, a \doteq ?\text{Number} \} \}$$

In the fifth step the rule *subst* is applied:

$$Eq_{set} = \{ \{ \text{Integer} \doteq ?\text{Number}, a \doteq \text{Integer} \}, \\ \{ ?\text{Number} \doteq ?\text{Number}, a \doteq ?\text{Number} \}, \\ \{ \text{Integer} \doteq \text{Number}, a \doteq \text{Integer} \}, \\ \{ ?\text{Number} \doteq \text{Number}, a \doteq ?\text{Number} \}, \\ \{ \text{Integer} \doteq ?\text{Integer}, a \doteq \text{Integer} \}, \\ \{ ?\text{Number} \doteq ?\text{Integer}, a \doteq ?\text{Number} \}, \\ \{ \text{Integer} \doteq \text{Integer}, a \doteq \text{Integer} \}, \\ \{ ?\text{Number} \doteq \text{Integer}, a \doteq ?\text{Number} \} \}$$

Now we have the continue with the first step (step six (a)). With the application of the *erase3* rule and step 7, we get

$$Uni = \{ \{ a \doteq ?\text{Number} \}, \{ a \doteq \text{Integer} \} \}$$

The other sets of type term pairs are not in solved form.

This means that there are two unifiers $\{ a \mapsto ?\text{Number} \}$ and $\{ a \mapsto \text{Integer} \}$.

In this example the subtyping ordering contains no infinite chains. We give another example, which shows, how the algorithm determines unifiers, if there are infinite chains.

Example 4.18 Let the following Java 5.0 program be given:

```
class A { ... }

class B<a> extends A { ... }
```

This means that there is an infinite chain in the ordering \leq^* :

$$\dots \leq^* B\langle B\langle B\langle ?A \rangle \rangle \rangle \leq^* B\langle B\langle ?A \rangle \rangle \leq^* B\langle ?A \rangle.$$

Let the the following start configuration be given:

$$Eq = \{ B\langle a \rangle < B\langle B\langle B\langle ?A \rangle \rangle \rangle, a < B\langle ?A \rangle \}.$$

After the fourth step we receive:

$$Eq_{set} = \{ \{ a \doteq B<B<B<?A>>>, a \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a \doteq B<?A> \} \\ \{ a \doteq B<B<B<?A>>>, a \doteq B<B<a''>> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a \doteq B<B<a''>> \} \}$$

With the fifth step we get

$$Eq'_{set} = \{ \{ a \doteq B<B<B<?A>>>, B<B<B<?A>>> \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<B<a'>>>> \doteq B<?A> \} \\ \{ a \doteq B<B<B<?A>>>, B<B<B<?A>>> \doteq B<B<a''>> \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<B<a'>>>> \doteq B<B<a''>> \} \}$$

Now we have the continue with the first step (step six (a)). With the application of the *reduce3* and the swap rule, we get

$$\{ \{ a \doteq B<B<B<?A>>>, B<B<B<?A>>> \doteq ?A \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<B<a'>>>> \doteq ?A \} \\ \{ a \doteq B<B<B<?A>>>, a'' \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a'' \doteq B<B<a'>> \} \}$$

As the first two set of type term pairs are not in solved form, this means that the set of unifier is given as:

$$\{ \{ a \mapsto B<B<B<?A>>> \}, \{ a \mapsto B<B<B<B<a'>>>> \} \}.$$

This example shows how $a < B<?A>$ is solved by a general type unifier, although there is an infinite chain. All unifiers smaller than $B<B<B<?A>>>$ respective $B<B<B<B<a'>>>>$ are instances of themselves.

The following theorem shows, that the type unification problem is solved by the type unification algorithm.

Theorem 4.19 *The type unification algorithm determines all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.*

Now we give a sketch of the proof.

Soundness

For the proof of the soundness, we take a substitution which is a result of the algorithm. Then we show that this substitution is a general unifier of the algorithm input. We do this by proving for each transformation, that if the substitution is a unifier of the result of the transformation then the substitution is also a unifier of the input of the transformation.

Completeness

For the completeness we prove that if there is a general unifier of a set of type term pairs then this unifier is also determined by the algorithm. We prove this, by showing for each transformation of the algorithm, that if a substitution is the unifier of a set of type terms before the transformation is done, then the substitution is also a unifier of at least one set of type term pairs after the transformation.

Corollary 4.20 (Finitary) *The type unification of Java 5.0 type terms with wildcards is finitary.*

The first function **smallerFC** determines all smaller types, which can be derived from the finite closure. This means that infinite chains (cp. example 3.15) are not built.

Definition 4.21 (smallerFC) *Let $\theta' \in \text{SType}_{TS}(BTV)$ be a simpletype. Then $\text{smallerFC}(\theta') =$*

$$\{ \sigma(\theta) \mid (\theta \leq^* \bar{\theta}') \in \text{FC}(\leq), \sigma \in \text{smallerSubst}(\text{Unify}(\theta', \text{fresh}(\bar{\theta}')) \}$$

where

$$\text{smallerSubst}(\sigma) = \bigotimes_{(a \mapsto ?\bar{\theta}, a \mapsto \bar{\theta} \mid \bar{\theta} \in \text{smallerFC}(\theta))} \{ a \mapsto ?\bar{\theta}, a \mapsto \bar{\theta} \mid \bar{\theta} \in \text{smallerFC}(\theta) \} \times \bigotimes_{(a \mapsto \theta) \in \sigma} \{ a \mapsto \theta \}$$

and $\text{fresh}(\theta)$ determines arbitrary fresh type variables in the type term θ .

Remark For each sub-term θ'_i of a type term θ' the function **smallerFC**(θ') determines all elements $\bar{\theta}_i$ of the finite closure, such that for all subtypes θ_i of θ'_i there is a substitution σ with $\sigma(\bar{\theta}_i) = \theta_i$ (cp. corollary ??).

The next function **smallerFC_?** is the corresponding function to **smallerFC** for the sub-terms of a type term. In this case, only if the sub-terms starts with “? extends”, subtypes exists (cp. the subtyping definition 3.10)

Definition 4.22 (smallerFC_?) *Let $\theta' \in \text{SType}_{TS}(BTV)$ be a simpletype. Then $\text{smallerFC}_?(\theta') =$*

$$\begin{cases} \{ \bar{\theta}, ?\bar{\theta} \mid \bar{\theta} \in \text{smallerFC}(\theta) \} & \text{if } \theta' = ?\theta \\ \{ \sigma(A<a_1, \dots, a_n>) \mid \\ \sigma \in \text{smallerSubst}([a_1 \mapsto \theta_1, \dots, a_n \mapsto \theta_n]) \} & \text{if } \theta = A<\theta_1, \dots, \theta_n> \\ \theta' & \text{if } \theta' = \text{const} \end{cases}$$

The following example explains the abstract definitions.

Example 4.23 *Let the following Java 5.0 programm be given:*

```
class A { ... }

class B<a> extends A { ... }

class C<a> extends B<a> { ... }

class D<a,b> { ... }

class E<a,b> extends D<B<a>,b> { ... }
```

This means that there is infinite chain in the ordering \leq^* :

$$\dots \leq^* B \langle B \langle B \langle ?A \rangle \rangle \rangle \leq^* B \langle B \langle ?A \rangle \rangle \leq^* B \langle ?A \rangle.$$

In [Smo89] the problem is, that it is not obvious how to solve an equation $a \leq_{\mathbf{B}} \langle \mathbf{A} \rangle$. The solution that it is sufficient to consider the smaller elements in the finite closure with new variables. As all smaller elements are substitutes from these smaller elements in the finite closure, the continuing of the unification algorithm determines the correct substitution. Now we build **smallerFC**($\mathbf{B} \langle \mathbf{B} \langle \mathbf{D} \langle \mathbf{a}, \mathbf{A} \rangle \rangle \rangle$):

- $- un1 := \text{Unify}(B<?B<?D<a, ?A>>>, B<a'>) = \{a' \mapsto ?B<?D<a, ?A>>\}$
 $- \text{smallerSubst}(un1) :$
 $- \text{smallerFC}(B<?D<a, ?A>>) :$
 $- un2 = \text{Unify}(B<?D<a, ?A>>, B<a''>) = \{a'' \mapsto ?D<a, ?A>\}$
 $- \text{smallerSubst}(un2) :$
 $- \text{smallerFC}(D<a, ?A>) :$
 $- un3 = \text{Unify}(D<a, ?A>, D<aa, bb>) = \{aa \mapsto a, bb \mapsto ?A\}$
 $- \text{smallerSubst}(un3) :$
 $- \text{smallerFC}(A) = \{A, B<aa'\>\}$
 $- \text{smallerFC}(a) = \{a\}$
Result of smallerSubst(un3) :
 $\{[aa \mapsto a, bb \mapsto ?A], [aa \mapsto a, bb \mapsto A],$
 $[aa \mapsto a, bb \mapsto ?B<aa'\>], [aa \mapsto a, bb \mapsto B<aa'\>]\}$
 $- un4 = \text{Unify}(D<a, ?A>, D<B<a'''>, b'\>) = \{a \mapsto B<a'''>, b' \mapsto ?A\}$
 $- \text{smallerSubst}(un4) :$
 $- \text{smallerFC}(B<a'''>) = \{B<a'''>\}$
 $- \text{smallerFC}(A) = \{A, B<a'''>\}^2$
Result of smallerSubst(un4) :
 $\{[a \mapsto B<a'''>, b' \mapsto ?A], [a \mapsto B<a'''>, b' \mapsto A],$
 $[a \mapsto B<a'''>, b' \mapsto ?B<a'''>], [a \mapsto B<a'''>, b' \mapsto B<a'''>]\}$
Result of smallerFC(D<a, ?A>) :
 $\{D<a, ?A>, D<a, A>, D<a, ?B<aa'\>>, D<a, B<aa'\>>,$
 $E<a'''>, ?A>, E<a'''>, A>, E<a'''>, ?B<a'''>>, E<a'''>, B<a'''>>\}$

²At this position not all small types are built.

Result of smallerSubst(un2):

$$\begin{aligned} & \{ [a'' \mapsto D\langle a, ?A \rangle], [a'' \mapsto ?D\langle a, ?A \rangle], [a'' \mapsto D\langle a, A \rangle], [a'' \mapsto ?D\langle a, A \rangle], \\ & [a'' \mapsto D\langle a, ?B\langle aa' \rangle \rangle], [a'' \mapsto ?D\langle a, ?B\langle aa' \rangle \rangle], [a'' \mapsto D\langle a, B\langle aa' \rangle \rangle], \\ & [a'' \mapsto ?D\langle a, B\langle aa' \rangle \rangle], [a'' \mapsto E\langle a''', ?A \rangle], [a'' \mapsto ?E\langle a''', ?A \rangle], \\ & [a'' \mapsto E\langle a''', A \rangle], [a'' \mapsto ?E\langle a''', A \rangle], [a'' \mapsto E\langle a''', ?B\langle aa'' \rangle \rangle], \\ & [a'' \mapsto ?E\langle a''', ?B\langle aa'' \rangle \rangle], [a'' \mapsto E\langle a''', B\langle aa'' \rangle \rangle], \\ & [a'' \mapsto ?E\langle a''', B\langle aa'' \rangle \rangle] \} \end{aligned}$$

Result of smallerFC(B<?D<a, ?A>>):

{ B<D<a, ?A>>, B<?D<a, ?A>>, B<D<a, A>>, B<?D<a, A>>,
B<D<a, ?B<aa'>>>, B<?D<a, ?B<aa'>>>, B<D<a, B<aa'>>>,
B<?D<a, B<aa'>>>, B<E<a''' , ?A>>, B<?E<a''' , ?A>>,
B<E<a''' , A>>, B<?E<a''' , A>>, B<E<a''' , ?B<a'''>>>,
B<?E<a''' , ?B<a'''>>>, B<E<a''' , B<a'''>>>,
B<E<a''' , B<a'''>>> }

Result of smallerSubst(un1):

$$\{ \begin{aligned} &[a' \mapsto B \langle D \langle a, A \rangle \rangle], [a' \mapsto ?B \langle D \langle a, A \rangle \rangle], [a' \mapsto B \langle ?D \langle a, A \rangle \rangle], [a' \mapsto ?B \langle ?D \langle a, A \rangle \rangle], \\ &[a' \mapsto B \langle D \langle a, A \rangle \rangle], [a' \mapsto ?B \langle D \langle a, A \rangle \rangle], [a' \mapsto B \langle ?D \langle a, A \rangle \rangle], [a' \mapsto ?B \langle ?D \langle a, A \rangle \rangle], \\ &[a' \mapsto B \langle D \langle a, ?B \langle aa' \rangle \rangle \rangle], [a' \mapsto ?B \langle D \langle a, ?B \langle aa' \rangle \rangle \rangle], [a' \mapsto B \langle ?D \langle a, ?B \langle aa' \rangle \rangle \rangle], \\ &[a' \mapsto ?B \langle ?D \langle a, ?B \langle aa' \rangle \rangle \rangle], [a' \mapsto B \langle D \langle a, B \langle aa' \rangle \rangle \rangle], [a' \mapsto ?B \langle D \langle a, B \langle aa' \rangle \rangle \rangle], \\ &[a' \mapsto B \langle ?D \langle a, B \langle aa' \rangle \rangle \rangle], [a' \mapsto ?B \langle ?D \langle a, B \langle aa' \rangle \rangle \rangle], [a' \mapsto B \langle E \langle a''' , A \rangle \rangle], \\ &[a' \mapsto ?B \langle E \langle a''' , A \rangle \rangle], [a' \mapsto B \langle ?E \langle a''' , A \rangle \rangle], [a' \mapsto ?B \langle ?E \langle a''' , A \rangle \rangle], \\ &[a' \mapsto B \langle E \langle a''' , A \rangle \rangle], [a' \mapsto ?B \langle E \langle a''' , A \rangle \rangle], [a' \mapsto B \langle ?E \langle a''' , A \rangle \rangle], \\ &[a' \mapsto ?B \langle ?E \langle a''' , A \rangle \rangle], [a' \mapsto B \langle E \langle a''' , ?B \langle a'''' \rangle \rangle \rangle], \\ &[a' \mapsto ?B \langle E \langle a''' , ?B \langle a'''' \rangle \rangle \rangle], [a' \mapsto B \langle ?E \langle a''' , ?B \langle a'''' \rangle \rangle \rangle], \\ &[a' \mapsto ?B \langle ?E \langle a''' , ?B \langle a'''' \rangle \rangle \rangle], [a' \mapsto B \langle E \langle a''' , B \langle a'''' \rangle \rangle \rangle], \\ &[a' \mapsto ?B \langle E \langle a''' , B \langle a'''' \rangle \rangle \rangle], [a' \mapsto B \langle ?E \langle a''' , B \langle a'''' \rangle \rangle \rangle], \\ &[a' \mapsto ?B \langle ?E \langle a''' , B \langle a'''' \rangle \rangle \rangle] \} \end{aligned}$$

Result of smallerFC(B<?B<?D<a, ?A>>>):

[illegible]

The result contains not all smaller elements of $B <_? B <_? D <_? A >>>$. There are infinite chains of smaller elements, which are smaller than the underlined elements. They are not determined by *smallerFC*.

The next definition $\leq_?$ is the corresponding definition to \leq^* for sub-terms of type terms.

Definition 4.24 ($\leq_?$) *Let \leq^* be a subtyping relation. Then*

$$\leq_? = \{ (X < \theta_1, \dots, \theta_n, Y < \theta'_1, \dots, \theta'_m) \mid X < \theta_1, \dots, \theta_n \leq^* Y < \theta'_1, \dots, \theta'_m \text{ and } (X = Y \text{ or } X = ?C' \text{ or } Y = ?D') \}$$

In this algorithm we unify a set of *equations* $Eq = \{ \theta_1 < \theta'_1, \dots, \theta_n < \theta'_n \}$ where $\theta_i, \theta'_j \in \mathbf{SType}_{TS}(BTV)$, and $\theta < \theta'$ means, that the two type terms should be unified, such that $\sigma(\theta) \leq^* \sigma(\theta')$.

During the unification algorithm $<$ is replaced by \doteq , where $\theta \doteq \theta'$ means that the two type terms should be unified, such that $\sigma(\theta) = \sigma(\theta')$.

Algorithm 4.25 The type unification algorithm **TUnify** $_{\leq^*}$ is given as follows

Input: Set of equations $Eq = \{ \theta_1 < \theta'_1, \dots, \theta_n < \theta'_n \}$

Precondition: $\theta_i, \theta'_i \in \mathbf{SType}_{TS}(BTV)$ for $1 \leq i \leq n$.

Output: Set of all general type unifiers $Uni = \{ \sigma_1, \dots, \sigma_m \}$

Postcondition: For all $1 \leq i \leq n$ and for all $1 \leq j \leq m$ holds $(\sigma_j(\theta_i) \leq^* \sigma_j(\theta'_i)) \in T_{TC}(BTV) \times T_{TC}(BTV)$

The algorithm itself is given in seven steps:

1. Repeated application of the rules *reduce1*, *reduce2*, *reduce3*, *erase1*, *erase2*, *erase3*, *swap*, and *adapt* of figure ?? to all elements of Eq . The end configuration Eq' is reached if for each element no rule is applicable.
2. $Eq_1 = \text{Subset of pairs, which consists only of type variables.}$
3. $Eq_2 = Eq' \setminus Eq_1$
4. $Eq_{set} = \{ Eq_1 \} \times (\bigotimes_{(a < \theta') \in Eq_2} \{ a \doteq \theta \mid \theta \in \mathbf{smallerFC}(\theta') \}) \times (\bigotimes_{(a < ? \theta') \in Eq_2} \{ a \doteq \theta \mid \theta \in \mathbf{smallerFC}_?(\theta') \}) \times (\bigotimes_{(\theta < a) \in Eq_2} \{ a \doteq \theta' \mid \theta \leq^* \theta' \}) \times (\bigotimes_{(\theta < ? a) \in Eq_2} \{ a \doteq \theta' \mid \theta \leq_? \theta' \})$
5. Application of the following *subst* rule

$$(\text{subst}) \quad \frac{Eq \cup \{ a \doteq \theta \}}{Eq[a \mapsto \theta] \cup \{ a \doteq \theta \}} \quad a \text{ occurs in } Eq \text{ but not in } \theta$$

for each $\{ a \doteq \theta \}$ in each element of $Eq \in Eq_{set}$. The result is Eq'_{set} .

6. (a) Foreach $Eq' \in Eq'_{set}$ which has changed in the last step start again with the first step.
 - (b) Build the union Eq''_{set} of all results of (a) and $Eq' \in Eq'_{set}$ which has not changed in the last step.
 7. $Uni = \{ Eq'' \mid Eq'' \in Eq''_{set} \text{ is in solved form} \}$
- The result is then a set of unifiers

$$\{ \sigma_1, \dots, \sigma_n \},$$

where for each $Eq_i \in Uni$ holds $\sigma_i = \{ a \mapsto \theta \mid a \doteq \theta \}$.

Example 4.26 In this example we use the standard Java 5.0 types like *Number*, *Integer*, *Vector*, and *Stack* (cp. example 3.14).

As a start configuration we use

$$Eq = \{ \text{Stack}\langle a \rangle < \text{Vector}\langle ?\text{Number} \rangle, \text{AbstractList}\langle ?\text{Integer} \rangle < \text{List}\langle a \rangle \}.$$

In the first step the *reduce1* rule is applied:

$$Eq' = \{ a < ? ?\text{Number}, ?\text{Integer} < ? a \}$$

In the second step nothing happens:

$$Eq_1 = \emptyset$$

The result of the third step is:

$$Eq_2 = Eq'$$

In the fourth step we receive:

$$Eq_{set} = \{ \{ a < ? ?\text{Number}, a \doteq ?\text{Integer} \}, \{ a < ? ?\text{Number}, a \doteq ?\text{Integer} \}, \{ a < ? ?\text{Number}, a \doteq \text{Number} \}, \{ a < ? ?\text{Number}, a \doteq ?\text{Number} \}, \{ a < ? \text{Number}, a \doteq ?\text{Integer} \}, \{ a < ? \text{Number}, a \doteq \text{Integer} \}, \{ a < ? \text{Number}, a \doteq \text{Number} \}, \{ a < ? \text{Number}, a \doteq ?\text{Number} \}, \{ a < ? ?\text{Integer}, a \doteq \text{Integer} \}, \{ a < ? ?\text{Integer}, a \doteq ?\text{Integer} \}, \{ a < ? ?\text{Integer}, a \doteq \text{Number} \}, \{ a < ? ?\text{Integer}, a \doteq ?\text{Number} \}, \{ a < ? \text{Integer}, a \doteq \text{Integer} \}, \{ a < ? \text{Integer}, a \doteq ?\text{Integer} \}, \{ a < ? \text{Integer}, a \doteq \text{Number} \}, \{ a < ? \text{Integer}, a \doteq ?\text{Number} \} \}$$

In the fifth step the rule *subst_eq* is applied:

$$Eq_{set} = \{ \{ \text{Integer} \leq ? \text{Number}, a \doteq \text{Integer} \}, \\ \{ ?\text{Integer} \leq ? \text{Number}, a \doteq ?\text{Integer} \}, \\ \{ \text{Number} \leq ? \text{Number}, a \doteq \text{Number} \}, \\ \{ ?\text{Number} \leq ? \text{Number}, a \doteq ?\text{Number} \}, \\ \{ \text{Integer} \leq ? \text{Integer}, a \doteq \text{Integer} \}, \\ \{ ?\text{Integer} \leq ? \text{Integer}, a \doteq ?\text{Integer} \}, \\ \{ \text{Number} \leq ? \text{Integer}, a \doteq \text{Number} \}, \\ \{ ?\text{Number} \leq ? \text{Integer}, a \doteq ?\text{Number} \}, \\ \{ \text{Integer} \leq ? \text{Integer}, a \doteq \text{Integer} \}, \\ \{ ?\text{Integer} \leq ? \text{Integer}, a \doteq ?\text{Integer} \}, \\ \{ \text{Number} \leq ? \text{Integer}, a \doteq \text{Number} \}, \\ \{ ?\text{Number} \leq ? \text{Integer}, a \doteq ?\text{Number} \} \}$$

With the again application (step six (a)) of the steps one to five we receive the result:

$$Uni = \{ \{ a \doteq \text{Number} \}, \{ a \doteq \text{Integer} \} \}$$

The following example shows, how the problem from [Smo89] is solved.

Example 4.27 Let the following Java 5.0 programm be given:

```
class A { ... }

class B<a> extends A { ... }
```

This means that there is infinite chain in the ordering \leq^* :

$$\dots \leq^* B<B<B<?A>>> \leq^* B<B<?A>> \leq^* B<?A>.$$

In [Smo89] the problem is, that it is not obvious how to solve an equation $a \leq B<?A>$. The solution that it is sufficient to consider the smaller elements in the finite closure with new variables. As all smaller elements are substitutes from these smaller elements in the finite closure, the continuing of the unification algorithm determines the correct substitution. Let the the following start configuration be given:

$$Eq = \{ B<a> \leq B<B<B<B<?A>>>, a \leq B<?A> \}.$$

After the fourth step we receive:

$$Eq_{set} = \{ \{ a \doteq B<B<B<?A>>>, a \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a \doteq B<?A> \} \\ \{ a \doteq B<B<B<?A>>>, a \doteq B<B<a''>> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a \doteq B<B<a''>> \} \}$$

With the fifth step we get

$$Eq'_{set} = \{ \{ a \doteq B<B<B<?A>>>, B<B<B<?A>>> \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<B<a'>>>> \doteq B<?A> \} \\ \{ a \doteq B<B<B<?A>>>, B<B<B<?A>>> \doteq B<B<a''>> \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<B<a'>>>> \doteq B<B<a''>> \} \}$$

Now we have the continue with the first step (step six (a)). With the application of the *reduce3* and the swap rule, we get

$$\{ \{ a \doteq B<B<B<?A>>>, B<B<?A>> \doteq ?A \} \\ \{ a \doteq B<B<B<B<a'>>>>, B<B<B<a'>>> \doteq ?A \} \\ \{ a \doteq B<B<B<?A>>>, a'' \doteq B<?A> \} \\ \{ a \doteq B<B<B<B<a'>>>>, a'' \doteq B<B<a'>> \} \}$$

As the first two set of type term pairs are not in solved form, this means that the set of unifier is given as:

$$\{ \{ a \mapsto B<B<B<?A>>> \}, \{ a \doteq B<B<B<B<a'>>>> \} \}.$$

Now we proof the correctness of the type unification algorithm for type terms with wildcards.

Theorem 4.28 The type unification algorithm determines all general type unifiers for a given set of type term pairs. This means that the algorithm is sound and complete.

The type unification problem for type terms with wildcards is solved by algorithm 4.25. This means that the algorithm 4.25 is sound and complete.

Proof: We do the proof in two steps, as the proof of theorem 4.28. First we prove the *soundness* and then the *completeness*.

Soundness

We have to prove that each calculated result of the algorithm is a unifier of the corresponding input. We do the proof as follows: For an arbitrary result σ we show that if σ is an unifier of the set of type terms after the transformation, then σ is also an unifier of the set of type term pairs before the transformation.

Let

$$\sigma = \{ a_1 \mapsto \theta_1, \dots, a_n \mapsto \theta_n \}$$

be a result of the algorithm.

Step seven: In step seven the type terms pairs are filtered, which are in solved form. This means that σ is also a unifier before applying step seven.

Step six: In step six different sets of type term pairs are united. This means that σ is also an unifier before applying step six.

Step five: The type variables a are substituted by θ . As for the unifier holds $\sigma(a) = \sigma(\theta)$, σ is also a unifier of the set of type terms before the substitution.

Step four: In step four on the one hand type term pairs of the form $(a < \theta')$ respectively $(a <_? \theta')$ are transformed to $(a \doteq \theta)$ for a type term θ with $\theta \leq^* \theta'$.

On the other hand type term pairs of the form $(\bar{\theta} < a)$ respectively $(\bar{\theta} <_? a)$ are transformed to $a \doteq \bar{\theta}'$ for a type term $\bar{\theta}'$ with $\bar{\theta} \leq^* \bar{\theta}'$.

If σ is a unifier of $(a \doteq \theta)$ then σ is also a unifier of $(a < \theta')$ and $(a <_? \theta')$ for $\theta \leq^* \theta'$ respectively $\theta \leq^*_? \theta'$.

If σ is a unifier of $(a \doteq \bar{\theta}')$ then σ is also a unifier of $(\bar{\theta} < a)$ and $(\bar{\theta} <_? a)$ for $\bar{\theta} \leq^* \bar{\theta}'$ respectively $\bar{\theta} \leq^*_? \bar{\theta}'$.

Step two and three: In these steps the pairs are only filtered. This means that the pairs are unchanged. But this means that σ is also a unifier before applying the steps.

Step one: The soundness of the *reduce3* rule, the *erase* rules, and the *swap* rule is obvious.

reduce1, reduce2 rules: If it holds for the unifier σ : $\sigma(\theta_i) = \sigma(\theta'_{\pi(i)})$ for $1 \leq i \leq n$ and

$$C(a_1, \dots, a_n) \leq D(a_{\pi(1)}, \dots, a_{\pi(n)})$$

then follows by definition 3.10

$$\sigma(C(\theta_1, \dots, \theta_n)) \leq^* \sigma(D(\theta'_{\pi(1)}, \dots, \theta'_{\pi(n)})).$$

adapt rule: If it holds for the unifier σ :

$$\sigma(D'(\bar{\theta}'_1, \dots, \bar{\theta}'_m)[a_i \mapsto \lambda'_i \mid 1 \leq i \leq n]) = \sigma(D'(\theta'_1, \dots, \theta'_m)),$$

then follows as it holds $(D(a_1, \dots, a_n) \leq^* D'(\bar{\theta}'_1, \dots, \bar{\theta}'_m))$ by definition 3.10:

$$\sigma(D(\theta_1, \dots, \theta_n)) \leq^* \sigma(D'(\theta'_1, \dots, \theta'_m)).$$

This means that σ is also a unifier of the type term set before this step.

We have proved for all transformations that if a substitution is a unifier of the result the same substitution is also unifier of the input. But this means that the algorithm is sound.

Completeness

For the completeness we have to prove that if there is an unifier of a set of type term pairs then this unifier is also determined by the algorithm. We do the proof as follows: For an arbitrary unifier σ of a given set of type term pairs, we show that in each step of the algorithm after the transformation there is one set of type term pairs, where σ is still an unifier.

Let σ be an unifier of $Eq = \{\theta_1 < \theta'_1, \dots, \theta_n < \theta'_n\}$

Step one: It is obvious that the application of the *erase* rules and the *swap* rule obtain all unifiers.

As it holds $\sigma(D < \theta_1, \dots, \theta_n) = D < \sigma(\theta_1), \dots, \sigma(\theta_n)$ the transformations by the rules *reduce1*, *reduce2*, *reduce3*, and *adapt* obtain also all unifiers.

This means that after step one σ is still an unifier of the transformed set of type term pairs Eq' .

Step two, three, and four: The pairs of type terms which are filtered in step two are unchanged in step four, such these pairs are not needed to be considered.

Now we have to consider four different cases in step four of type term pairs, which are filtered in step three:

$(a < \theta') \in Eq_2$: This means $\sigma(a) \leq^* \sigma(\theta')$. From this follows by definition ???3.10???

that there is a $(\bar{\theta} \leq^* \bar{\theta}') \in \mathbf{FC}(\leq)$ and a substitution σ' with $\sigma'(\bar{\theta}') = \theta'$ and $\sigma'(\bar{\theta}) \in \mathbf{smallerFC}(\sigma'(\bar{\theta}'))$ such that $\sigma(a) = \sigma(\sigma'(\bar{\theta}))$.

This means for the pair $(a < \theta') \in Eq'$, where σ is an unifier, there is after the transformation an equations set in Eq_{set} with a pair $(a \doteq \sigma'(\bar{\theta}))$, where σ is still an unifier.

$(a <_? \theta') \in Eq_2$: This case is analogous to the first case.

$(\theta < a) \in Eq_2$: This means $\sigma(\theta) \leq^* \sigma(a)$. From this follows by definition ???3.10???

that there is a type term θ' with $\theta \leq^* \theta'$ and $\sigma(a) = \sigma(\theta')$.

This means for the pair $(\theta < a) \in Eq'$, where σ is an unifier, there is after the transformation an equations set in Eq_{set} with a pair $(a \doteq \theta')$, where σ is still an unifier.

$(\theta <_? a) \in Eq_2$: This case is analogous to the previous case.

As in step four the cartesian product of the particular results is built there is one set of type term pairs of Eq_{set} , where σ is still an unifier.

Step five: If σ is an unifier of $\{a \doteq \theta\}$, then it holds $\sigma(a[a \mapsto \theta]) = \sigma(\theta)$. This means that for all pairs $(\bar{\theta} < \bar{\theta}') \in Eq$, $(\bar{\theta} <_? \bar{\theta}') \in Eq$, respectively $(\bar{\theta} \doteq \bar{\theta}') \in Eq$ with $\sigma(\bar{\theta}) \leq^* \sigma(\bar{\theta}')$ respectively $\sigma(\bar{\theta}) = \sigma(\bar{\theta}')$ it holds $\sigma(\bar{\theta}[a \mapsto \theta]) \leq^* \sigma(\bar{\theta}[a \mapsto \theta'])$ respectively $\sigma(\bar{\theta}[a \mapsto \theta]) = \sigma(\bar{\theta}[a \mapsto \theta'])$.

This means that after step five σ is still an unifier of the transformed set of type term pairs Eq' .

Step six: As no pair of type terms is transformed, the unifiers are maintained.

Step seven: In step seven the pairs of type terms which are in solved form are filtered. All other sets of pairs of type terms have no unifier, thus all unifier are obtained.

This means that for an arbitrary given unifier σ of $\{\theta_1 < \theta'_1, \dots, \theta_n < \theta'_n\}$, σ is calculated by the algorithm. This means that the algorithm is complete. ■

4.2.3 Unified Minimal Upper Bound

As an extension of the type unification algorithm \mathbf{TUnify}_{\leq^*} we consider now the problem, determining a substitution σ and a set of type terms Mub for two given type terms θ_1, θ_2 such that $\bar{\theta} \in Mub$ is a minimal upper bound of $\sigma(\theta_1)$ and $\sigma(\theta_2)$.

Algorithm 4.29 The minimal upper bound type unification algorithm \mathbf{TUnify}_{Mub} is given as follows

Input: A pair (θ_1, θ_2)

Precondition: $\theta_1, \theta_2 \in T_{TC}(BTV)$.

Output: A pair (Mub, σ) with $Mub = \{\bar{\theta}_1, \dots, \bar{\theta}_m\}$ a set suprema and a most general unifier σ .

Postcondition: For all $1 \leq j \leq m$ holds $\bar{\theta}_j$ is a minimal upper bounds of $\sigma(\theta_1)$ and $\sigma(\theta_2)$.

The algorithm itself is given in three steps:

1. If θ_1 or θ_2 is a type variable. The result is given as $(\{\theta_2\}, [\theta_1 \mapsto \theta_2])$, respectively $(\{\theta_1\}, [\theta_2 \mapsto \theta_1])$.
In the other cases let $\theta_1 = D' \langle \theta'_1, \dots, \theta'_n \rangle$ and $\theta_2 = D'' \langle \theta''_1, \dots, \theta''_m \rangle$:
Then, let the set of minimal upper bounds of $D' \langle a_1, \dots, a_n \rangle$ and $D'' \langle b_1, \dots, b_m \rangle$ in $(T_{TC}(BTV), \mathbf{FC}(\leq))$ be given as $\mathbf{MUB}(D' \langle a_1, \dots, a_n \rangle, D'' \langle b_1, \dots, b_m \rangle)$.
2. Let $\sigma = \mathbf{TUnify}_{\leq^*}(a_1 \doteq \theta'_1, \dots, a_n \doteq \theta'_n, b_1 \doteq \theta''_1, \dots, b_m \doteq \theta''_m)$ if it exists.
3. The result is then given as

$$(\{\sigma(\bar{\theta}) \mid \bar{\theta} \in \mathbf{MUB}(D' \langle a_1, \dots, a_n \rangle, D'' \langle b_1, \dots, b_m \rangle)\}, \sigma).$$

Now, we give an example for the algorithm \mathbf{TUnify}_{mub} .

Example 4.30 We consider again the type term ordering from example 3.16, *DARGESTELLT* in figure 3.4.

We apply the algorithm to the pair

$$(\mathbf{Vector} \langle \mathbf{Vector} \langle \mathbf{b} \rangle \rangle, \mathbf{PriorityQueue} \langle \mathbf{c} \rangle).$$

1. We construct $\mathbf{MUB}(\mathbf{Vector} \langle \mathbf{a} \rangle, \mathbf{PriorityQueue} \langle \mathbf{a} \rangle) = \{\mathbf{Collection} \langle \mathbf{a} \rangle, \mathbf{Serializable}\}$.
2. $\sigma = \mathbf{TUnify}_{\leq^*}(\mathbf{a} \doteq \mathbf{Vector} \langle \mathbf{b} \rangle, \mathbf{a} \doteq \mathbf{c}) = [\mathbf{a} \mapsto \mathbf{Vector} \langle \mathbf{b} \rangle, \mathbf{c} \mapsto \mathbf{Vector} \langle \mathbf{b} \rangle]$.
3. The result is then given as

$$(\{\mathbf{Collection} \langle \mathbf{Vector} \langle \mathbf{b} \rangle \rangle, \mathbf{Serializable}\}, \sigma).$$

Correctness of \mathbf{TUnify}_{mub}

It is obvious that \mathbf{TUnify}_{mub} is correct if θ_1 or θ_2 is a type variable.

Let us consider the other case:

Let $\bar{D} \langle \bar{\tau}_1, \dots, \bar{\tau}_k \rangle$ be a minimal upper bound of $D' \langle \tau'_1, \dots, \tau'_n \rangle$ and $D'' \langle \tau''_1, \dots, \tau''_m \rangle$ in $(T_{TC}(BTV), \leq^*)$. Then, from definition 3.10 follows, that there are $\bar{\tau}'_1, \dots, \bar{\tau}'_k$ with $\bar{D} \langle \bar{\tau}'_1, \dots, \bar{\tau}'_k \rangle$ is a minimal upper bound of $D' \langle a_1, \dots, a_n \rangle$ and $D'' \langle b_1, \dots, b_m \rangle$ ($a_1, \dots, a_n, b_1, \dots, b_m \in TV$) in $(T_{TC}(BTV), \mathbf{FC}(\leq))$ and

$$\bar{D} \langle \bar{\tau}_1, \dots, \bar{\tau}_k \rangle = \bar{D} \langle \bar{\tau}'_1, \dots, \bar{\tau}'_k \rangle [a_1 \mapsto \tau'_1, \dots, a_n \mapsto \tau'_n, b_1 \mapsto \tau''_1, \dots, b_m \mapsto \tau''_m].$$

From this follows that in \mathbf{TUnify}_{mub} for $\theta_1 = D' \langle \theta'_1, \dots, \theta'_n \rangle$ and $\theta_2 = D'' \langle \theta''_1, \dots, \theta''_m \rangle$ first all minimal upper bounds $\bar{D} \langle \bar{\theta}'_1, \dots, \bar{\theta}'_k \rangle$ of $D' \langle a_1, \dots, a_n \rangle$ and $D'' \langle b_1, \dots, b_m \rangle$ in $(T_{TC}(BTV), \mathbf{FC}(\leq))$ must be determined. This is done in step 1. Then for each pair $a_i = b_j$ the subterms θ'_i and θ''_j of θ_1 and θ_2 must be unified. This is done in step 2 by the unification algorithm \mathbf{TUnify}_{\leq^*} with the input pairs $a_i \doteq \theta'_i$ for $1 \leq i \leq n$ and $b_j \doteq \theta''_j$ for $1 \leq j \leq m$. Finally, in step 3 the unifier is applied to all minimal upper bounds of $D' \langle a_1, \dots, a_n \rangle$ and $D'' \langle b_1, \dots, b_m \rangle$. The result is then a minimal upper bound of θ_1 and θ_2 in $(T_{TC}(BTV), \leq^*)$.

4.3 Implementation

- Wie detailliert soll eine Implementierung beschrieben werden?

$$\begin{aligned}
(\text{reduce1}) \quad & \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \leq D \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq \theta'_1, \dots, \theta_{\pi(n)} \leq \theta'_n \}} \\
& \text{where} \\
& - C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& - \{ a_1, \dots, a_n \} \subseteq TV \\
& - \pi \text{ is a permutation} \\
\\
(\text{reduceExt}) \quad & \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq Y \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq \theta'_1, \dots, \theta_{\pi(n)} \leq \theta'_n \}} \\
& \text{where} \\
& - X \langle a_1, \dots, a_n \rangle \leq Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& - \{ a_1, \dots, a_n \} \subseteq TV \\
& - \pi \text{ is a permutation} \\
\\
(\text{reduceSup}) \quad & \text{HIER WEITERMACHEN, UNKLAR WIE SUPER AUF SUBTERMEN FORTGESETZT WIRD} \\
& \text{where} \\
& - X \langle a_1, \dots, a_n \rangle \leq Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& - \{ a_1, \dots, a_n \} \subseteq TV \\
& - \pi \text{ is a permutation} \\
\\
(\text{reduce3}) \quad & \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \doteq C \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_1 \doteq \theta'_1, \dots, \theta_n \doteq \theta'_n \}} \\
\\
(\text{erase1}) \quad & \frac{Eq \cup \{ \theta \leq \theta' \}}{Eq} \quad \theta \leq^* \theta' \\
\\
(\text{erase2}) \quad & \frac{Eq \cup \{ \theta \leq \theta' \}}{Eq} \quad \theta \leq^* \theta' \\
\\
(\text{erase3}) \quad & \frac{Eq \cup \{ \theta \doteq \theta' \}}{Eq} \quad \theta = \theta' \\
\\
(\text{swap}) \quad & \frac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}} \quad \theta \notin TV, a \in TV \\
\\
(\text{adapt}) \quad & \frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto \theta_i \mid 1 \leq i \leq n] \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}
\end{aligned}$$

where there are $\bar{\theta}'_1, \dots, \bar{\theta}'_m$ with

- $(D \langle a_1, \dots, a_n \rangle \leq^* D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle) \in \mathbf{FC}(\leq)$

Figure 4.3: Java 5.0 type unification with wildcards

$$\begin{aligned}
(\text{reduce1}) \quad & \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \leq D \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq \theta'_1, \dots, \theta_{\pi(n)} \leq \theta'_n \}} \\
& \text{where} \\
& - C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& - \{ a_1, \dots, a_n \} \subseteq TV \\
& - \pi \text{ is a permutation} \\
\\
(\text{reduce2}) \quad & \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq Y \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq \theta'_1, \dots, \theta_{\pi(n)} \leq \theta'_n \}} \\
& \text{where} \\
& - X \langle a_1, \dots, a_n \rangle \leq Y \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle \\
& - \{ a_1, \dots, a_n \} \subseteq TV \\
& - \pi \text{ is a permutation} \\
\\
(\text{reduce3}) \quad & \frac{Eq \cup \{ X \langle \theta_1, \dots, \theta_n \rangle \leq Y \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \leq \theta'_1, \dots, \theta_{\pi(n)} \leq \theta'_n \}} \\
\\
(\text{erase1}) \quad & \frac{Eq \cup \{ \theta \leq \theta' \}}{Eq} \quad \theta \leq^* \theta' \\
\\
(\text{erase2}) \quad & \frac{Eq \cup \{ \bar{\theta} \leq \bar{\theta}' \}}{Eq} \quad \theta \leq^* \theta' \\
\\
(\text{erase3}) \quad & \frac{Eq \cup \{ \theta \doteq \theta' \}}{Eq} \quad \theta = \theta' \\
\\
(\text{swap}) \quad & \frac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}} \quad \theta \notin TV, a \in TV \\
\\
(\text{adapt}) \quad & \frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto \theta_i \mid 1 \leq i \leq n] \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}
\end{aligned}$$

where there are $\bar{\theta}'_1, \dots, \bar{\theta}'_m$ with

- $(D \langle a_1, \dots, a_n \rangle \leq^* D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle) \in \mathbf{FC}(\leq)$

Figure 4.4: Java 5.0 type unification with wildcards

Chapter 5

Type Inference

5.1 Introduction to Type Inference

[Mit84] [Rey85] [Car88]

We will now introduce some type inference systems and its type reconstruction algorithm. We will start with the Hindley–Milner type system [Mil78, DM82]. Then, we show type systems, that extend the Hindley–Milner type system by subtyping and overloading in different variants. An overview of type systems is given in [CW85].

In well-known imperative programming languages like PASCAL [JW74, JWM85], C [KR88], or Modula-2 [Wir88] and in object-oriented languages like C++ [Str91], Eiffel [Mey92], or Java [GJS96] with static typing, declaration of a type for any variable and any function is mandatory before it is possible to use it.

Since the definition of ML (later SML [MTH90, Mil97]) there is a functional programming language which is static typed, but does not require that the types are explicitly declared. For ML there is a type inference system which specifies types for a given function declarations. This type inference system is derived from a system which is formulated in combinatory logic by Hindley [Hin69]. Therefore, this type system is called *the Hindley–Milner type system*. In [Mil78] Robin Milner defines the rules for the type inference system, but the problem if there is an algorithm which reconstructs the type for a given function declaration is in this paper open. In [DM82] Luis Damas and Robin Milner presents the algorithm \mathcal{W} , which satisfies the type inference rules. Furthermore, they proved that the types reconstructed by \mathcal{W} are the most general types of the function declaration, which means that all other types of the function declaration are instances of these types.

Another algorithm which satisfies the type inference system of Milner is called \mathcal{M} and is presented by Lee and Yi in [LY98]. \mathcal{M} stops earlier than \mathcal{W} if a type error occurs and the error information is more detailed. Therefore this algorithm is often used for the implementation of functional programming languages.

Mycroft [Myc84] was the first, who proposed an extension of ML type discipline that allows *polymorphic recursion*. Polymorphic recursion means that the declaration of a function f can contain recursive calls of f with different instances of the type of f . Therefore this type system is often called *Milner–Mycroft*. Unfortunately, the type reconstruction problem

for the Milner–Mycroft type system is undecidable, which is proved by Henglein [Hen88, Hen93]. He proves that the Milner–Mycroft type reconstruction problem is equivalent to the semi-unification problem, which is given as the problem of solving subsumption inequations between first-order terms. The semi-unification problem is undecidable, which is proved in [KTU90]. The functional programming languages Hope [BMS80], MirandaTM [Tur86], and Haskell 98 [eAB⁺02] allow polymorphic recursive function declaration at top-level, but they require explicit type declarations for such functions.

Kfoury, Tiuryn, and Urzyczyn [KTU93] extends the Hindley–Milner type system by *polymorphic abstraction*, which means that for a λ -bound variable occurrences of different instances are allowed in the λ -term. They proved that the type reconstruction problem of Milner–Mycroft, the problem of type reconstruction for the Hindley–Milner type system extended by polymorphic abstraction, and the semi-unification problem are equivalent. This means that also the problem of type reconstruction for the Hindley–Milner type system extended by polymorphic abstraction is undecidable.

Another extension of the Hindley–Milner type system is the introducing of overloading and subtyping. Fuh and Mishra [FM88, FM90] allow to declare a subtype relation on constant type terms. The type reconstruction algorithm determines additionally a set of coercion constraints. Finally, these coercion constraints are handled again by two algorithms, where the first one determines the instances for the type variables and the second one checks the consistence of the set of coercion constraints.

Mitchell [Mit91] considers also subtypes in the λ -calculus [Bar84]. Beside the syntactic type inference he considers the semantic soundness of two different type inference systems. Furthermore, he gives two type reconstruction algorithms for the λ -calculus with coercions. The algorithms compute in addition to a type and a substitution, as usual, a minimal set of coercions required to make the term typable. The first one allows arbitrary coercions between types, while the second one allows only atomic coercions, i.e. coercions between atomic terms (type variables and constants).

Kaes considers in [Kae88, Kae92] type inference systems where Hindley–Milner is extended by overloading and subtyping. In [Kae88] he considers an extension of the Hindley–Milner type system allowing parametric overloading, i.e. overloading is restricted to function symbols and the result type of a function application $f(t_1, \dots, t_n)$ is uniquely determined by the outermost type constructors of the types of the arguments. His approach to overloading is based on a restriction of generalized types. For example the typing $f : \forall \alpha. \alpha \rightarrow \alpha$ is restricted such that only some types are allowed as instances of α . In this case f is overloaded by these types. For this extension he gives a type inference system and a corresponding type reconstruction algorithm.

In [Kae92] he unites overloading and subtyping in a way such that the restrictions which hold for the overloaded function symbols are considered as additional constraints in the set of coercion constraints.

In both papers the overloading is restricted to function symbols. Some restrictions on overloading are necessary, as type inference without any restrictions is undecidable. This is proved by Leivant [Lei83].

In [AW93] a general algorithm for solving systems of inclusion constraints over type ex-

pressions is presented. The algorithm works incrementally transforming a system of constraints until the system is discovered to be inconsistent (i.e. has no solution) or to be inductive, which means that the system is equivalent to a system of equations, which have always a solution. The type language includes a least type 0, a universal type 1, intersection and union types, function types, constructor types, and recursive types. The intersection type are restricted to so-called *liberal* intersection types, which means that for example overloaded function symbols are not allowed. In comparison our type system is restricted to intersection types of overloaded function symbols. In the system of [AW93] every λ -term typeable in the Hindley–Milner type system has all of its Hindley–Milner types, which means that this type system is a generalization of the Hindley–Milner type system.

Another form of overloading is given in Haskell 98 [eAB⁺02]. In Haskell types classes and its instances (cf. [WB89]) are introduced. The type class declares at least a signature of function symbols (furthermore it is possible to define some equations, which must be fulfilled by the instances). The corresponding instances define the interpretation of the function symbols. Additionally, it is possible to declare contexts for type classes and the instances. For example `class $\chi_1(a) \Rightarrow \chi_2(a)$ where ...` or `instance $\chi_1(a) \Rightarrow \chi_2(a)$ where ...` declares the class χ_2 or the instance χ_2 , respectively, in the context χ_1 . This means in the view of object-oriented programming, that the type classes or the instances, respectively, inherit from its contexts.

The possibility to declare different instances of one type class implies that all function symbols of such a type class are overloaded.

In [WB89] a type inference system is given, which is more powerful than the type system of Haskell. Volpano and Smith [VS91] showed that this type system is undecidable.

The inheritance ordering of type classes in Haskell is considered in [NS91] as an ordered set of sorts of a (non-polymorphic) order-sorted signature (cf. section ??). The function symbols of these order-sorted signatures are given as the type constructors, for which instances of the type classes are declared. The arities are given as the tuples of class names of contexts and the coarities are defined as the names of the classes which are instanced. For example, if an instance of the predefined class `Eq` is declared for the type constructor `Pair` by `instance (Eq a, Eq b) => Eq (Pair(a, b)) where ...` then `Pair` gets the rank `Pair: (Eq, Eq) -> Eq`. For the type inference, the type variables additionally typed by the class names. The Robinson unification [Rob65] in the algorithm \mathcal{W} of Milner is substituted by an order-sorted unification algorithm (cf. figure ??). This approach is similar to the approach of Kaes, where overloading bases on the restriction of generalized types.

In the functional programming language Gofer [Jon94] there are another two extensions. While in Haskell only parameterless type classes are allowed, Gofer has parameters for type classes. It is possible that these parameters stand not only for type terms, but also for type constructors (higher-order polymorphism). In [CHO92] there is a type inference system and a type reconstruction algorithm for parameterized type classes, which is based also on the Hindley–Milner type reconstruction algorithm. Only the unification algorithm is substituted by a *context-preserving* unification algorithm.

The constructor classes (type classes with parameters for type constructors) are detailed considered in [Jon93]. Jones gives a lot of examples for constructor classes and a type reconstruction algorithm.

Weber [Web93] considered the integration of coercions (in the sense of automatic type transformations) and the concept of type classes. With these ideas he describes a type system for a subset of Axiom [JS92] (Scratchpad [Jen84]) and gives a type reconstruction algorithm.

Odersky, Wadler, and Wehr [OWW95] considers overloading, such that a program possesses a meaning that can be determined independently of its type, which is impossible in Haskell. For this they make a simple restriction to type classes of Haskell: In a type class over `a` each overloaded function symbol must have a type of the form `a -> t`. Furthermore, they generalize the type classes such that polymorphic records can be handled. For this changed language they line out a type reconstruction algorithm, where again the Robinson unification is substituted in the Hindley–Milner algorithm by a constrained unification algorithm.

A complete different approach of subtyping and overloading gives Freeman [Fre94]. He considers refinement types. Refinement types require two type systems. On the one hand a simple basic type system is needed (Hindley–Milner type system) and on the other hand a more expressive type system, which allows overloading and subtyping. The second type system refines the first one. It is possible for the user to declare monomorphic as well as polymorphic refinements.

A generalization of the these type inference systems with constraints is given in [OSW99]. They present a general framework $\mathbf{HM}(X)$, which is the Hindley–Milner (HM) type system, with a constraint system X . The Hindley–Milner type system is extended on the one hand, as types are members over an arbitrary term algebra, which means that besides \rightarrow other type constructors are admissible. On the other hand type schemes are extended, such that they can contain a constraint component $C: \forall \alpha. C \Rightarrow \tau$, which restricts the types, that can be substituted for the type variable α . They give a type inference algorithm, which computes under sufficient conditions on X a principal type for each expression. As instances of their framework they give the standard Herbrand constraint system, which represents the original Hindley–Milner type inference algorithm, the constraint based systems, given in [AW93], `??EST95a???` WAS HAT DAS MIT EST95 ZU TUN???, and an example from [Oho95] with polymorphic records.

Last, we will consider two papers about *local type inference*. In [PT00] the item is introduced by Pierce and Turner. In [OZZ01] the concept is extended by Odersky, C. Zenger, and M. Zenger to *colored local type inference*. These ideas are used for the yet implemented aspects of type inference in Java 5.0 (cp. section 2.1.1).

The type system of local type inference bases on F_{\leq} , the second-order λ -calculus with subtyping e.g. [CW85]. It allows omitting some type annotations with two techniques. On the one hand type arguments in applications of polymorphic functions are inferred, which means, that for example the type of the application of the identity function $id = \lambda x.x : \forall \alpha. \alpha \rightarrow \alpha$ to a number `id1` infers the type `int`. The type annotations are determined by a local constraint solver without any long-distance constraints as unification

variables. On the other hand known type informations are propagated down the syntax tree to infer types of bounded parameters of anonymous functions and type annotations on local variable bindings. For example a function f with the type $(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$ is applied: $f(\lambda x : \text{int}. x + 1)$ the type annotation of x can be omitted.

If the function f would have the type $\forall \alpha. (\text{Int} \rightarrow \alpha) \rightarrow \alpha$ the type annotation of x in $f(\lambda x. x + 1)$ would not be able to determine by downward propagating. It is necessary to propagate the propagate the for α instanced type int upward. In [OZZ01] the propagation direction is expressed by coloring the types.

As application of the colored local type inference, in this paper pattern matching is presented. In PIZZA pattern matching is implemented as PIZZA allows algebraic datatypes. ???In GJ pattern matching is implemented by visitor patterns [GHJV94]???

Im comparison to our approach these approaches of local type inference consider only restricted type inference. For example in [PT00] it is considered *how much type inference is enough?* From a statistical analysis they derive the most important cases of type inference and assert these as the properties of local type inference. For our approach a type unification algorithm (cp. chapter 4) is necessary. ???This approach includes the cases of local type inference.???

5.1.1 Cecil

First, we consider [Lit98], where a static type system for object-oriented languages, which provides type checking and local type inference, similar as in [PT00], is presented. They implemented their type system in the language Cecil. The type system supports *bounded parametric polymorphism*, *F-bounded polymorphism*, *inheritance*, *subtyping*, *overloading*, *method constraints*, and *first-class functions*. In the paper type-checking/-inference rules are presented.

5.2 Type inference in object oriented languages

In this section we present existing approaches of type inference in object-oriented languages. In section 5.7 we compare these approaches to our approach, which is introduced in section 5.4.

[CCH⁺89, CHC90] F-bounded types, subtypes and inheritance

There are basically two different approaches of type-inference in object-oriented languages. On the one hand types are inferred as precise as possible. This is done for optimizations in the compiler construction. This approach is followed by Palsberg and Schwartzbach and all following papers (cp. figure 5.1). These approaches we will consider in section 5.2.1.

Another approach is given following the Hindley-Milner approach. Eifrig, Smith, and Trifonov describes an approach for object-oriented languages following the λ -calculus. Furthermore the language *ocaml*, which is object-oriented extension of *caml* has a type-inference system. In these approaches principle types respectively most general types are

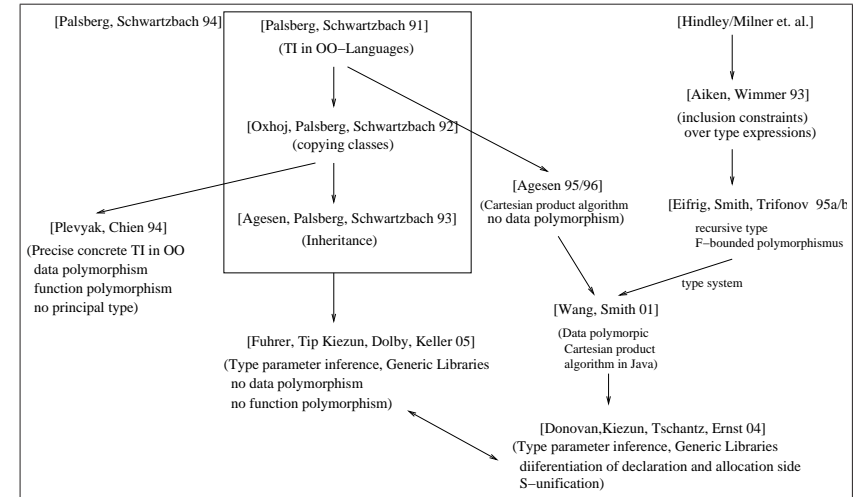


Figure 5.1: Overview of related work

inferred. This is done from the point of writing application code, especially code re-use. We will consider these approaches in section 5.2.3.

A third class of type-inference algorithms considers the automatic inference of type parameters of raw declared types. This approach is used for refactoring Java programmes by introducing type parameters and removing redundant type-casts. This approach is following the idea of Palsberg and Schwartzbach. We consider this approach in section 5.2.2.

5.2.1 Flow analysis approaches and constraint solving

The idea of viewing type inference as a problem of solving constraints has been explored in [Wan87] for functional languages and in [Sch91] for imperative languages.

IST [PS92, KPS94] RELEVANT???

In [PS91] a type inference system is given for a language which is oriented at *Smalltalk* [DHH81, GR83]. The idea in this system is to build a uniform set of type constraints, which is solved by a fixed point derivation. Their type system is defined such that types are finite sets of classes and subtyping is a set inclusion.

The algorithm itself has four steps. First all classes are expanded, which means among other things copying the text of a class to its subclasses. In the second step a trace graph is constructed. The nodes are obtained from the various methods implemented in the program and the edges will reflect the possible connections between a message send and a method that may implement it. In the third step from this trace graph a set of type

constraints are extracted. Finally, in the fourth step, the least solution of the set of type constraints is computed, if it exists. Afterwards, the solution is mapped back onto the program.

In [OPS92] the algorithm of [PS91] is improved on the one hand by handling container classes (collection classes) as **Lists**. This is done by copying the class code. On the other hand the implementation is improved including new techniques.

In [APS93] the algorithm is applied to **Self** [US91]. In this approach the algorithm is extended to handle dynamic and multiple inheritance.

All these approaches are summerized in [PS94]. Here, object-oriented type systems are described in detail. The book uses a language named **BOPL**. **BOPL** is given in differents variants and contains common features of **Simula**, **Smalltalk**, **C++**, and **Eiffel**.

Concurrent Aggregates

In [PC94] type inference of the language **Concurrent Aggregates (CA)** [Chi93, CKPZ93] is described. **CA** is a dynamically typed cuncurrent object-oriented language with *single inheritance*, *first class selectors*, and *continuations*. The language allows *data polymorphism* and *functional polymorphism*.

They present an incremental constraint-based type inference algorithm. The algorithm contains a data flow analysis, which produce an interprocedural call graph, in which function/method calls and container creations are splittedA to eliminate function and data polymorphism. The analysis is efficient as they use *entry sets* and *container sets*, which collect similar flow histories together, reducing the costs of the data flow analysis. In contrast in [OPS92] for each syntactic **new** a copy of the corresponding class is created. A drawback of this approach is that iterations a necessary. This is caused by the possibility that modifications (splittings) of the developing constraint network could lead to solutions inconsistent to the network. Therefore in this case a further iteration is necessary.

This lack has been addressed by the cartesian product algorithm.

The Cartesian Product Algorithm (CPA)

The CPA [Age95] uses for each method a *dictionary* of templates. Given a method call **rcvr.m(arg₁ ldo arg_n)** with sets of known types for **m** and **arg_i**, respectively. Then for each tuple of the cartesian product of the known types a new template is created, if it is not in the dictionary. In the other case the corresponding template already exists and no new template has be created. In contrast to the iterative algorithm the CPA needs no iteration after adding new templates. This is caused by the monotonicity of the cartesian product, which means if one set of types grows also the coresponding cartesian product grows.

In the approaches of [PS91, OPS92, PS94] inheritance is handled by copying methods down into classes inheriting them. The CPA approach removes the need for expansion, as it always creates one template for each possible type. i.e. CPA always analyses a method in the context of one class at a time.

This algorithm is more efficient than the iterative approach and is in some cases also more precise. The loss of the algorithm is handling data polymorphism. It losses precise for

programs with container classes as **Lists**.

Data-Polymorphic CPA (DCPA)

The DCPA [WS01] addresses the problem of precise types of instances of container classes, like **java.util.Vector** in **Java**.

They define a framework of object-oriented constraint inference. The type bases on that of [AW93] und is close to that of [EST95b] (cp. section 5.2.3). The constraints generated first in an initial phase for the expressions. Then a closure is built.

They show that the result corresponds to a CPA result.

Then, in the data polymorphic analysis the method invocations and the object creations via **new** are devided in CPA-safe and CPA-unsafe ones. CPA-safe means thta no data polymorphism occurs and CPA-unsafe menas that data polymorphism occurs. For CPA-safe invocations templates (contours) are created as in the CPA algorithm. For the CPA-unsafe ones additional contours are created.

Finally the DCPA closure algorithm solves the constraints.

5.2.2 Converting raw types to parametrized types

Now we consider two approaches, where type inference supports replacing raw types of **Java** programs with parametrized type of **Java 5.0** programs. Two similar approaches are given [DKTE04, FTK⁺05]. The algorithms in both approaches rewrites raw references to generic class with parametrized references. This is done in the declaration sides as well as in the allocation sides. Furthermore redundant type-casts are removed.

The following small example will explain, what is done.

Using raw references

```
Vector v = new Vector ();
v.addElement(5);
Integer i = (Integer)v.get(0)
```

Result of rewriting

```
Vector<Integer> v = new Vector<Integer> ();
v.addElement(5);
Integer i = <Integer>v.get(0)
```

In this small example first in declaration side of the assignment **Vector v = new Vector()** the parameter **Integer** is added. Then the parameter is also added in the allocation side. Finally in the third line the redundant type-cast is removed.

In [DKTE04] there is differed between references to raw types in the allocation sides and in the declaration sides. First the type parameters of the allocation sides are determined by an algorithm consisting of a context-sensitive pointer analysis, a S-unification (subtype unification), and a resolution of parametric types.

The pointer analysis algorithm is similar to the algorithm of [WS01]. The contour selection function is reduced. Only those parameter positions and **this**, which contains a type variable are analyzed polymorphically.

In a second step the type parameters of the raw references to generic classes in the declaration sides are determined. This is done by creating a type constraint system, which is derived by typing rules of **Java** expressions and statements. Finally the constraint system is solved. The constraint solver needs possibly backtracking.

The approach of [DKTE04] allows no modifications of method signatures and fields. It is not possible to infer parameters of references to classes defined in application code.

The algorithm in [FTK⁺05] differs not between types in the allocation sides and in the declaration sides. This algorithm builds a constraint system on the base of the constraint system in [PS94]. The constraint system is extended to generic types. Then the constraint solving is done in a standard iterative fashion. The constraint solver goes without backtracking. In contrast to [DKTE04] this approach modifies method signatures and the algorithm is capable to infer parameters of types, which corresponding classes are defined in application code.

In both approaches the constraint systems are typically underconstrained. This means that more than one typing of the parameters is correct. The goal of both approaches is to get a solution such that most type-casts could be removed.

5.2.3 Milner-like approaches

In this section we consider three approach following the Hindley–Milner approach. First we consider two approaches, where the λ -calculus is enriched by object-oriented features, the languages I-SOOP and I-LOOP. Then we consider `ocaml`, the object-oriented extension of `caml`.

I-SOOP

In [EST95b] a type system with recursively constrained types are introduced. The base of this system is similar to the system of [AW93]. Types are of the form $\tau \setminus C$, where τ is a conventional type and C is a set of type constraints of the form $\tau_1 \leq \tau_2$. The type system allows multiple lower and multiple upper bounds, which can be understood as a restricted form of intersection and union types. During the type inference process the constraints accumulated such that the results have a small set of constraints. In contrast to [AW93, Kae92, PS94] in this approach a constraint system is considered as consistent if it contains no *obvious* contradiction as `Nat` \leq `Bool`. The soundness of the type system is proved without showing that a, in the sense of the system, *consistent* constraint system have solutions.

In this approach the type inference algorithm is given for the language I-SOOP (Inference Semantics of OOP). I-SOOP is not an object-oriented language. But it is possible that the object-oriented language OOP is encoded in I-SOOP, as I-SOOP records, record subtyping, and a notion of state. The encoding of the object-oriented features is done by the definition of a collection of simple macros, where the *class* definition is described by a λ -expression as a nested `let`-expression and the `new`-operator is described by a λ -expression with a call-by-value `Y`-combinator.

This approach is powerful. The main problems of this type system are that the types are large and less easily readable for programmers.

I-LOOP

In [EST95a] the language I-LOOP (Implicitely typed Little Object Oriented Programming Language), which contains object-oriented features itself, is described. It includes notions of class, multiple inheritance, object creation (`new`), and single dispatch of messages. The language contains λ -terms, which means that the type system contains the function constructor \rightarrow . Types, which calculated during the type-inference are also recursively constrained types as in I-SOOP. The semantics of I-SOOP is given by a translation to I-SOOP. The type-soundness of I-LOOP is proved by the translation to I-SOOP, as the type-soundness of I-SOOP is proved in [EST95b].

In both approaches for the type-inference of I-SOOP and I-LOOP, respectively no principle type property is given. They proved only that the type-inference is complete. This means that a program, which is typable under the general typing has a type inferred by the type-inference algorithm.

OCAML

The programming language OCAML[CMP00] is a functional programming language, which is developed from `Standard ML` [Mil97], where the syntax is changed, but the type system is largely adopted. Then, the language is extended by object oriented features. The Hindley–Milner type system [Hin69, DM82] is extended to to these object oriented features.

Objects in OCAML

classes: The declaration of classes allows instance variables and methods.

```
class point =
object
  val mutable x = 0
  method get_x = x
  method move d = x <- x + d
end;;
```

As shown in the example the instance variables are changable if they are declared `mutable`.

inheritance: The inheritance is declared by the keyword `inherit`.

```
class colored_point =
object
  inherit point
  ...
end;;
```

class parameters: It is also possible to declare class parameters.

```

class ['a] vector a =
object
  val mutable l = Array.make 0 (a : 'a);
  method add e = l <- Array.append l (Array.make 1 e)
  method size = Array.length(l)
  method elementAt i = l.[i]
end;;

```

Moreover, this example shows that it is allowed to declare an uninitialized instance variable. Hence, this means that we must initialize the list `l` of vector elements as an empty array of elements with an arbitrary type. But this leads to the property that the class must be additionally parameterized of this initial value `a`. The type of the expression is then

```

class ['a] vector :
'a ->
object
  val mutable l : 'a array
  method add : 'a -> unit
  method elementAt : int -> 'a
  method size : int
end

```

parameter constraints: Similar as the F-bounded parameters in Java 5.0 the parameters can be constrained.

```

class ['a] circle (c : 'a) =
object
  constraint 'a = #point
  val mutable center = c
  method set_center c = center <- c
end;;

```

In this example the initial center `c` of the circle must be a subtype of `point`.

further features: OCAML allows class interfaces, multiple inheritance, virtual, and private methods.

5.2.4 ImplicitPoly–Tiger

Appel describes in [App02] a type reconstruction algorithm for his language **ImplicitPoly–Tiger**. **ImplicitPoly–Tiger** is an imperative language with polymorphic recursive record types. The base of his approach similar as our approach is the Hindley–Milner type reconstruction algorithm. The algorithm is extended to the imperative language constructs.

5.3 Non-explicitly typed Generic Java Programs

5.4 The Type System of Java 5.0

<i>Source</i>	$:=$	$(\text{class} \text{interface})^*$
<i>class</i>	$:=$	Class(<i>type</i> , [extends(<i>type</i>),] [implements(<i>type</i> *),] <i>InstVarDecl</i> *, <i>Method</i> *)
<i>interface</i>	$:=$	interface(<i>type</i> , [extends(<i>type</i>),] <i>MethodHeader</i> *)
<i>InstVarDecl</i>	$:=$	InstVarDecl(<i>type</i> , <i>var</i>)
<i>MethodHeader</i>	$:=$	MethodHeader(<i>mname</i> , <i>type</i> , (<i>var</i> : <i>type</i>)*)
<i>Method</i>	$:=$	Method(<i>mname</i> , <i>type</i> , (<i>var</i> : <i>type</i>)*, <i>block</i>)
<i>block</i>	$:=$	Block(<i>stmt</i> *)
<i>stmt</i>	$:=$	<i>block</i> Return(<i>expr</i>) while(<i>expr</i> , <i>block</i>) LocalVarDecl(<i>var</i> , [<i>type</i>]) If(<i>expr</i> , <i>block</i> [], <i>block</i>) <i>stmtepr</i>
<i>stmtepr</i>	$:=$	Assign(<i>var</i> , <i>expr</i>) New(<i>type</i> , <i>expr</i> *) NewArray(<i>type</i> , <i>expr</i>) MethodCall([<i>expr</i> ,], <i>f</i> (<i>expr</i> *))
<i>expr</i>	$:=$	<i>stmtepr</i> this super LocalOrFieldVar(<i>var</i>) InstVar(<i>expr</i> , <i>var</i>) ArrayAcc(<i>expr</i> , <i>expr</i>)

Figure 5.2: Abstract syntax of core Java 5.0

In this section we consider the type system of Java 5.0. First, we give a definition for the Java 5.0 types of expressions and methods in Java 5.0 classes. Then, we define the Java 5.0 type inference system, whose rules determine the types of Java 5.0 methods. For the whole section let BTV be a set of bounded type variables.

Definition 5.1 (Java 5.0 types) Let $(S\text{Type}_{TS}(BTV), TC)$ be a type signature of declared Java 5.0 classes. Then the corresponding set of Java 5.0 types $\text{Type}(TC(BTV))$ is the smallest set with the properties:

1. Let *Basetype* be the set of the Java 5.0 base types.
Then $\text{Basetype} \subseteq \text{Type}(T_{TC}(BTV))$ (**base type**).
2. $T_{TC}(BTV) \subseteq \text{Type}(T_{TC}(BTV))$ (**simple type**).
3. If for all $0 \leq i \leq n$: $\theta_i \in (T_{\Theta}(TV) \text{ ODER } T_{TC}(BTV) \cup \text{basetype})$ then $\theta_1 \times \dots \times \theta_n \rightarrow \theta_0 \in \text{Type}(T_{TC}(BTV))$ (**function type**).
4. If $ty_1, ty_2 \in \text{Type}(T_{TC}(BTV))$, then $ty_1 \wedge ty_2 \in \text{Type}(T_{TC}(BTV))$ (**intersection type**).

Vergleich
Intersection types
zu
in [GJSB05] ziehen

Remark INTERSECTION SECTION TYPE IM VERGLEICH ZU DEZANI COPPO

The *base types* and the *simple types* describe the types of fields of classes, expressions, and as we will see also the types of statements in methods. The types of the methods, finally, are given as *function types*. The *intersection type* is necessary to describe the complete type of a method, whose code can have more than one function type. If a method has an intersection type, this means that more than one type is derivable for the code of the method by the inference rules. The inference system do not need intersection types, really. If the code of one method has more than one type, different derivations are possible. This means all types are separately derived by the inference system. The **intersection rule** (fig. 5.3) summarizes the different type to one intersection type.

Remark (Commutativity, Associativity, and Neutral element of the intersection operation) The infix operator \wedge is defined similar as the \cap operator in the set theory as commutative and associative. Furthermore, the empty intersection type is assumed as neutral element wrt. \wedge .

In an intersection type $ty = ty_1 \wedge \dots \wedge ty_k$ each ty_i is denoted as a *component* of ty .

Definition 5.2 (Type scheme) The set of type schemes $TS_{\Theta}(TV)$ is the smallest set with the following conditions:

- $T_{\Theta}(TV) \subseteq TS_{\Theta}(TV)$
- If $ty \in TS_{\Theta}(TV)$ then $\forall a. ty \in TS_{\Theta}(TV)$ where $a \in TV$.

Type schemes are the types of classes. If a type variable in a type schemes is generalized the variable can be substituted during the type inference.

For the inference rules we need a set of type assumptions for the methods and variables.

Definition 5.3 (Set of type assumptions) The *set of type assumptions* is a family (map of indices to sets)

$$TS_{\Theta}(TV) \mapsto \{ \text{id} : \tau \mid \text{id is an identifier}, \tau \in \text{Type}(T_{TC}(BTV)) \}.$$

The indices of the family are the class names and the identifiers are the field respectively the method names of the class.

Example 5.4 The sets

$$O_{\text{Math}} = \{ + : \text{int} \times \text{int} \rightarrow \text{int}, \\ \text{sqr} : \text{int} \rightarrow \text{int} \}$$

and

$$O_{\text{Vector}} = \{ \text{add} : \text{Object} \rightarrow \text{void}, \\ \text{len} : \text{int} \}$$

forms the family of type assumptions

$$O = \{ O_{\text{Math}}, O_{\text{Vector}} \}.$$

The meaning of this set of type assumptions is as follows: In the classes **Math** respectively **Vector** exist methods **+** and **sqr** respectively **add** and **len** with the respective types.

Now we declare the type inference system of Java 5.0. The type inference system consists of rules for deriving types of identifiers, expressions, statements, and methods. Because of this the type inference system is divided in four parts. Each part consists of an own type implication relation:

First we consider the main implication relation for the type inference of a whole class.

We write

$$p \triangleright O$$

This means from the Java 5.0 program p the type assumptions of O for the fields and methods of the classes of p are derivable. The class rule (fig. 5.3) describes the condition for this derivation.

The condition in the class rule demanded the derivation of types for the statements of the methods of the classes

$$(O, \tau, \tau') \triangleright_{\text{stmt}} \text{stmts} : \theta$$

This implication means that under the type assumptions O in the class τ , which superclass is τ' the statements stmts have the type θ .

We write

$$(O, \tau, \tau') \triangleright_{\text{expr}} \text{exp} : \theta$$

if for the expression exp the type θ is derivable under the type assumptions O in the class τ , which superclass is τ' .

The rules for the expression are given in the figures 5.8, 5.9, 5.10, 5.11, and 5.12. For the application of fields, variables, and methods a rule is necessary, which derives a type for the respective identifier.

We write therefore

$$O_{\theta} \triangleright_{\text{Id}} f : ty$$

which means that for the identifier f the type ty is derivable in the class θ . The ident rule is given in figure 5.4.

5.4.1 Type inference rule for one class

- Formalisierung der Spezifikation [GJSB05]: Wie kann man beweisen, dass dies korrekt ist.

Man muss dann die Regeln in einem weiteren Kapitel noch fuer Dateien von mehreren Klassen erweitern
Den Typ void betrachten
Beispiel 5.4

DAS SUBTYPING BEISPIEL IN PIZZA+/JAVA.BSP EINBEZIEHEN UND ALGORITHMUS ANPASSEN

Now we will describe the rules for the derivation of the types of one class. These rules are verbalized for the abstract syntax of Java 5.0 classes (see appendix A and cp. [BH98]). In the following we describe the different rules of the type inference system. All rules have the following form: Under the assumption *from a set of type assumptions a type is derivation* it holds that *from another set of type assumptions a further type is derivation*. For this we assume that there are previous classes (called p), which methods are already typed. We denote as an assumption for the derivation of the types of the methods from a further class: $p \blacktriangleright O$, which means that the set of type assumptions O is derived by the Java 5.0 classes p , as explained above.

The basic idea of the inference system is, that we derive from known types of methods and fields the types the methods, which types are not determined so far. Therefore we make some assumptions for theses so far not known types and prove with the inference rules that these type assumptions are correct.

The main inference rules

COMPARE to [IPW99]

Class rule: bind AUSWIRKUNGEN BETRACHTEN

The idea of the **class rule** (5.3) is, that we form three sets of type assumptions \overline{O}_τ , O_{field} , O_{local} . In \overline{O}_τ we make type assumptions for the methods of the class τ . In O_{field} the types of the fields of τ are saved. Finally, in the set O_{local} type assumptions for local variables (the fields of the class, the parameters of the respective methods, and the local declared variables) are made.

These three sets are added to the family of type assumptions O , which holds the declared respectively derived types of other classes.

From this union of type assumptions for each method f_1, \dots, f_m of the class τ the assumed types are proved. This is done by the derivation of the types of the method bodies, which are declared in the abstract syntax by a **Block** constructor. The type of the **Block** constructor is derived by the statement rules (5.5).

If this prove is correct the already typed program is supplemented by the class τ and from this union the generalization (**gen**) of the assumed types for the methods of τ are derivable.

The function $\text{gen} : T_\Theta(TV) \rightarrow TS_\Theta(TV)$ is given as:

$$\text{gen}(\theta) = \forall a_1 \dots \forall a_n \tau, \text{ for } TVar(\theta) = \{a_1, \dots, a_n\}$$

$p \blacktriangleright O$ $\forall 1 \leq i \leq m :$ $(O \cup \{ \tau \mapsto (\overline{O}_\tau \cup O_{field}) \} \cup$ $\{ \langle local \rangle \mapsto (O_{field} \cup \{ v_{i,1} : \theta_{i,1}, \dots, v_{i,k_i} : \theta_{i,k_i} \}) \}, \tau, \tau')$ $\triangleright_{Stmt} \text{Block}(B_i) : \theta_i$	
where $\overline{O}_\tau = \{ f_1 : \theta_{1,1} \times \dots \times \theta_{1,k_1} \rightarrow \theta'_1, \dots, f_m : \theta_{m,1} \times \dots \times \theta_{1,k_m} \rightarrow \theta'_m \}$ $O_{field} = \{ x_1 : \tau_1, \dots, x_n : \tau_n \}$ for $1 \leq i \leq m : \theta_i \leq^* \theta'_i$	
[Class]	$\frac{(p \cup \text{Class}(\text{ClassName}(\tau), \text{extends}(\tau') \text{InstVarDecl}(x_1, \tau_1), \dots, \text{InstVarDecl}(x_n, \tau_n) \text{Method}(f_1, \theta_1, (v_{1,1} : \theta_{1,1}, \dots, v_{1,m_1} : \theta_{1,k_1}), \text{Block}(B_1)) \dots, \text{Method}(f_m, \theta_m, (v_{m,1} : \theta_{m,1}, \dots, v_{m,m_m} : \theta_{m,k_m}), \text{Block}(B_m))))}{p \blacktriangleright O \cup \{ \text{gen}(\tau) \mapsto \overline{O}_\tau \cup O_{field} \}}$
[IntSec]	$\frac{p \blacktriangleright O \cup \{ \tau \mapsto \{ f_1 : ty_1, \dots, f_k : ty_k, \dots, f_m : ty_m \} \} \quad p \blacktriangleright O \cup \{ \tau \mapsto \{ f_1 : ty_1, \dots, f_k : ty'_k, \dots, f_m : ty_m \} \}}{p \blacktriangleright O \cup \{ \tau \mapsto \{ f_1 : ty_1, \dots, f_k : ty_k \wedge ty'_k, \dots, f_m : ty_m \} \}}$

Figure 5.3: Class rules

The generalization means that the type variables can be substituted by any other types. In \overline{O}_τ and in O_{field} the type variables are not generalized. The reason for this is, that during the type inference these type variables must not be substituted, as it even should be proved that the method work for any type at this position. We will explain the effect of type variables binding by an example for the **ident**-rule (figure 5.4).

IntSec rule: The **IntSec rule** (fig. 5.3) summarizes the possibly different types, which are derivable for a method to an intersection type.

The identifier inference rule

[Ident]	$\frac{(O \cup \{ id : \bigwedge_{i \in I} \theta_i \})_{\forall a_1 \dots \forall a_n, \theta'} \quad (O \cup \{ id : \bigwedge_{i \in I} \theta_i \})_{\sigma _{\{a_1, \dots, a_n\}}(\theta) \triangleright_{Id} id : \sigma _{\{a_1, \dots, a_n\}}(\theta_j)}}{\theta \leq^* \theta', j \in I}$
----------------	--

Figure 5.4: Ident rule

Ident rule: The **ident rule** (fig. 5.4) derives the types for methods and variables.

θ' IN CLASS REGEL EINGEFUEHRT, IM ALGO UEBERPRUEFEN, OB IN DEN EXPRESSIONS OLSUB ODER = STEHEN MUSS
IntSec weglassen, man muss in der Ident Regel Durchschnittstypen nicht wieder auseinander nehmen.

The substitute function σ substitutes the generalized type variables by other types. This means that the *ident* rule specializes the known types of the methods and variables.

The following example shows the effect of type variables binding.

Example 5.5 *Let the untyped class*

```
class Add<a> {
  addone(v) {
    return v + 1;
  }
}
```

be given. If we would make the obviously wrong assumption $O_{\forall a.A\langle a \rangle} = \{ \text{addone} : a \rightarrow \text{int} \}$ the *ident*-rule can substitute the type variable a by int . As the argument type int of the operator $+$ is correct the assumed type $\text{addone} : a \rightarrow \text{int}$ would be proved. This is obviously wrong. If we would not bind the type variable a , we would have to prove that the argument type of $+$ is a . This is not possible. Which would mean that the first assumption is wrong.

The statement inference rules

The statement rules (fig. 5.5) derives the result type of a list of statements, which are summerized in a **Block**. This means that only the **Return** statement(s) determine the type of theses lists of statements. Therefore all other statements rules do not change the result type.

The rules for statements type derivation are of the form

$$(O, \tau, \tau') \triangleright_{\text{stmt}} \text{statement} : \text{type}.$$

This means: Under the assumptions O and in the class τ , which superclass is τ' the statement *statement* has the type *type*.

In the following we will consider the statement rules seperately:

BlockInit rules: The rules **BlockVoidInit**, **BlockReturnInit**, and **BlockIfInit** allows to form a block of statements with only one statement. The type of this lonely statement is passed to the block.

In the **BlockIfInit** rule the *then*- and the *else*-branch must have a type. In the other case it would not be sure, that the respective method application gives a result value.

Block.1 rule: The **Block.1** rule adds a new statement of the type **void** to a consisting block of statements. The type of the new statement is **void**. The type of the extended block of statements is unchanged.

There are tree different possibilities if s_1 is a **IfStmt** (cp figure 5.6). Either the **IfStmt** has the type **void**. Then the usual **Block**-rule is applicable. Otherwise the **Block.If1**-rule respectively the **Block.If2**-rule is applicable.

[BlockInit]	$\frac{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{stmt} : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(\text{stmt}) : \theta}$
[BlockReturnInit]	$\frac{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Return}(e) : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(\text{Return}(e)) : \theta}$
[Block.1]	$\frac{(O, \tau, \tau') \triangleright_{\text{stmt}} s_1 : \text{void}, (O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_2; \dots; s_n) : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_1; s_2; \dots; s_n) : \theta}$
[Block.2]	$\frac{(O, \tau, \tau') \triangleright_{\text{stmt}} s_1 : \theta, (O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_2; \dots; s_n) : \theta' \quad \bar{\theta} \in \text{MUB}(\theta, \theta')}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_1; s_2; \dots; s_n) : \bar{\theta}}$
[Return]	$\frac{(O, \tau, \tau') \triangleright_{\text{expr}} e : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Return}(e) : \theta}$
[WhileStmt]	$\frac{(O, \tau, \tau') \triangleright_{\text{expr}} e : \text{boolean}, (O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(B) : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{While}(e, \text{Block}(B)) : \theta}$
[LoVarDecl1]	$\frac{O'_{\langle \text{local} \rangle} = O_{\langle \text{local} \rangle} \setminus \{v : \theta'\} \cup \{v : \bar{\theta}\} \quad ((O \setminus O_{\langle \text{local} \rangle}) \cup O'_{\langle \text{local} \rangle}, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_2; \dots; s_n) : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(\text{LocalVarDecl}(v); s_2; \dots; s_n) : \theta}$
[LoVarDecl2]	$\frac{O'_{\langle \text{local} \rangle} = O_{\langle \text{local} \rangle} \setminus \{v : \theta'\} \cup \{v : \bar{\theta}\} \quad ((O \setminus O_{\langle \text{local} \rangle}) \cup O'_{\langle \text{local} \rangle}, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(s_2; \dots; s_n) : \theta}{(O, \tau, \tau') \triangleright_{\text{stmt}} \text{Block}(\text{LocalVarDecl}(v, \bar{\theta}); s_2; \dots; s_n) : \theta}$

Figure 5.5: Statement Rules

Block.2 rule: The **Block.1** rule adds a new statement of a type unequal to **void** to a consisting block of statements. In this case the already existing **Block**-statement must also have a type unequal to **void**. This means that it contains at least one **return**-statement. The result type of the extended **Block**-statement is the greatest lower bound of the both types.

Return rule: The type of the return statement is determined by the type of the expres-

sion. Finally, this type determines the type of the whole Block, which includes this return statement.

WhileStmt rule: The **WhileStmt** rule types a while statement by the type of the block, which is the body of the while statement. Furthermore the expression which represents the condition must be of the type **boolean**.

LoVarDecl rules: These rules extend the set of type assumptions by a typed variable for the type determination of the following statements. If there is already a type assumption for this variable in the set of type assumptions, this type assumption is erased.

The two rules differ: In the first rule the local variable is declared typeless, while in the second rule a type for the local variable is declared.

In **Java 5.0** there is a possibility to extend a local variable declaration by a initial assign of an expression to the variable. In the abstract syntax these assigns are done by additional assign statements.

Here, we give no rule for the *for statement*. The *for statement* can be transformed to a *while statement*. Because of this no rule for the *for statement* is necessary.

[IfStmt_if_ty1]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e : \text{boolean} \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_1) : \theta}{(O, \tau, \tau') \triangleright_{Stmt} \text{If}(e, \text{Block}(B_1), \epsilon) : \theta}$
[IfStmt_if_ty2]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e : \text{boolean} \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_1) : \theta, \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_2) : \text{void}}{(O, \tau, \tau') \triangleright_{Stmt} \text{If}(e, \text{Block}(B_1), \text{Block}(B_2)) : \theta}$
[IfStmt_el_ty]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e : \text{boolean} \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_1) : \text{void}, \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_2) : \theta'}{(O, \tau, \tau') \triangleright_{Stmt} \text{If}(e, \text{Block}(B_1), \text{Block}(B_2)) : \theta'}$
[IfStmt_if_el_ty]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e : \text{boolean} \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_1) : \theta, \quad (O, \tau, \tau') \triangleright_{Stmt} \text{Block}(B_2) : \theta' \quad \bar{\theta} \in \text{MUB}(\theta_1, \theta_2)}{(O, \tau, \tau') \triangleright_{Stmt} \text{If}(e, \text{Block}(B_1), \text{Block}(B_2)) : \bar{\theta}}$

Figure 5.6: IfStmt statement rules

IfStmt rules: UEBERARBEITEN ALEE NEUEN REGELN BESCHREIBEN The **If-Stmt** rules type if statements. Therefore the types of the blocks representing the if branch respectively the else branch must have comparable types. The greater type is the result type of the if statement. Furthermore the expression which represents the condition must be of the type **boolean**.

[AssignStmt]	$\frac{(O, \tau, \tau') \triangleright_{Expr} \text{Assign}(e_1, e_2) : \theta}{(O, \tau, \tau') \triangleright_{Stmt} \text{Assign}(e_1, e_2) : \text{void}}$
[NewStmt]	$\frac{(O, \tau, \tau') \triangleright_{Expr} \text{New}(\theta, (e_1, \dots, e_n)) : \theta}{(O, \tau, \tau') \triangleright_{Stmt} \text{New}(\theta, (e_1, \dots, e_n)) : \text{void}}$
[NewArray]	$\frac{(O, \tau, \tau') \triangleright_{Expr} \text{NewArray}(\bar{\theta}, e) : \theta[]}{(O, \tau, \tau') \triangleright_{Stmt} \text{NewArray}(\bar{\theta}, e) : \text{void}}$
[MethodCall]	$\frac{(O, \tau, \tau') \triangleright_{Expr} \text{MethodCall}(e, f(e_1, \dots, e_n)) : \theta}{(O, \tau, \tau') \triangleright_{Stmt} \text{MethodCall}(e, f(e_1, \dots, e_n)) : \text{void}}$

Figure 5.7: Expression statement rules

In the expression statement rules (fig. 5.7) types for statements are derived. The only difference between expression statements and statements is, that expression statements are additionally expressions. Therefore there are also expression rules for these statements (compare fig. 5.8).

In the sense of statement type derivation all rules are axioms. The inferred types are always **void**.

The meanings of the conditions for the derivations in the four rules are explained during the explanation of the type derivation for expressions.

The expression inference rules

The structure of the type derivation of expressions is the following: The type of an expression which includes subexpressions is determined from the types of the subexpressions and the type of the applied function.

The rules for expression type derivation are of a similar form as the rules of the statement type derivation:

$$(O, \tau, \tau') \triangleright_{Expr} exp : type.$$

This means: Under the assumptions O and in the class τ , which superclass is τ' the expression exp has the type $type$.

In some rules we need a function **lub**, which determines the smallest arity of a method in a given class.

Ueberprüfen,
ob der Assign-
Expression
den Typ der
Variablen oder
der Expression
bekommt.

[Assign]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e_1 : \theta', (O, \tau, \tau') \triangleright_{Expr} e_2 : \theta}{(O, \tau, \tau') \triangleright_{Expr} \text{Assign}(e_1, e_2) : \theta'} \quad \theta \leq^* \theta'$
[New]	$\frac{\forall 1 \leq i \leq n : (O, \tau, \tau') \triangleright_{Expr} e_i : \theta_i, (\theta'_1 \dots \theta'_n, \theta) = \text{lub}(\theta, \langle \text{init} \rangle_\theta, (\theta_1, \dots, \theta_n))}{(O, \tau, \tau') \triangleright_{Expr} \text{New}(\theta, (e_1, \dots, e_n)) : \theta}$
[New-Array]	$\frac{(O, \tau, \tau') \triangleright_{Expr} e : \text{int}}{(O, \tau, \tau') \triangleright_{Expr} \text{NewArray}(\bar{\theta}, e) : \bar{\theta} []}$
[Method-Call]	$\frac{(O, \tau, \tau') \triangleright_{Expr} re : \bar{\theta}, \forall 1 \leq i \leq n : (O, \tau, \tau') \triangleright_{Expr} e_i : \theta_i, (\theta'_1 \dots \theta'_n, \theta) = \text{lub}(\bar{\theta}, f, (\theta_1, \dots, \theta_n))}{(O, \tau, \tau') \triangleright_{Expr} \text{MethodCall}(re, f(e_1, \dots, e_n)) : \theta}$
[This]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{this} : \tau}$
[Super]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{super} : \tau'}$
[LocalOr-FieldVar]	$\frac{(v : \theta) \in O_{\langle \text{local} \rangle}}{(O, \tau, \tau') \triangleright_{Expr} \text{LocalOrFieldVar}(v) : \theta}$
[InstVar]	$\frac{(O, \tau, \tau') \triangleright_{Expr} re : \bar{\theta}, \quad O_{\bar{\theta}} \triangleright_{Id} v : \theta}{(O, \tau, \tau') \triangleright_{Expr} \text{InstVar}(re, v) : \theta}$
[Array-Acc]	$\frac{(O, \tau, \tau') \triangleright_{Expr} ind : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e : \bar{\theta} []}{(O, \tau, \tau') \triangleright_{Expr} \text{ArrayAcc}(e, ind) : \bar{\theta}}$

Figure 5.8: Expression rules

Definition 5.6 (lub (least upper bound)) *The function*

$$\text{lub} : T_\Theta(TV) \times \text{Identifiers} \times T_\Theta(TV)^* \rightarrow T_\Theta(TV)$$

is defined by:

$$\text{lub}(\bar{\theta}, f, \theta_1 \dots \theta_n) = (\theta'_1 \dots \theta'_n, \theta),$$

if $(\theta'_1 \dots \theta'_n)$ is the smallest tuple with

$$(\theta_1, \dots, \theta_n) \leq^* (\theta'_1, \dots, \theta'_n)$$

and

$$O_{\bar{\theta}} \triangleright_{Id} f : \theta'_1 \times \dots \times \theta'_n \rightarrow \theta.$$

Remark The function lub is a partial function. The result is defined if there is an unambiguously determined tuple with the given properties. If lub is called and no result can be given then the Java 5.0 program is ambiguous, which means that the program is wrong.

The following two examples give wrong Java 5.0 programs where the function lub is not defined.

Example 5.7 *Let the following Java 5.0 program be given:*

```

class M<A> {
    M<A> p(Vector<A> v, Stack<A> w) {
        return new M<A>();
    }
}

class N<A> extends M<A> {
    N<A> p(Stack<A> w, Vector<A> v) {
        return new N<A>();
    }
}

class Main {
    public static void main(String[] args) {
        Stack<Integer> s = new Stack<Integer>();
        System.out.println(new N<Integer>().p(s,s));
    }
}

```

The program is not correct, as the call `new N<Integer>.p(s,s)` is ambiguous. The result of the corresponding call

$$\text{lub}(N\langle \text{Integer} \rangle, p, (\text{Stack}\langle \text{Integer} \rangle, \text{Stack}\langle \text{Integer} \rangle))$$

is undefined. The reason is that it holds

$$O_{N\langle \text{Integer} \rangle} \triangleright_{Id} m : \text{Vector}\langle \text{Integer} \rangle \times \text{Stack}\langle \text{Integer} \rangle \rightarrow M\langle \text{Integer} \rangle$$

and

$$O_{N\langle \text{Integer} \rangle} \triangleright_{Id} m : \text{Stack}\langle \text{Integer} \rangle \times \text{Vector}\langle \text{Integer} \rangle \rightarrow M\langle \text{Integer} \rangle,$$

but there is no derivation for the least upper bound

$\text{Stack}\langle\text{Integer}\rangle \times \text{Stack}\langle\text{Integer}\rangle$

as an arity of \mathbf{m} .

JEDE KLASSE BILDET EIN POSIG, TYPKORREKTEIT DER DES METHODEN-CODE WIRD DURCH DIE REGEL BESTIMMT.

Proof: It is to prove that $\bar{\theta}'$ in definition 5.6 is unambiguous defined, if there is $\bar{\theta}'$.
HIER WEITERMACHEN!!! Verweis theorem ??.

Now we will consider the expression inference rules (fig. 5.8) separately:

Assign rule: The **Assign rule** types an assignment of an expression to a variable as an expression. The assign expression get the type of the variable.

New rule: The **new rule** types the application of constructors. In the abstract syntax of a class the constructors of the class θ is denoted by $\langle\text{init}\rangle_\theta$.

In the rule first the types of the subexpressions and the argument types of the constructor are determined. If these are fit, the applied constructor to the subtypes gets the type of the class.

NewArray rule: A new array is determined by the type of the elements of the array and by an index, which determine the dimension of the array. Therefore the expression, which calculates the index must be of the type **int**. Then the type of the array is the array type of the type of the elements.

Multidimensional arrays are declared by multiple application of the **NewArray** constructor.

MethodCall rule: The **MethodCall rule** is similar to the New rule. First the type of the receiver object of the method application is determined. Then the types of the subexpressions of the method application are determined. Finally the type of the method is determined. Therefore the function **lub** (definition 5.6) is needed.

The result type of the application is then given as the result type of the determined method.

This, Super rule: The type of **this** respectively **super** is given as the type of the actual class respectively as the type of its superclass.

LocalOrFieldVar rule: These variables are known in the actual class. Either the variable is a field of the actual class or it is a parameter respectively a local variable of the actual method. This means that the type of the variable must be derivable from the actual class or its superclasses. The smallest superclass $\bar{\tau}$, which includes this variable is determined. Then the expression gets the type of the variable in $\bar{\tau}$.

InstVar rule: In the **InstVar rule** first the type of the receiver is determined as in the **MethodCall rule**. Then the smallest class, which greater than the receiver type, where the field is declared, is determined. From this class the type of the variable is derivable. Then, this type is the type of the expression.

ArrayAcc rule: In the **ArrayAcc rule** first the type of the index expression must be **int**. Then the array type of the array is determined. From this follows the type of the element of the array.

Literal expression rules

[IntLiteral]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{IntLiteral}(n) : \text{int}}$
[BoolLiteral]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{BoolLiteral}(b) : \text{boolean}}$
[CharLiteral]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{CharLiteral}(c) : \text{char}}$
[NullLiteral]	$\frac{}{(O, \tau, \tau') \triangleright_{Expr} \text{Null} : \theta'}$

Figure 5.9: Literal expression rules

The inference rules in figure 5.9 determines the types of literals. Literals are also expressions.

IntLiteral, BoolLiteral, and CharLiteral rule: These three rules determines the (base)types for the respective literals.

NullLiteral rule: The literal **null** is the null pointer for objects of any class. This means that **null** gets any reference type.

Unary expression rules

The unary expression rules (fig. 5.10) determines types for the standard functions of Java 5.0 with one argument:

UnaryMinus rule: The unary minus transforms an **int** expression to its negative. Therefore in the pre-condition of the rule the type of the expression must be **int**. Then the type of the application of **UnaryMinus** to the expression is also **int**.

[UnaryMinus]	$(O, \tau, \tau') \triangleright_{Expr} e : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{UnaryMinus}(e) : \text{int}$	
[Not]	$(O, \tau, \tau') \triangleright_{Expr} e : \text{boolean}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Not}(e) : \text{boolean}$	
[Cast]	$(O, \tau, \tau') \triangleright_{Expr} e : \theta$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Cast}(\bar{\theta}, e) : \bar{\theta}$	

Figure 5.10: Unary expression rules

Not rule: The **Not** rule works as the **UnaryMinus** rule.

Cast rule: A type cast transforms an expression of any type in an expression of the type which is given in the cast. Therefore the type of the cast expression is the type, which is given in the cast operation.

Integer binary expression rules

[Add]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Add}(e_1, e_2) : \text{int}$	
[Minus]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Minus}(e_1, e_2) : \text{int}$	
[Mul]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Mul}(e_1, e_2) : \text{int}$	
[Divide]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Div}(e_1, e_2) : \text{int}$	
[Modulo]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Mod}(e_1, e_2) : \text{int}$	

Figure 5.11: Integer binary expression rules

The integer binary expression rules (fig. 5.11) have the same structure. In each precondition two **int** expression are demanded. The functions **Add**, **Minus**, **Mul**, **Div**, and **Mod** are applied to these two expressions. The result type is in each case **int**.

Boolean binary expression rules

[Equal]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \theta, (O, \tau, \tau') \triangleright_{Expr} e_2 : \bar{\theta}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Eq}(e_1, e_2) : \text{boolean}$	
[NotEqual]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \theta, (O, \tau, \tau') \triangleright_{Expr} e_2 : \bar{\theta}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{NEq}(e_1, e_2) : \text{boolean}$	
[Less]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Less}(e_1, e_2) : \text{boolean}$	
[LessEqual]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{LessEq}(e_1, e_2) : \text{boolean}$	
[Greater]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{Greater}(e_1, e_2) : \text{boolean}$	
[GreaterEqual]	$(O, \tau, \tau') \triangleright_{Expr} e_1 : \text{int}, (O, \tau, \tau') \triangleright_{Expr} e_2 : \text{int}$	
	$(O, \tau, \tau') \triangleright_{Expr} \text{GreaterEq}(e_1, e_2) : \text{boolean}$	

Figure 5.12: Boolean binary expression rules

The structure of the binary predicates (boolean binary expressions, fig. 5.12) is very similar to the structure integer binary expression. In the boolean binary expressions merely the result type is boolean.

A little different are the **Equal** and the **NotEqual** rule. Here, the argument types are arbitrary.

Now we consider an example for the type inference rules.

5.4.2 Type inference example

HIER WEITERMACHEN UND BEISPIEL AUF KONFORMITÄT MIT REGELN PRÜFEN

In this section we show as an example the type inference for the **mul** method from the **Matrix** example (example ??). For this we omit the type declarations of the parameter and the local variables. Furthermore, we transform the *for-loops* into *while-loops*, as the type inference for *while-loops* is more obvious, although it would work for the *for-loops* as well (fig. 5.13).

This concret syntax is transformed to the **Java 5.0** abstract syntax (cp. appendix A):

```

class Matrix extends Vector<Vector<Integer>> {

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

```

Figure 5.13: The Java 5.0 class Matrix

```

absMatrix =
class(
    ClassName(Matrix),
    extends( Vector<Vector<Integer>> ),
    Method( mul,
        [],
        (m:[]),
        Block(
            LocalVarDecl( ret );
            Assign( LocalOrFieldVar( ret ), New( Matrix, () ) );
            LocalVarDecl( i );
            Assign( LocalOrFieldVar( i ), IntLiteral( 0 ) );

```

```

WhileStmt(
    Less( LocalOrFieldVar( i ), MethodCall( this, size() ) ),
    Block(
        LocalVarDecl( v1 );
        Assign( LocalOrFieldVar( v1 ),
            MethodCall( this, elementAt( LocalOrFieldVar( i ) ) ) );
        LocalVarDecl( v2 );
        Assign( LocalOrFieldVar( v2 ), New( Vector<Integer>, () ) );
        LocalVarDecl( j );
        Assign( LocalOrFieldVar( j ), IntLiteral( 0 ) );
        WhileStmt(
            Less( LocalOrFieldVar( j ),
                MethodCall( LocalOrFieldVar( v1 ), size() ) ),
            Block(
                LocalVarDecl( erg );
                Assign( LocalOrFieldVar( erg ), IntLiteral( 0 ) );
                LocalVarDecl( k );
                Assign( LocalOrFieldVar( k ), IntLiteral( 0 ) );
                WhileStmt(
                    Less( LocalOrFieldVar( k ),
                        MethodCall( LocalOrFieldVar( v1 ), size() ) ),
                    Block(
                        Assign(
                            LocalOrFieldVar( erg ),
                            Add( LocalOrFieldVar( erg ),
                                Mul( MethodCall(
                                    MethodCall( LocalOrFieldVar( v1 ),
                                        elementAt( LocalOrFieldVar( k ) ) ),
                                    intValue() )
                                MethodCall(
                                    MethodCall(
                                        LocalOrFieldVar( m ),
                                        elementAt( LocalOrFieldVar( k ) ) ),
                                        elementAt( LocalOrFieldVar( j ) ) ),
                                    intValue() ) ) ) );
                        Assign(
                            LocalOrFieldVar( k ),
                            Add( LocalOrFieldVar( k ), IntLiteral( 1 ) ) ) );
                    MethodCall(
                        LocalOrFieldVar( v2 ),
                        addElement( New( Integer, (LocalOrFieldVar( erg )) ) ) );
                    Assign(
                        LocalOrFieldVar( j ),

```



```

    Add( LocalOrFieldVar( j ), IntLiteral( 1 ) ) ) ) );
    MethodCall( LocalOrFieldVar( ret ),
        addElement( LocalOrFieldVar( v2 ) ) );
    Assign( LocalOrFieldVar( i ),
        Add( LocalOrFieldVar( i ), IntLiteral( 1 ) ) ) );
    Return( LocalOrFieldVar( ret ) ) ) ) )

```

Now we will present the derivation of the types for the above program. Derivations of inference rules like this works in the following way. First we will make an assumption for the family of type assumptions $O_{\mathbf{VA.Vector}\langle A \rangle}$, $O_{\mathbf{Integer}}$, and $O_{\mathbf{Matrix}}$. The sets of type assumptions $O_{\mathbf{VA.Vector}\langle A \rangle}$ and $O_{\mathbf{Integer}}$ are directly given by a cutout of the signature of the corresponding classes:

$$O_{\mathbf{VA.Vector}\langle A \rangle} = \{ \text{elementAt} : \text{int} \rightarrow A, \\ \text{addElement} : A \rightarrow \text{void}, \\ \text{size} : \text{int} \}$$

and

$$O_{\mathbf{Integer}} = \{ \text{<init>}_{\mathbf{Integer}} : \text{int} \rightarrow \mathbf{Integer}, \\ \text{intValue} : \text{int} \}.$$

Furthermore, we need a set of type assumptions $O_{\mathbf{Matrix}}$. As the class **Matrix** has no fields, only an assumption for the method **mul** is needed. If we had made such an assumption, we must prove that this assumption is correct by the given inference rules. In this section we will give the assumption not until we have given the prove by the inference rules. This means we will derive this assumption by the inference prove.

Let in the following $\tau = \mathbf{Matrix}$ and $\tau' = \mathbf{Vector}\langle \mathbf{Vector}\langle \mathbf{Integer} \rangle \rangle$.

If we regard the **Block** rule, we see that we have to start at the end of the block. Therefore the first type which must be inferred, is the type of the statement **Return(LocalOrFieldVar(ret))**. If we, furthermore regard the **LocalVarDecl** rules, we see, that by these rules the set of type assumption is extended. This means that for the derivation of **Return(LocalVarDecl(ret))** the set of type assumptions $O_{\langle \text{local} \rangle}$ must be extended by $\{ \text{ret} : \mathbf{Matrix}, i : \text{int} \}$ ¹. The set extended in this way is denoted by $O'_{\langle \text{local} \rangle}$ and the whole family is denoted by O' .

Now, we can start:

$$\begin{array}{c}
 \text{[LocalOr-FieldVar]} \frac{O'_{\langle \text{local} \rangle} \triangleright_{Id} \text{ret} : \mathbf{Matrix}}{(O', \tau, \tau') \triangleright_{Expr} \text{LocalOrFieldVar(ret)} : \mathbf{Matrix}} \\
 \text{[Return]} \frac{}{(O', \tau, \tau') \triangleright_{Stmt} \text{Return(LocalOrFieldVar(ret))} : \mathbf{Matrix}} \\
 \text{[Block-Init]} \frac{}{(O', \tau, \tau') \triangleright_{Stmt} \text{Block(Return(LocalOrFieldVar(ret)))} : \mathbf{Matrix}} \quad (5.1)
 \end{array}$$

The next step is to derive the type of the outermost loop. With this result, we can then apply the **Block** rule, again (compare (5.14)). For this loop the set of type assumptions

¹If we would make the type assumption $\text{ret} : \mathbf{Vector}\langle \mathbf{Vector}\langle \mathbf{Integer} \rangle \rangle$ we would get also an correct inference.

$O'_{\langle \text{local} \rangle}$ must be extended as in the main block. Here the extension is given by $\{ \mathbf{v1} : \mathbf{Vector}\langle \mathbf{Integer} \rangle, \mathbf{v2} : \mathbf{Vector}\langle \mathbf{Integer} \rangle, j : \text{int} \}$. The set extended in this way is denoted by $O''_{\langle \text{local} \rangle}$ and the whole family is denoted by O'' .

The last statement in this loop is

```
Assign( LocalOrFieldVar( i ), Add( LocalOrFieldVar( i ), IntLiteral( 1 ) ) ).
```

The result of the rule **LocalOrFieldVar** is

$$(O'', \tau, \tau') \triangleright_{Expr} \text{LocalOrFieldVar(i)} : \text{int}$$

and the result of the rule **Add** is

$$(O'', \tau, \tau') \triangleright_{Expr} \text{Add(LocalOrFieldVar(i), IntLiteral(1))} : \text{int}.$$

From this follows:

$$\begin{array}{c}
 \text{[AssignStmt]} \frac{(O'', \tau, \tau') \triangleright_{Expr} \text{LocalOrFieldVar(i)} : \text{int} \quad (O'', \tau, \tau') \triangleright_{Expr} \text{Add(LocalOrFieldVar(i), IntLiteral(1))} : \text{int}}{(O'', \tau, \tau') \triangleright_{Stmt} \text{Assign(LocalOrFieldVar(i), Add(LocalOrFieldVar(i), IntLiteral(1)))} : \text{void}} \quad (5.2)
 \end{array}$$

The next step is the derivation of the type of

```
MethodCall( LocalOrFieldVar( ret ), addElement( LocalOrFieldVar( v2 ) ) ).
```

The type is derived by the expression statement **MethodCall** rule:

$$\begin{array}{c}
 \text{[MethodCall]} \frac{(O'', \tau, \tau') \triangleright_{Expr} \mathbf{v2} : \mathbf{Vector}\langle \mathbf{Integer} \rangle, \quad (O'', \tau, \tau') \triangleright_{Expr} \text{LocalOrFieldVar(ret)} : \mathbf{Matrix} \quad (\mathbf{Vector}\langle \mathbf{Integer} \rangle, \text{void}) = \text{lub}(\mathbf{Matrix}, \text{addElement}, \mathbf{Vector}\langle \mathbf{Integer} \rangle)}{(O'', \tau, \tau') \triangleright_{Stmt} \text{MethodCall(LocalOrFieldVar(ret), addElement(LocalOrFieldVar(v2)))} : \text{void}} \quad (5.3)
 \end{array}$$

The next step is to derive the type of the next loop. With this result, we can apply the **Block** rule. For this loop the set of type assumptions $O''_{\langle \text{local} \rangle}$ must be extended as in the outermost block. Here the extension is given by $\{ \mathbf{erg} : \text{int}, \mathbf{k} : \text{int} \}$. The in this way extended set is denoted by $O'''_{\langle \text{local} \rangle}$ and the whole family is denoted by O''' .

Analogous to (5.2) in the outermost loop the following type is derived:

$$\begin{array}{c}
 (O''', \tau, \tau') \triangleright_{Stmt} \text{Assign(LocalOrFieldVar(j), Add(LocalOrFieldVar(j), IntLiteral(1)))} : \text{void} \quad (5.4)
 \end{array}$$

Then, the type for

`MethodCall(LocalOrFieldVar(v2), addElement(New(Integer, (LocalOrFieldVar(erg)))))`

is derived.

First the **New** rule is applied

$$(O''', \tau, \tau') \triangleright_{Expr} \text{New}(\text{Integer}, (\text{LocalOrFieldVar}(\text{erg}))) : \text{Integer}$$

With the **MethodCall** rule follows then

$$(O''', \tau, \tau') \triangleright_{Stmt} \text{MethodCall}(\text{LocalOrFieldVar}(v2), \text{addElement}(\text{New}(\text{Integer}, (\text{LocalOrFieldVar}(\text{erg})))) : \text{void}. \quad (5.5)$$

The next step is the type inference for the inner loop. First analogous to (5.2) and (5.4) the following type is derived:

$$(O''', \tau, \tau') \triangleright_{Stmt} \text{Assign}(\text{LocalOrFieldVar}(k), \text{Add}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(1))) : \text{void} \quad (5.6)$$

In the assignment to the local variable `erg` a type assumption for the method parameter `m` is needed. From the method `elementAt`, which is called on `m` follows that `m` must be a `Vector<A>`. If we further consider that on the result of this method the method `elementAt` is called again, then the type of `m` must be `Vector<Vector<A>>`. Finally, on the result of the method, the method `intValue` is called, which means that `m` must have the type `Vector<Vector<Integer>>`². These considerations lead to the extension of O_{Matrix} by $\{m : \text{Vector}<\text{Vector}<\text{Integer}>>\}$. This means that O'_{Matrix} , O''_{Matrix} , and O'''_{Matrix} are also extended in this way.

Then with the **MethodCall**, the **Mul**, the **Add**, and the **AssignStmt** rule follows

$$(O''', \tau, \tau') \triangleright_{Stmt} \text{Assign}(\text{LocalOrFieldVar}(\text{erg}), \text{Add}(\text{LocalOrFieldVar}(\text{erg}), \text{Mul}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{elementAt}(\text{LocalOrFieldVar}(k))), \text{intValue}()), \text{MethodCall}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(m), \text{elementAt}(\text{LocalOrFieldVar}(k))), \text{elementAt}(\text{LocalOrFieldVar}(j))), \text{intValue}())))) : \text{void} \quad (5.7)$$

For the termination condition of the inner loop it holds with the **Less** rule:

$$(O''', \tau, \tau') \triangleright_{Expr} \text{Less}(\text{LocalOrFieldVar}(k), \text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{size}())) : \text{boolean} \quad (5.8)$$

²As `Matrix` is a subtype of `Vector<Vector<Integer>>` the assumption `m : Matrix` would also be possible. Later we will see that really both assumptions are solutions and that these two assumptions leads to a more general type of the method `mul` than we declared in chapter 2.

From (5.6), (5.7), and (5.8) with the **BlockInit** and the **Block** rule the pre-conditions of the **WhileStmt** rule are fulfilled. Then, with the **WhileStmt** rule follows

$$(O''', \tau, \tau') \triangleright_{Stmt} \text{WhileStmt}(\text{Less}(\text{LocalOrFieldVar}(k), \text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{size}()))), \text{Block}(\text{Assign}(\text{LocalOrFieldVar}(\text{erg}), \text{Add}(\text{LocalOrFieldVar}(\text{erg}), \text{Mul}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{elementAt}(\text{LocalOrFieldVar}(k))), \text{intValue}()), \text{MethodCall}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(m), \text{elementAt}(\text{LocalOrFieldVar}(k))), \text{elementAt}(\text{LocalOrFieldVar}(j))), \text{intValue}()))))) : \text{void} \quad (5.9)$$

With the **Assign**, the **BlockInit**, and the **Block** rule follows:

$$(O''', \tau, \tau') \triangleright_{Stmt} \text{Block}(\text{Assign}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(0)) ; \text{WhileStmt}(\dots); \text{MethodCall}(\text{LocalOrFieldVar}(v2), \dots); \text{Assign}(\text{LocalOrFieldVar}(j), \dots)) : \text{void} \quad (5.10)$$

Then the **LocalVarDecl1** rule can be applied:

$$\frac{[\text{LocalVar-Decl1}] \quad (5.10)}{(\{O'''_{\text{Vector}<\text{A}>}, O'''_{\text{Integer}}, O'''_{\text{Matrix}}, O'''_{\text{Local}} \setminus \{k : \text{int}\}\}, \tau, \tau') \triangleright_{Stmt} \text{Block}(\text{LocalVarDecl}(k); \text{Assign}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(0)); \text{WhileStmt}(\dots); \text{MethodCall}(\text{LocalOrFieldVar}(v2), \dots); \text{Assign}(\text{LocalOrFieldVar}(j), \dots)) : \text{void} \quad (5.11)}$$

Analogous follows then

$$(O'', \tau, \tau') \triangleright_{Stmt} \text{Block}(\text{LocalVarDecl}(\text{erg}) \text{Assign}(\text{LocalOrFieldVar}(\text{erg}), \text{IntLiteral}(0)) \text{LocalVarDecl}(k); \text{Assign}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(0)); \text{WhileStmt}(\dots); \text{MethodCall}(\text{LocalOrFieldVar}(v2), \dots); \text{Assign}(\text{LocalOrFieldVar}(j), \dots)) : \text{void} \quad (5.12)$$

From this follows

$$\begin{aligned}
 (O'', \tau, \tau') \triangleright_{Stmt} & \text{WhileStmt}(\\
 & \text{Less}(\text{LocalOrFieldVar}(j), \\
 & \quad \text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{size}(),), \\
 & \text{Block}(\text{LocalVarDecl}(erg) \\
 & \quad \text{Assign}(\text{LocalOrFieldVar}(erg), \text{IntLiteral}(0)) \quad (5.13) \\
 & \quad \text{LocalVarDecl}(k); \\
 & \quad \text{Assign}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(0)); \\
 & \quad \text{WhileStmt}(\dots); \\
 & \quad \text{MethodCall}(\text{LocalOrFieldVar}(v2), \dots); \\
 & \quad \text{Assign}(\text{LocalOrFieldVar}(j), \dots)) : \text{void}
 \end{aligned}$$

Analogous follows that the type of the middle while loop and the outermost while loop has the type `void`. With the Block rule and (5.1) follows then

$$\begin{aligned}
 (O, \tau, \tau') \triangleright_{Stmt} & \text{Block}(\text{LocalVarDecl}(\text{ret}); \\
 & \text{Assign}(\text{LocalOrFieldVar}(\text{ret}), \text{New}(\text{Matrix}, ())) ; \\
 & \text{LocalVarDecl}(i); \\
 & \text{Assign}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(0)) ; \\
 & \text{WhileStmt}(\dots); \\
 & \text{Return}(\text{LocalOrFieldVar}(\text{ret})) : \text{Matrix} \quad (5.14)
 \end{aligned}$$

This allows to apply the Class rule:

$$\begin{aligned}
 & \{ \text{Class}(\text{Classname}(\text{Integer}), \dots), \\
 & \quad \text{Class}(\text{Classname}(\text{Vector}<\text{A}>), \dots) \} \triangleright \{ O_{\text{Integer}}, O_{\text{Vector}<\text{A}>} \}, \quad (5.14) \\
 \text{[Class]} \quad & \frac{}{ \{ \text{Class}(\text{Classname}(\text{Integer}), \dots), \\
 & \quad \text{Class}(\text{Classname}(\text{Vector}<\text{A}>), \dots) \} \cup \\
 & \quad \text{Class}(\text{ClassName}(\text{Matrix}), \text{extends}(\text{Vector}<\text{Vector}<\text{Integer}>>), \\
 & \quad \quad \text{Method}(\text{mul}, \text{Matrix}, (\text{m} : \text{Vector}<\text{Vector}<\text{Integer}>>), \\
 & \quad \quad \quad \text{Block}(\text{LocalVarDecl}(\text{ret}), \dots)) \\
 & \quad \triangleright \{ O_{\text{Integer}}, O_{\text{Vector}<\text{A}>} \} \cup \\
 & \quad \{ \text{Matrix} \mapsto \text{gen}(\{\text{mul} : \text{Vector}<\text{Vector}<\text{Integer}>> \rightarrow \text{Matrix}\}) \} \quad (5.15)
 \end{aligned}$$

This means that the result of this inference is the typed method

$$\text{mul} : \text{Vector}<\text{Vector}<\text{Integer}>> \rightarrow \text{Matrix}.$$

If we remember that for the parameter `m` of the method `mul` we can make also the type assumption `Matrix`, then we can derive analogously the type `Matrix` \rightarrow `Matrix` for this method. As we can additionally make the assumption that `ret` has the type `Vector<Vector<Integer>>` the **IntSec** rule derives an intersection type with four com-

ponents:

$$\begin{aligned}
 p \triangleright & \{ O_{\text{Integer}}, O_{\text{Vector}<\text{A}>} \} \cup \\
 & \{ \text{Matrix} \mapsto \{ \text{mul} : \text{Vector}<\text{Vector}<\text{Integer}>> \rightarrow \text{Matrix} \\
 & \quad \wedge \text{Matrix} \rightarrow \text{Matrix} \\
 & \quad \wedge \text{Matrix} \mapsto \text{Vector}<\text{Vector}<\text{Integer}>> \\
 & \quad \wedge \text{Vector}<\text{Vector}<\text{Integer}>> \mapsto \text{Vector}<\text{Vector}<\text{Integer}>> \} \quad (5.16)
 \end{aligned}$$

where p is the abstract syntax of the classes `Integer`, `Vector<A>` and `Matrix`, as in (5.15). If we compare the result of (5.16) with the typed example (example ??) in chapter 2 we see that the method `mul` is also applicable to an object typed by `Vector<Vector<Integer>>` and the result type can be also `Vector<Vector<Integer>>`. In we try to interpret this semantically, this means that the instance, which is mapped to the parameter `m` and the instance, which is returned by the method `mul`, needs respective no method `mul`. Furthermore the means that the inferred type is more general than the declared type. This is an example for the major advantage of type inference, that the user need not to declare the exact subtype and the exact type instances for parameters, local variables, and results of methods.

WEITERES BEISPIEL VectorAdd.tex

5.4.3 Principal type property

bind AUSWIRKUNGEN BETRACHTEN

In this section we will define most principal types for the methods of Java 5.0 classes. Then, we will show that the most principal types are derivable by the type inference system. Finally the most principal types will be the results determined in the type reconstruction algorithm, which we will present in the next section.

First, we introduce the problem by a small example:

Example 5.8 We consider the method `get` of the class `Vector<a>`. The well-known type of the method is

$$\text{get} : \rightarrow \text{a}.$$

But if we would consider the declaration the type

$$\text{get} : \rightarrow \text{Object}$$

and the type

$$\text{get} : \rightarrow \text{Integer}$$

would also be derivable. But this means that then by the **IntSec** rule the intersection type

$$\text{get} : \rightarrow \text{Integer} \wedge \rightarrow \text{Integer}$$

would also be valid.

In this section we will no define one type, the *most principal type*, which is the initial point for all type derivations.

Definition 5.9 (Most principal type) Let $p \blacktriangleright O$ with $(O' \cup \{id : ty\})_\theta \in O$. The type ty is called most principal type of the identifier id if for all deriveable types τ of the identifier id (which means that there is \bar{O} with $p \blacktriangleright \bar{O}$ and $\bar{O}_\theta \triangleright_{Id} id : \tau$) it is also valid

$$(O' \cup \{id : ty\})_\theta \triangleright_{Id} id : \tau.$$

Theorem 5.10 (Existence of the most principal type) Let p be a Java 5.0 program. Then, for each derivation $p \blacktriangleright O$ with $(O \cup \{id : ty\})_\theta \triangleright_{Id} id : \tau$ there is a most principal type \bar{ty} of id with $\bar{ty} = \bigwedge_{i \in I} \bar{\theta}_i$ where the index set I is finite and there is a $j \in I$ and a substitution σ with $\sigma(\bar{\theta}_j) = \tau$.

Proof: From the **Ident** rule and the **IntSec** rule follows that a type $\bar{ty} = \bigwedge_{i \in I} \bar{\theta}_i$ with the property that there is a $j \in I$ and a substitution σ with $\sigma(\bar{\theta}_j) = \tau$ exists. But it is not obvious if I is finite.

For the proof of the finiteness of I we consider the causes for $p \blacktriangleright O$. In the **class** rule for the respective block of the method the return type must be derived under the type assumptions of the class methods. The only possibilities to multiply types of the methods are the identifier type derivation in the **MethodCall** rule (figure 5.8) and the **NullLiteral** rule (figure 5.9).

In the **MethodCall** rule there are only a finite number of type assumptions for the respective identifiers. The types θ_i for $1 \leq i \leq n$ are derived if the most general instances of these finite number of types, with which a type for the block is deriveable, are derived in the **MethodCall** rule.

The **NullLiteral** rule leads only to different types of a method, if the statement **return : null** is given. But in this case all different derived type are instances of the most general type represented by a type variable. ■

We have proved, that there is for each method of any Java 5.0 class a most principal type. Now we will consider the uniqueness of the most principal type. For this we consider the structure of the type system more detailed. First we define an equivalence relation on the set of type terms.

Definition 5.11 (Equivalent type terms) Two type terms θ and τ are equivalent (in sign: $\theta \sim \tau$) if they are in the reflexiv, symmetric, and transitive closure of the relation which defined by the following conditions:

1. It holds $\sigma(\theta) = \tau$ and $\sigma'(\tau) = \theta$, where θ and τ are either base types or simple types.
2. It holds $\theta = \bigwedge_{i \in I} \theta_i$, $\tau = \bigwedge_{i \in I} \tau_i$, where all θ_i and τ_i are base types or simple types, and there is a permutation π such that for all $1 \leq i \leq n$: $\theta_i \sim \tau_{\pi(i)}$.
3. There is a type $\hat{\theta}$ with $\tau = \theta \wedge \hat{\theta}$, where $\hat{\theta}$ is a base type or a simple type and there are types $\bar{\theta}, \bar{\theta}$ with $\theta = \bar{\theta} \wedge \hat{\theta}$ and there is a substitution σ with $\bar{\theta} = \sigma(\bar{\theta})$.

The equivalence relation \sim defines equivalence classes $[\theta]_\sim = \{\theta' \mid \theta' \sim \theta\}$.

A representant of an equivalence class $[\theta]_\sim$ is a type term $\bigwedge_{i \in I} \theta_i$ where the cardinal number of the index set I is minimal.

With the first condition types with renamed variables are considered as equivalent. The second condition allows any range of the intersections in equivalent types. The third condition determines that if an intersection type has one type and an instance of this type as type, then the types are considered as equivalent.

Now we give for the first three conditions examples.

Example 5.12 Let $\Theta = \{\Theta^{(0)}, \Theta^{(1)}\}$ with $\Theta^{(1)} = \{\mathbf{Vector}\}$, $\Theta^{(0)} = \{\mathbf{Integer}\}$ and $\mathbf{a}, \mathbf{b} \in TV$ type variables. Then, the following holds:

1. $\mathbf{Vector}\langle \mathbf{a} \rangle \sim \mathbf{Vector}\langle \mathbf{b} \rangle$
2. $(\mathbf{Vector}\langle \mathbf{a} \rangle \wedge \mathbf{Object}) \sim (\mathbf{Object} \wedge \mathbf{Vector}\langle \mathbf{a} \rangle)$
3. $\mathbf{Vector}\langle \mathbf{a} \rangle \sim \mathbf{Vector}\langle \mathbf{a} \rangle \wedge \mathbf{Vector}\langle \mathbf{Object} \rangle$

$\mathbf{Vector}\langle \mathbf{a} \rangle$ is a representant of the equivalence class $[\mathbf{Vector}\langle \mathbf{a} \rangle]_\sim$.

The next example shows, why the three conditions of definition 5.11 determine no equivalence relation respectively why the reflexive, symmetric, and transitive closure building is needed.

Example 5.13 Let $\theta = \mathbf{Vector}\langle \mathbf{a} \rangle$ and $\tau = \mathbf{Vector}\langle \mathbf{b} \rangle \wedge \mathbf{Vector}\langle \mathbf{Integer} \rangle$. Then it holds $\tau \sim \mathbf{Vector}\langle \mathbf{b} \rangle$ (condition 3) and $\theta \sim \mathbf{Vector}\langle \mathbf{b} \rangle$ (condition 1). But with the first three conditions it do not hold $\theta \sim \tau$. With the transitivity property it holds $\theta \sim \tau$.

Remark The definition of the representant is very important as the type reconstruction algorithm in the next section determines for each method a representant of the most principal type.

Theorem 5.14 (Uniqueness of the most principal type) The most principle type of a method id in a class θ is unique modulo equivalence. This means if $p \blacktriangleright O$ with $(O \cup \{id : ty\})_\theta \in O$ and $p \blacktriangleright O'$ with $(O' \cup \{id : ty'\})_\theta \in O$, where ty and ty' are both most principal types, then it holds $ty \sim ty'$.

Proof: We assume $p \blacktriangleright O$ with $(O \cup \{id : ty\})_\theta \triangleright_{Id} id : \tau$. As ty' is also a most principal type it holds $p \blacktriangleright O'$ with $(O' \cup \{id : ty'\})_\theta \triangleright_{Id} id : \tau$. Then, we have to prove $ty \sim ty'$.

With the **Ident** rule follows that ty and ty' contains simple types or base types $\bar{\tau}$ respectively $\bar{\tau}'$, such that there are substitutions σ and σ' with $\sigma(\bar{\tau}) = \tau$ and $\sigma'(\bar{\tau}') = \tau$. But this means that either holds $\bar{\tau} = \bar{\tau}'$ or in the other case from the fact that each type of a type assumption of a identifier itself is also derivable as type of the identifier and the fact that ty and ty' are most principal types follows $(O \cup \{id : ty\})_\theta \triangleright_{Id} id : \bar{\tau}'$ and $(O \cup \{id : ty'\})_\theta \triangleright_{Id} id : \bar{\tau}$. But this means that $\bar{\tau} \wedge \bar{\tau}'$ is a part of ty and ty' .

Inductively over all types of ty respectively ty' follows then $ty \sim ty'$. ■

correct => sound

5.5 Type Reconstruction

In the following type inference rules as well as in the type reconstruction algorithm (section 5.5) sometimes the type variables and the variables of the pattern assumptions must be refreshed. This means that all variables are substituted by new variables. We denote this refreshing process by the function call `fresh(θ)`.

Now, we will present a type reconstruction algorithm for OBJ-P programs. The goal is to reconstruct the type declarations of non-explicitly typed programs such that it corresponds to the type inference rules of the previous section.

In the next section we present the algorithm and in section ?? we will prove its soundness and completeness wrt. the type inference rules.

5.5.1 Type Reconstruction Algorithm

- Wie detailliert soll der Beweis sein, dass TI-Regeln dem Algorithmus entsprechen?

The type reconstruction algorithm `TRprog` reconstructs types and replicates equations for groups of equations declaring mutually recursive functions. This means `TRprog` must be applied step by step to each group of equations of an OBJ-P program. The result of an application of `TRprog` to a group of equations is the input of the `TRprog` application of the next group of equations. The type reconstruction algorithm works only if the function symbols of the group or the length of the arities of the function symbols are pairwise different. This precondition assures that there is no overloaded mutual recursive function call, as demanded in section ??.

As said in section ??, the types of the equations are determined by two different type declarations in the program. On the one hand the types of the function symbols are declared by `op` declarations and on the other hand the types of the variables of the patterns of the equations are declared by `vars` declarations. These unusual variable type declarations are necessary such that no equation is overloaded.

Our type reconstruction algorithm reconstructs the type declarations of the function symbols as well as the variables type declarations. Therefore, equations can be overloaded in a yet not typed OBJ-P-program. In this case the type reconstruction algorithm adds copies of equations and rebuild the patterns such that the equations are not overloaded. The type reconstruction algorithm needs the type unification algorithms of chapter 4. We denote the algorithms by `TUnify~` and `TUnify($\leq^* \cup \leq^{*-1}$)`, respectively.

Type Reconstruction for groups of Mutual Recursive Equations

Consider a group of equations:

```
rec is
  exp1l = exp1r
  ⋮
  expnl = expnr
endr.
```

braucht
vielleicht
man

The reconstruction of the types and the replicate process of the corresponding equations are determined in two steps. First, the types of the left hand sides (patterns of the equations) are determined by the function `TRL`. After that, the types of the equations are determined by the function `TRR`, which reconstructs the types of the right hand sides, where the results of `TRL` are the inputs for `TRR`. The reconstruction of the left hand sides is easier, as patterns are terms without any recursive function calls. For the reconstruction of the right hand side an assumption of the type of the actual function symbols is needed, which must be adapted if the type of a recursive function call or the types of the variables are reconstructed.

The start function `TRprog` takes as arguments a set of type assumptions A , a set of typed variables V_{old} , and a group of OBJ-P equations. A contains type assumptions for the function symbols and V_{old} contains typed variables which are determined before, either user declared or reconstructed from previous groups of equations.

The result of `TRprog` is a triple of a set of reconstructed type assumptions for the function symbols, a set of reconstructed typed variables, and a set of equations. The set of type assumptions contains the union of A and the reconstructed types for the function symbols of the group of OBJ-P equations. The set of typed variables is the union of V_{old} and the set of reconstructed typed variables. The result set of equations contains the rebuilt copies of the given group of OBJ-P equations.

Furthermore, there are four conditions on the type term ordering and the set of type assumptions, which are transferred from the polymorphic order-sorted theory:

- The ordered set of type terms must be unitary (definition ??)
- The closure of type term ordering must be well-formed.
- Each type term over the function symbols must have a least principal type (regularity, definition ??).
- For any result type of an overloaded function symbols there is a maximal principal arity of the function symbol (coregularity, definition ??).

These conditions are preconditions for the set of type assumptions A as well as conditions for the result of the type reconstruction algorithm. If these conditions are not fulfilled for the result then the OBJ-P-program is invalid.

The set of type assumptions A contains a type assumption $ite : (bool, x) \times (a, y) \times (a, z) \rightarrow a$ which stands for the `if_then_else_fi` construction. During the whole algorithm we assume that A and V_{old} are global variables, as they are not changed.

BEARBEITET START

Before we present the type reconstruction algorithm we give its main data structures:

The set of type assumption A is, similar to the set of type assumption of the type inference rules, a family, which maps class names (types) to the family of its typed fields and method names. In difference to the family of type assumptions of the inference rules, now intersection types are needed. For technical reasons, we map each method and its number of arguments (as its index) to an intersection type:

Irgendwie gab es noch eine Ueberlegung von wechselseitiger Rekursion in Klassen und Packages, wenn mehrere Klassen in einer Datei implementiert sind, dass dann die Methoden alle gemeinsam getypt werden. Vielleicht steht es auch noch woanders.

The types of the fields can not be a type schema, because a type variable in a type of a field means, that in instances of this class objects of all types can be instantiated. Therefore during

$$(m^{(n)} : (\forall \alpha_{1,1} \dots (\forall \alpha_{1,p_1} (\theta_{1,1} \times \dots \times \theta_{1,n} \rightarrow \theta_1)) \dots) \\ \wedge \dots \wedge \\ (\forall \alpha_{o,1} \dots (\forall \alpha_{o,p_o} (\theta_{o,1} \times \dots \times \theta_{o,n} \rightarrow \theta_o)) \dots) \in A_\theta)$$

where $A = (A_\theta)_\theta$ is a class

Let $T_{TC}(BTV)$ be the set of type terms over the type signature ($\mathbf{SType}_{TS}(BTV)$, TC) on declared Java 5.0 classes and \leq a type term ordering on $T_{TC}(BTV)$. Furthermore we need $\mathbf{FC}(\leq)$.

The Java 5.0 class which methods should be typed is given as an abstract syntax tree (cp. appendix A) of the following form:

```
class( ClassName(  $\tau$  ), extends(  $\tau'$  ),
      InstVarDecl(  $x_1, \theta_1$  ) , ... , InstVarDecl(  $x_o, \theta_o$  )
      Method(  $f_1$ ,
              ret $y_1$ ,
              (  $v_{1,1} : \theta_{1,1} \dots v_{1,k_1} : \theta_{1,k_1}$  )
              Block(  $B_1$  ) )
      ...
      Method(  $f_m$ ,
              ret $y_m$ ,
              (  $v_{m,1} : \theta_{m,1} \dots v_{m,k_m} : \theta_{m,k_m}$  )
              Block(  $B_m$  ) ) )
```

In the following we denote sets of type assumption as $A_{(f,n)}$ which is an abbreviation for the set of type assumptions A excluding the type assumptions for the function symbols f with n argument positions.

During the type reconstruction for each assumed type of the intersection type of a method there is a triple

$$(\sigma, \theta, V)$$

generated, where

- σ is a substitution which maps type variables of the originally assumed types to the types which are reconstructed for the type variables
- θ stands for the result type of the actual expression or statement list.
- V is set of type assumptions of
 - the actual method (for recursive calls)
 - the fields of the actual class
 - local variables

The following variables are considered as global in the whole type reconstruction algorithm.

INHALT VON A: !!! A_θ erbt alle Methoden m von $A_{\theta'}$ mit $(\theta, \theta') \in Inh$, wobei es in A_θ keine Methode gibt deren Argumenttypen mit denen von m unifizieren. !!!
Wahrscheinlich sollte eher festgelegt werden, dass keine Instanz eines Typs einer Methode Typ einer anderen Methode ist.

- act_cl contains the type τ of actual class $jclass$
- the type term ordering of the declared Java 5.0 classes: \leq
- the finite closure $\mathbf{FC}(\leq)$.
- A is the set of type assumptions for the already reconstructed type, indexed by the class-names.

HIER WEITERMACHEN:

- Alte obj-p Funktionen rausschmeien
- neue Funktionen erlutern und nochmals berprfen
- Beispiel durchrechnen
- Feststellen, ob \forall bei den Klassen ntig
- σ ueberpruefen, ob die Rueckagabe noetig und ob bei der Unifikation einheitlich Teile rausgeschissen werden.
- actual functions symbols durch methods of the actual class ersetzen - LocalOrFiledVar in LocalVar abndern. Die Unterscheidung ist in der abstrakten Syntax nach einem ersten Durchlauf der statischen Semantik klar.
- Beweis, dass der Algorithmus den Regeln entspricht

DAS GANZE NOCHMALSFUER WECHSELSEITIG REKURSIVES BEISPIEL AUSPROBIEREN!!!

Algorithm 5.15 (TRprog) The algorithm is the main procedure of the type reconstruction which calls the different sub-functions.

TRprog($A, jclass$) =

```
let
  NewTVar( $jclass$ ) = class( ClassName( $\tau$ , extends( $\tau'$ ) ),
                          InstVarDecl( $x_1, \theta_1$ ) , ... , InstVarDecl( $x_o, \theta_o$ )
                          Method( $f_1$ , ret $y_1$ , ( $v_{1,1} : \theta_{1,1} \dots v_{1,k_1} : \theta_{1,k_1}$ ),  $Bl_1$ )
                          ...
                          Method( $f_m$ , ret $y_m$ , ( $v_{m,1} : \theta_{m,1} \dots v_{m,k_m} : \theta_{m,k_m}$ ),  $Bl_m$ ) ) )

  (1)  $V_{fields\_methods} = \{ \text{this}.x_1 : \theta_1, \dots, \text{this}.x_o : \theta_o \}$ 
       $\cup \{ f_1 : \theta_{1,1} \times \dots \times \theta_{1,k_1} \rightarrow ret_1 \}$ 
       $\vdots$ 
       $\cup \{ f_m : \theta_{m,1} \times \dots \times \theta_{m,k_m} \rightarrow ret_m \}$ 

  (2)  $V_1 = \{ v_{1,1} : \theta_{1,1} \dots v_{1,k_1} : \theta_{1,k_1} \}$ 
       $\vdots$ 
       $V_m = \{ v_{m,1} : \theta_{m,1} \dots v_{m,k_m} : \theta_{m,k_m} \}$ 

  (3)  $\{ (\sigma_1, \bar{V}_1), \dots, (\sigma_l, \bar{V}_l) \} = \text{TRstart}((Bl_1, \dots, Bl_m), (V_1, \dots, V_m), V_{fields\_methods})$ 

in
let
   $A_\tau = \text{intersec} ( A \cup \text{clear} ( \bigcup_{i \leq l} \text{bind}(\text{TVar}(\tau), \bar{V}_i) ) )$ 
```

gehört das A tatsaechlich ins intersec?

```

in
   $A \cup A_\tau$ 
end
end

```

The algorithm **TRprog** first determines by the function **NewTVar** new type variables as types for parameters and return types of the methods where the type is not explicitly given. In explicitly given types the containing type variables are considered as type constants, because it is not allowed to substitute these type variables during the type reconstruction.

The set of type assumptions $V_{fields.methods}$ contains the type assumptions for all methods of the class *jclass*. The sets V_1, \dots, V_m contain the type assumptions for the parameters of the respective methods.

The function **TRstart** calls the type reconstruction of the blocks of the respective methods. The results are sets of pairs, which contain a type unifier σ_i and a set of type assumptions, with the reconstructed types. The type unifier is no longer needed at this point, because they have been already used to reconstruct the type assumptions.

The function **clear** removes the assumptions for the parameters and the local variables. Finally, the intersection types of the reconstructed types of the methods are built (**intersec**).

Sub-functions of the type reconstruction

The type reconstruction is started by the function **TRstart**.

Algorithm 5.16 (**TRstart**) The function **TRstart** controls the type reconstruction of the different methods.

```

TRstart( $(Bl_1, \dots, Bl_m), (\bar{V}_1, \dots, \bar{V}_m), V_{fields.methods}$ ) =
  let
     $\bar{V}_0 = \emptyset$ 
     $ret_0 = \{([], V_{fields.methods})\}$ 
  let-foreach  $1 \leq i \leq m$ :
     $ret_i =$ 
      let
         $\{(\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n)\} = \text{TRNextMeth}(\bar{V}_{i-1}, \bar{V}_i, ret_{i-1}, Bl_i)$ 
      let-foreach  $1 \leq j \leq n$ :
         $\theta_j = \text{RetType}(f_i, V_j)$ 
         $unify_j = \text{TUnify}_{\leq^*}(ty_j, \theta_j)$ 
      in
         $\bigcup_{1 \leq j \leq n} \{unify_j, \text{sub}(unify_j, V_j) \mid unify_j \in unify_j\}$ 
      end
    in
       $ret_m$ 
  end

```

Wird die Substitution σ nach der Rueckgabe ueberall auf die vrher berechneten Typen angewandt? vgl. TRmultiply

Man sollte ueberpruefen, ob es sinnvoll, dass alle Typen die groesser sind, hier hinzugefuegt werden.

The types of the blocks, which determine the respective methods are reconstructed step by step by the function **TRNextMeth**. During this reconstruction the types are adapted. After the type reconstruction of a block, the return type of the block and the return type of the method are unified. Finally, for each reconstructed triple (type unifier, return type, set of adpted type assumptions) of the previous method the types of the next method are reconstructed by the function **TRNextMeth**.

$$\text{TRNextMeth}(\bar{V}_{last}, \bar{V}_{next}, \{(\sigma_1, V_1), \dots, (\sigma_n, V_n)\}, Bl) = \bigcup_{1 \leq i \leq n} \text{TRStmt}(\sigma_i, V_i \setminus \{v : ty \mid (v : ty') \in \bar{V}_{last}\} \cup \text{sub}(\sigma_i, \bar{V}_{next}), Bl)$$

The **RetType** determines the return type of the given method in a set type assumptions.

```

RetType( $f_i, V$ ) =
  let
     $(f_i : \omega_1 \times \dots \times \omega_n \rightarrow \theta) \in V$ 
  in
     $\theta$ 
  end

```

In **TRNextMeth** the function **TRStmt** is called, which determines the type of a statement.

TRStmts

Now we describe the type reconstruction algorithm of the statements in different functions. The goal of these functions is to determine the result type of the corresponding method. In principle we have to differ two cases. Either a method has no return type, which is described by **void** or it has a type term as result type. In **Java 5.0** an existing result type is determined by a return statement, which stands always at the end of a list of statements. The type reconstruction algorithm make allowance for this by first assuming that the result type is **void**. If finally a return statement is given, the type of the respective expression is determined and the result type of the respective method is identified.

The structure of the statements in the abstract syntax is given such that for each statement there is one construct. A list of statements is subsumed by the **Block**-construct. From this construction follows the structure of the function **TRStmts** respectively **TRStmt**. The function **TRStmts** takes a list of statements and calls the function **TRStmt** for each statement. For each statement construct there is a function **TRStmt** which reconstructs the types for the respective statement.

Algorithm 5.17 (**TRStmts**) The function **TRStmts** calls for each element of a list of statements the function **TRStmt**.

First we consider the algorithm for a block with one statement. As **TRStmts** is called recursively, this is the recursion base, where *s* is the last element of the block.


```

TRStmts( $\bar{\sigma}, V, s :: [], V_{start}$ ) =
  let
     $\{(\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n)\} = \text{TRStmt}(\bar{\sigma}, V, s)$ 
  let-foreach  $1 \leq i \leq n$ :
     $V'_i = \{f : ty' \mid (f : ty') \in V_i \wedge \exists ty \in \text{Type}(T_{TC}(BTV)) : (f : ty) \in V_{start}\}$ 
  in
     $\{(\sigma_1, ty_1, V'_1), \dots, (\sigma_n, ty_n, V'_n)\}$ 
  end

```

In this case the local variables, which are added during the type reconstruction of the block are removed.

Now, we consider the case, that a block has more than one statement.

```

TRStmts( $\bar{\sigma}, V, s :: stmts, V_{start}$ ) =
  let
     $\{(\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n)\} = \text{TRStmt}(\bar{\sigma}, V, s)$ 
  in
    if  $ty_1 = \text{void}$  then
       $\bigcup_{1 \leq i \leq n} \text{TRStmts}(\sigma_i, V_i, stmts, V_{start})$ 
    else
      let-foreach  $1 \leq i \leq n$ :
         $\{(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,o_i}, ty_{i,o_i}, V_{i,o_i})\} = \text{TRStmts}(\sigma_i, V_i, stmts, V_{start})$ 
      in
        let-foreach  $1 \leq i \leq n, 1 \leq j \leq o_i$ :
           $(Mub_{i,j}, \sigma_{i,j}) = \text{TUnify}_{mub}(ty_i, ty_{i,j})$ 
        in
           $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq o_i}} \{(\sigma_{i,j}, \bar{\theta}, \text{sub}(\sigma, V_{i,j})) \mid \bar{\theta} \in Mub_{i,j}\}$ 
        end
      end
    end
  end

```

In this other case, where the block contains more than one element, first the function `TRStmt` is called. With the result of the function call, `TRStmts` is called recursively for the rest of the statement list. If the result type of the first statement is `void`, the result type of the block is determined by the rest list of statements. If one result type is `void` all result types are `void`, as an type unequal to `void` is induced by a `return`-statement. If the type of the first statement is unequal to `void`, the result types of the statement and the rest list of statements must be unified. This means that in this case the type of the rest list of statements must not be `void`, which corresponds to the original Java 5.0 specification.

In the following we consider the function `TRStmt`. We give for each statement an own algorithm, which determines the types of the respective statement. The different algorithms call each other mutually recursive.

Algorithm 5.18 (`TRStmt` for a block) The function `TRStmt` for a `Block` removes the `Block`-construct and calls the function `TRStmts` for the including statement list.

```

TRStmt( $\bar{\sigma}, V, \text{Block}(B)$ ) = TRStmts( $\bar{\sigma}, V, B, V$ )

```

Algorithm 5.19 (`TRStmt` for an if-statement) The function `TRStmt` for the if-statement reconstructs first the types for the conditional expression, second the types for the *then*-branch and last the types for the *else*-branch, if it exists.

First, we consider the case, that no *else*-branch exists:

```

TRStmt( $\bar{\sigma}, V, \text{If}(e, \text{Block}(B_1), \epsilon)$ ) =
  let
     $\{(\sigma'_1, \text{boolean}, V'_1), \dots, (\sigma'_m, \text{boolean}, V'_m)\} = \text{TRExp}(\bar{\sigma}, V, e)$ 
  in
     $\bigcup_{1 \leq k \leq m} \text{TRStmts}(\sigma'_k, V'_k, B_1, V'_k)$ 
  end

```

Now, we consider the case, that an *else*-branch exists:

```

TRStmt( $\bar{\sigma}, V, \text{If}(e, \text{Block}(B_1), \text{Block}(B_2))$ ) =
  let
     $\{(\sigma'_1, \text{boolean}, V'_1), \dots, (\sigma'_m, \text{boolean}, V'_m)\} = \text{TRExp}(\bar{\sigma}, V, e)$ 
     $\{(\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n)\} = \bigcup_{1 \leq k \leq m} \text{TRStmts}(\sigma'_k, V'_k, B_1, V'_k)$ 
  in
    let-foreach  $1 \leq i \leq n$ :
       $\{(\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,o_i}, ty_{i,o_i}, V_{i,o_i})\} = \text{TRStmts}(\sigma_i, V_i, B_2, V_i)$ 
    in
      if  $ty_1 \neq \text{void}$  and  $ty_{1,1} \neq \text{void}$  then
        let-foreach  $1 \leq i \leq n, 1 \leq j \leq o_i$ :
           $(Mub_{i,j}, \sigma_{i,j}) = \text{TUnify}_{mub}(ty_i, ty_{i,j})$ 
        in
           $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq o_i}} \{(\sigma_{i,j}, \bar{\theta}, \text{sub}(\sigma, V_{i,j})) \mid \bar{\theta} \in Mub_{i,j}\}$ 
        end
      else
        if  $ty_1 = \text{void}$  and  $ty_2 = \text{void}$  then
           $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq o_i}} \{(\sigma_{i,j}, \text{void}, \text{sub}(\sigma_{i,j}, V_{i,1}))\}$ 
        else
          if  $ty_1 = \text{void}$  then
             $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq o_i}} \{(\sigma_{i,j}, \sigma_{i,j}(ty_{i,j}), \text{sub}(\sigma_{i,j}, V_{i,1}))\}$ 
          else

```

$$\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq d_i}} \{ (\sigma_{i,j}, \sigma_{i,j}(ty_i), \mathbf{sub}(\sigma_{i,j}, V_{i,1})) \}$$

end
end

The type reconstruction of the conditional expression must have the result types **boolean**. All other reconstructed types are not considered.

With the possibly different result types of the conditional expression, the types of the *then*-branch are determined.

Foreach result of this reconstruction the types of the *else*-branch are reconstructed. It is necessary to know which reconstructed type of the *then*-branch belongs to which reconstructed type of the *else*-branch, as these types are unified for the result type of the if-statement.

If the *then*- and the *else*-branch have no result type (in the algorithm described as the type **void**) the result type of the if-statement is also **void**. It is enough to consider one type to determine if the type is **void**, as the type **void** is determined by the not existing of a **Return**-statement in each branch, respectively.

Algorithm 5.20 (TRStmt for an **Return**-statement) The result type of an **Return**-statement is determined by the type of the returned expression. Because of this the function **TRExp** is called for the expression, which reconstructs the types of the expression.

$\text{TRStmt}(\bar{\sigma}, V, \text{Return}(e)) = \text{TRExp}(\bar{\sigma}, V, e)$

Algorithm 5.21 (TRStmt for an **While**-statement) In the function **TRStmt** for the **While**-statement first the types for the ending condition are reconstructed. The result type must be **boolean**. All other reconstructed types are not considered.

Then, the function **TRStmts** is called for the list of statements, which is included in the corresponding block.

$\text{TRStmt}(\bar{\sigma}, V, \text{While}(e, \text{Block}(B))) =$
 let
 $\{ (\sigma'_1, \text{boolean}, V'_1), \dots, (\sigma'_n, \text{boolean}, V'_n) \} = \text{TRExp}(\bar{\sigma}, V, e)$
 in
 $\bigcup_{1 \leq i \leq n} \text{TRStmts}(\sigma'_i, V'_i, B, V'_i)$
 end

Algorithm 5.22 (TRStmt for an **LocalVarDecl**-statement) In the function **TRStmt** for the **LocalVarDecl**-construct two cases must be **UNTERSCHIEDEN**. In the first case the local variable is declared without type. In the second case the type is also declared.

$\text{TRStmt}(\bar{\sigma}, V, \text{LocalVarDecl}(v)) = \{ (\bar{\sigma}, \text{void}, V \cup \{v : \text{fresh}(a)\}) \}$

HIER MÜSSEN
MOGLICHER-
WEISE EINE
SCHON
VORHANDENE
VARIABLE
MIT DEN GLE-
ICHEN NAMEN
GELOESCHT
WERDEN.
Problem gelöst:
Lokale Variablen
drfen nicht
verschattet werden
und die fieldvars

In the first case for the declared variable a fresh type variable is assumed as its type. This typed local variable is added to the set of type assumptions V .

$\text{TRStmt}(\bar{\sigma}, V, \text{LocalVarDecl}(v, \theta)) = \{ (\bar{\sigma}, \text{void}, V \cup \{v : \theta\}) \}$

In the second case the typed local variable is added to the set of type assumptions V .

The next three algorithms reconstruct the types of statements, which are also expressions. In the algorithms the function **TRExp** is called, which reconstructs the types for the corresponding expressions.

Algorithm 5.23 (TRStmt for an **Assign**-statement) In the function **TRStmt** for the **Assign**-statement the types of the corresponding **Assign**-expression are determined. For each correct reconstructed type of the expression the substitutions and the sets of type assumptions are also given as result of the **Assign**-statement. Only, the result types are changed to **void**.

$\text{TRStmt}(\bar{\sigma}, V, \text{Assign}(e_1, e_2)) =$
 let
 $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, \text{Assign}(e_1, e_2))$
 in
 $\{ (\sigma_1, \text{void}, V_1), \dots, (\sigma_n, \text{void}, V_n) \}$
 end

Algorithm 5.24 (TRStmt for an **New**-statement)

$\text{TRStmt}(\bar{\sigma}, V, \text{New}(\theta, (t_1, \dots, t_n))) =$
 let
 $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, \text{New}(\theta, (t_1, \dots, t_n)))$
 in
 $\{ (\sigma_1, \text{void}, V_1), \dots, (\sigma_n, \text{void}, V_n) \}$
 end

Algorithm 5.25 (TRStmt for an **NewArray**-statement)

$\text{TRStmt}(\bar{\sigma}, V, \text{NewArray}(\theta, t)) =$
 let
 $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, \text{NewArray}(\theta, t))$
 in
 $\{ (\sigma_1, \text{void}, V_1), \dots, (\sigma_n, \text{void}, V_n) \}$
 end

Algorithm 5.26 (TRStmt for an **MethodCall**-statement)

$\text{TRStmt}(\bar{\sigma}, V, \text{MethodCall}(e, f(e_1, \dots, e_n))) =$
 let
 $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, \text{MethodCall}(e, f(e_1, \dots, e_n)))$
 in
 $\{ (\sigma_1, \text{void}, V_1), \dots, (\sigma_n, \text{void}, V_n) \}$
 end

TRExp

First, we give two functions **TRtuple** and **TRmultiply**, which are called from other functions to determine different types for a tuple of subterms. The function **TRtuple** calls the function **TRmultiply** for each subterm one after another. The function **TRmultiply** determines the different correct types of the next subterm and multiplies the result triple if there is more than one correct type for the subterm.

Algorithm 5.27 **TRtuple** calls step by step for each subterm **TRmultiply** and unites the results. $\text{TRtuple}(\text{result}, (t_1, \dots, t_n)) =$

```

let
  { result1, ..., resultl } = TRmultiply(result, t1)
in
  if n ≠ 1 then
    ⋃1 ≤ l ≤ n TRtuple(resulti, (t2 ... tn))
  else
    { result1, ..., resultl }
end

```

Algorithm 5.28 **TRmultiply** determines the different types of the subterms and multiplies the result triples, which represent respectively one type of the tuple of expressions.

```

TRmultiply(σ, (θ1 ... θm), V, t) =
let
  { result1, ..., resultl } = TRExp(σ, V, t)
in
  { (σ', (σ'(θ1) ... σ'(θm), θ'), V') | (σ', θ', V') = resulti, 1 ≤ i ≤ l }
end

```

Algorithm 5.29 (**TRExp** for **Assign**) In the function first the both expression are considered as a tuple. Their types are reconstructed one after another by the function **TRtuple**. After that the two result types are unified.

```

TRExp(σ̄, V, Assign(e1, e2)) =
let
  { (σ1, (ty1,1, ty1,2), V1), ..., (σn, (tyn,1, tyn,2), Vn) } = TRtuple(σ̄, ε, V), (e1, e2)
in
  let-foreach 1 ≤ i ≤ n:
    unify = TUnify≤*(tyi,2, tyi,1)
    subseti = { (σ ∘ σi, σ(tyi,1), sub(σ, V)) | σ ∈ unify }
  in
    (⋃1 ≤ i ≤ n subseti)
  end
end

```

PROBLEM:
TYPVARIABLEN
IN TYPEN VON
ATTRIBUTEN
UND LOKALEN
VARIABLEN
DUERFEN
BEI DER
UNIFIKATION
NICHT EINFACH
ERSETZT
WERDEN.
SIE MUESSEN
BLEIBEN, DA
JA DIE AT-
TRIBUTE FUER
ALLE BELIEBI-
GEN WERTE IN
METHODEN EIN-
SETZBAR SEIN
MUESSEN!!!!
ANSATZ: MAN
MACHT DIE
TYPVARIABLEN
VON TYPEN
DIE ERSETZT
WERDEN ZU
TYPE-SCHEMES
(MIT ALL-
QUANTOREN)
VERGLEICH
CLASS-REGEL in
ti.tex

auch Typvariablen
in Typen von
lokalen Variable
muessen als
konstante Typen
deklariert werden,
damit sie hier bei
der Unifikation
nicht einfach
ersetzt werden.

Algorithm 5.30 (**TRExp** for the **New**-operator) The application of the **New**-operator is equivalent to the application of a method. Therefore as for the method application, the function **TRMCallApp** is called (algorithm 5.32). The type of the receiver of the **New**-operator is given as the type argument of the **New**-operator itself.

$$\text{TRExp}(\bar{\sigma}, V, \text{New}(\theta, ())) = \{(\bar{\sigma}, \theta, V)\}$$

```

TRExp(σ̄, V, New(θ, (t1, ..., tn))) =
let
  { (σ̄1, (v̄1 ... v̄n)1, V1), ..., (σ̄l, (v̄1 ... v̄n)l, Vl) } = TRtuple(σ̄, ε, V), (t1, ..., tn)
in
  ⋃1 ≤ l ≤ n TRMCallApp(σ̄i, (θ(v̄1 ... v̄n)i), Vi, <init>θ)
end

```

Algorithm 5.31 (**TRExp** for the **NewArray**-operator) For a new array an **int**-expression must be evaluated to determine the length of the array. The array type is determined by the argument of the **NewArray**-operator.

```

TRExp(σ̄, V, NewArray(θ, t)) =
let
  (σ, ty, V') = TRExp(V, t)
in
  if ty ≠ int then
    ∅
  else
    { (σ, θ[], V') }
end

```

The next algorithm reconstructs types for method applications. This reconstruction is divided in the functions **TRExp** and **TRMCallApp**. The function **TRExp** is the main procedure which starts the recursive type reconstruction by calling **TRtuple** for the subterms t_1, \dots, t_n .

If for all subterms of a function application the type reconstruction is completed, **TRExp** calls the function **TRMCallApp**. In **TRMCallApp** the type assumptions of the function are unified with the result types of the subterms. In this function the overloading of methods is propagated. If there is more than one unifiable type of a method, another set of type assumptions V for the actual method is generated.

Algorithm 5.32 (**TRExp** for method applications) First we give the main procedure **TRExp**. Here we consider the receiver of the method application as a further argument.

Therefore, the function `TRtuple` is called with the receiver expression and the argument expressions of the method application.

```

TRExp( $\bar{\sigma}$ ,  $V$ , MethodCall( $re$ ,  $f(t_1, \dots, t_n)$ )) =
  let
    { $result_1, \dots, result_i$ } = TRtuple( $(\bar{\sigma}, \epsilon, V)$ , ( $re$ ,  $t_1, \dots, t_n$ ))
  in
     $\bigcup_{1 \leq i \leq n}$  TRMCallApp( $result_i$ ,  $f$ )
  end

```

The following function `TRMCallApp` determines the types of the application of a method.

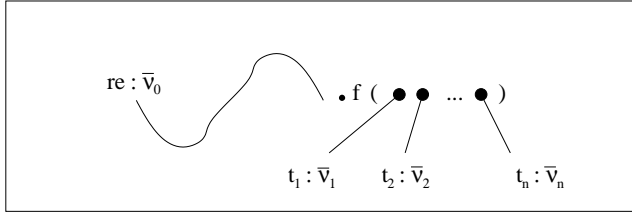


Figure 5.14: Type reconstruction of method applications

Algorithm 5.33 (`TRMCallApp`) For the function `TRMCallApp` let $\bar{\theta}_{j,1} \times \dots \times \bar{\theta}_{m,j} \rightarrow \bar{\theta}_j$ be the j 'th type assumption of the intersection type of the method f in the class $\bar{\theta}_{j,0}$. Then, `TRMCallApp` type unifies the type assumptions of the class $\bar{\theta}_{j,0}$ and of the arguments $\bar{\theta}_{j,1}, \dots, \bar{\theta}_{m,j}$ of the method f with the receiver type \bar{v}_0 and the result types $\bar{v}_1, \dots, \bar{v}_n$ of the subterms t_1, \dots, t_n (cf. figure 5.14). For each type assumption of f , where the type unifications do not fail, a new result triple is generated, which represents one type of the actual subterm. The type term $\sigma(\bar{\theta}_j)$ represents the result type of the term $f(t_1, \dots, t_n)$ and $\text{sub}(\sigma, V)$ consists of the new assumptions for the variables and the methods of the actual class. In the algorithm the not recursive application and the recursive application differ. The result of the not recursive application is denoted by $case_1$ and the result of the recursive application is denoted by $case_2$.

```

TRMCallApp( $(\bar{\sigma}, (\bar{v}_0 \bar{v}_1 \dots \bar{v}_m), V)$ ,  $f$ ) =
  let
     $case_1 =$ 
      let-foreach  $A_{\forall a_1, \dots, \forall a_n. C \langle a_1, \dots, a_n \rangle}$  with  $(f^{(m)} : ty) \in A_{\forall a_1, \dots, \forall a_n. C \langle a_1, \dots, a_n \rangle}$ :
         $A_{\forall a_1, \dots, \forall a_n. C \langle a_1, \dots, a_n \rangle} = A_{\forall a_1, \dots, \forall a_n. C \langle a_1, \dots, a_n \rangle} \setminus (f^{(m)})^3$ 

```

```

 $\cup \{ f^{(m)} : (\theta_{1,1}^C \times \dots \times \theta_{1,m}^C \rightarrow \theta_1^C) \wedge \dots \wedge (\theta_{o_C,1}^C \times \dots \times \theta_{o_C,m}^C \rightarrow \theta_{o_C}^C) \}$ 

 $res_C =$ 
  let-foreach  $1 \leq j \leq o_C$ :
     $(\bar{\theta}_{j,0}, \bar{\theta}_{j,1} \times \dots \times \bar{\theta}_{j,m} \rightarrow \bar{\theta}_j) = \text{fresh}((C \langle a_1, \dots, a_n \rangle), \theta_{j,1}^C \times \dots \times \theta_{j,m}^C \rightarrow \theta_j^C)$ 

     $unify_j = \text{TUnify}_{\leq^*}((\bar{v}_0, \bar{v}_1, \dots, \bar{v}_m), (\bar{\theta}_{j,0}, \bar{\theta}_{j,1}, \dots, \bar{\theta}_{j,m}))$ 
     $subset_j = \{ (\sigma \circ \bar{\sigma}, \sigma(\bar{\theta}_{j,0}), (\sigma(\bar{\theta}_{j,1}) \dots \sigma(\bar{\theta}_{j,m})), \sigma(\bar{\theta}_j), \text{sub}(\sigma, V)) \mid \sigma \in unify_j \}$ 
  in
     $(\bigcup_{1 \leq j \leq o_C} subset_j)$ 
  end
in
   $(\bigcup_C ret_C)$ 
end

 $case_2 =$ 
  if  $(f^{(m)} : ty) \in V$  then
    let
       $V = V \setminus (f^{(m)}) \cup \{ f^{(m)} : \theta_1 \times \dots \times \theta_m \rightarrow \theta \}$ 
       $unify = \text{TUnify}_{\leq^*}((\bar{v}_0 \bar{v}_1, \dots, \bar{v}_m), (act.cl, \theta_1, \dots, \theta_m))$ 
    in
       $\{ ((\sigma \circ \bar{\sigma}), \sigma(\bar{\theta}_0), (\sigma(\bar{\theta}_1) \dots \sigma(\bar{\theta}_m)), \sigma(\theta), \text{sub}(\sigma, V)) \mid \sigma \in unify \}$ 
    end
  else
     $\emptyset$ 
  end
in
  let
     $clres = \{ \theta_0 \mid (\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \in (case_1 \cup case_2) \text{ and } \sigma(\bar{v}_0) = \theta_0 \}$ 
  in
    let-foreach  $\theta_0 \in clres$ :
       $arity_{\theta_0} = \{ (\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \mid (\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \in (case_1 \cup case_2) \}$ 
       $res_{\theta_0} = \{ (\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \mid (\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \in arity_{\theta_0} \text{ and } \nexists (\sigma', \theta_0, (\theta'_1 \dots \theta'_m), \theta', V'') \in arity_{\theta_0} \text{ with } (\theta'_1 \dots \theta'_m) \neq (\theta_1 \dots \theta_m) \text{ and there is a lower bound } (\bar{\theta}_1 \dots \bar{\theta}_m) \text{ of } (\theta_1 \dots \theta_m) \text{ and } (\theta'_1 \dots \theta'_m) \}$ 
    in
      HIER WEITERMACHEN erledigt 050908
    end
  end

```

Hier knnte man die Unifikation des receivers rausloesen und immer das kleinste Ergebnis nehmen. Problem case2, muss mit einbezogen werden. Deshalb wird es erst am Schluss rausgefiltert. Ist so etwas ineffizienter. Es ist noch unklar, ob man V_{old} benoetigt? ggfs. einkommentieren

Wie stellt man bei der Rekursion fest, dass es sich tatschlich um einen rekursiven Aufruf handelt: Man muss wissen, dass der Receiver gleich der aktuellen Klasse ist.

REKURSIVE AUFRUFE MIT INSTANZEN DER DERZEITIGEN KLASSE SIND NICHT MOEGLICH. DAHER KANN STATT $act.cl$ KEINE INSTANZ VON $act.cl$ STEHEN

NOCHMAL'S UEBERPRUEFEN, OB \bar{v}_0 BEI DER UNIFIKATION KLEINER WERDEN DARF, WAS HEISST DAS DANN????

³ $A_{\theta} \setminus (f^{(m)})$ is an abbreviation which stands for the set of type assumptions A_{θ} without the type assumptions for the method f with m arguments.

- DIE AUSWAHL DER METHODEN MUSS SO ERFOLGEN, DASS DIE KLEINSTE KLASSE GEWAHLT WIRD (vergl. JAVA.EXAMPLES/ClassSelector.java) erledigt 050908. - WAS PASSIERT WENN ES EIN ECHTES INFIMUM IST? - KEIN PROBLEM WENN $act.cl$ NICHT DIE KLEINSTE KLASSE, SONST NICHT KLAR, $case_1$ DARF NICHT ANGEWANDT WERDEN. erledigt 050908.

$f(t_1 \dots t_n)$ in allen Beispielaufrufen von `TRMCallApp` nach f aendern.

```

    {  $(\sigma, \theta, V') \mid \theta_0 \in clres$  and  $(\sigma, \theta_0, (\theta_1 \dots \theta_m), \theta, V') \in res_{\theta_0}$  }
  end
end
end

```

Now, we present an example, which shows how the correct typing of a method application is selected.

Example 5.34 Let the following program as an extension from the program of example 5.7 be given:

```

class M<A> {
  M<A> m(Stack<A> v) {
    return new M<A>();
  }

  M<A> p(Vector<A> v, Stack<A> w) {
    return new M<A>();
  }
}

class N<A> extends M<A> {

  N<A> m(Vector<A> v) {
    return new N<A>();
  }

  N<A> p(Stack<A> w, Vector<A> v) {
    return new N<A>();
  }
}

class O<A> {
  O<A> m(Vector<A> v) {
    return new O<A>();
  }

  O<A> p(Vector<A> v, Stack<A> w) {
    return new O<A>();
  }
}

```

We consider now the following two applications of *TRMCallApp*:

- *TRMCallApp*($\bar{\sigma}$, (*N*<Integer>, Stack<Integer>), *V*, *m*)

```

-  $\sigma_1 = \{ A \mapsto \text{Integer} \}$ 
-  $\sigma_2 = \{ A \mapsto \text{Integer} \}$ 
-  $\sigma_3 = \{ A \mapsto \text{Integer} \}$ 
-  $case_1 \cup case_2 =$ 
  {  $(\sigma_1 \circ \bar{\sigma}, M<\text{Integer}>, \text{Stack}<\text{Integer}>, M<\text{Integer}>, \text{sub}(\sigma_1, V))$ ,
     $(\sigma_2 \circ \bar{\sigma}, N<\text{Integer}>, \text{Stack}<\text{Integer}>, M<\text{Integer}>, \text{sub}(\sigma_2, V))$ ,
     $(\sigma_3 \circ \bar{\sigma}, N<\text{Integer}>, \text{Vector}<\text{Integer}>, N<\text{Integer}>, \text{sub}(\sigma_3, V))$  }
-  $clres = \{ N<\text{Integer}> \}$ 
-  $arity_{N<\text{Integer}>} = \{ (\text{Stack}<\text{Integer}>), (\text{Vector}<\text{Integer}>) \}$ 
-  $res_{N<\text{Integer}>} = \{ (\text{Stack}<\text{Integer}>) \}$ 

```

The result of the application is then

$$\{ (\sigma_2 \circ \sigma, M<\text{Integer}>, \text{sub}(\sigma_2, V)) \}.$$

- *TRMCallApp*($\bar{\sigma}$, (*B*, Stack<Integer>, Stack<Integer>), *V*, *p*)

```

-  $\sigma_1 = \{ A \mapsto \text{Integer}, B \mapsto M<\text{Integer}> \}$ 
-  $\sigma_2 = \{ A \mapsto \text{Integer}, B \mapsto N<\text{Integer}> \}$ 
-  $\sigma_3 = \{ A \mapsto \text{Integer}, B \mapsto N<\text{Integer}> \}$ 
-  $\sigma_4 = \{ A \mapsto \text{Integer}, B \mapsto O<\text{Integer}> \}$ 
-  $case_1 \cup case_2 =$ 
  {  $(\sigma_1 \circ \bar{\sigma}, M<\text{Integer}>, (\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>), M<\text{Integer}>, \text{sub}(\sigma_1, V))$ ,
     $(\sigma_2 \circ \bar{\sigma}, N<\text{Integer}>, (\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>), M<\text{Integer}>, \text{sub}(\sigma_2, V))$ ,
     $(\sigma_3 \circ \bar{\sigma}, N<\text{Integer}>, (\text{Stack}<\text{Integer}>, \text{Vector}<\text{Integer}>), N<\text{Integer}>, \text{sub}(\sigma_3, V))$ ,
     $(\sigma_4 \circ \bar{\sigma}, O<\text{Integer}>, (\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>), O<\text{Integer}>, \text{sub}(\sigma_4, V))$  }
-  $clres = \{ N<\text{Integer}>, O<\text{Integer}> \}$ 
-  $arity_{N<\text{Integer}>} =$ 
  {  $(\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>), (\text{Stack}<\text{Integer}>, \text{Vector}<\text{Integer}>) \}$ 
-  $arity_{O<\text{Integer}>} = \{ (\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>) \}$ 
-  $res_{N<\text{Integer}>} = \emptyset$ , (as  $(\text{Stack}<\text{Integer}>, \text{Stack}<\text{Integer}>)$  is a lower bound of
   $(\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>)$  and  $(\text{Stack}<\text{Integer}>, \text{Vector}<\text{Integer}>)$ ).
-  $res_{O<\text{Integer}>} = \{ (\text{Vector}<\text{Integer}>, \text{Stack}<\text{Integer}>) \}$ 

```

The result of the application is then

$$\{ (\sigma_4 \circ \sigma, O<\text{Integer}>, \text{sub}(\sigma_4, V)) \}.$$

If we change one of the argument types we get two results, respectively:

- *TRMCallApp*($\bar{\sigma}$, (*B*, Vector<Integer>, Stack<Integer>), *V*, *p*)

The result of this application is then

$$\{ (\sigma_2 \circ \sigma, M<\text{Integer}>, \text{sub}(\sigma_2, V)), (\sigma_4 \circ \sigma, O<\text{Integer}>, \text{sub}(\sigma_4, V)) \}.$$

– TRMCallApp($\bar{\sigma}$, (B, Stack<Integer>, Vector<Integer>), V, p)

The result of this application is then

$$\{ (\sigma_3 \circ \sigma, N<\text{Integer}>, \text{sub}(\sigma_3, V)), (\sigma_4 \circ \sigma, 0<\text{Integer}>, \text{sub}(\sigma_4, V)) \}.$$

Algorithm 5.35 (TRExp for this) The **this** command means the actual class. Therefore the type of the term is the argument *act_cl*.

$$\text{TRExp}(\bar{\sigma}, V, \text{this}) = \{ (\bar{\sigma}, \text{act_cl}, V) \}$$

Algorithm 5.36 (TRExp for super) The **super** command means the direct super class of the actual class.

$$\begin{aligned} \text{TRExp}(\bar{\sigma}, V, \text{super}) = \\ \text{let} \\ (act_cl, \tau') \in \leq \\ \text{in} \\ \{ (\bar{\sigma}, \tau', V) \} \\ \text{end} \end{aligned}$$

Algorithm 5.37 (TRExp for local or field variables) There are two possibilities. The first is that the variable is a local variable. A local variable is either an argument of the method or a local variable of the actual block. The type of a local variable is given as the type assumption of the variable in the set *V*. Therefore the substitution $\bar{\sigma}$ is not extended.

The second possibility is that the variable is a field. This is given if the variable is no local variable. In this case the variable is qualified by **this** in the set *V*. Then the type is also given as the type assumption of the variable in the set *V*. The substitution $\bar{\sigma}$ is also not extended.

$$\begin{aligned} \text{TRExp}(\bar{\sigma}, V, \text{LocalOrFiledVar}(x)) = \\ \text{if } (x : \theta) \in V \text{ then} \\ \{ (\bar{\sigma}, \theta, V) \} \\ \text{else} \\ \text{if } (\text{this}.x : \theta) \in V \text{ then} \\ \{ (\bar{\sigma}, \theta, V) \} \end{aligned}$$

Algorithm 5.38 (TRExp for instance variables) The type reconstruction for instance variables is divided two functions. In the first, the possibly different types of the receiver *re* are determined.

$$\begin{aligned} \text{TRExp}(\bar{\sigma}, V, \text{InstVar}(re, v)) = \\ \text{let} \\ \{ result_1, \dots, result_l \} = \text{TRExp}(\bar{\sigma}, V, re) \end{aligned}$$

in
 $\bigcup_{i \leq l} \text{TRInstVar}(result_i, v)$
 end

For all results of this, the function **TRInstVar** is called, where the type of the respective instance variable *v* is determined.

$$\begin{aligned} \text{TRInstVar}(\sigma, \nu, V, v) = \\ \text{let-foreach } C<a_1, \dots, a_n> \text{ with } (((v^{(0)} : \theta_C) \in A_{\forall a_1, \dots, \forall a_n. C<a_1, \dots, a_n>} \text{ or } \\ (C<a_1, \dots, a_n> = \text{act_cl} \text{ and } (\text{this}.v^{(0)} : \theta_C) \in V)) : \\ \theta_0 = \text{fresh}(C<a_1, \dots, a_n>) \\ \sigma_C = \text{TUnify}_{\leq^*}(\nu, \theta_0) \\ \text{in} \\ \bigcup_{\substack{C \\ \sigma_C}} \{ (\sigma_C \circ \sigma, \sigma_C(\theta_C), \text{sub}(\sigma_C, V)) \} \\ \text{end} \end{aligned}$$

Ist das act.cl
und das this hier
richtig? eher
LocalOrFiled-
Var???

BRAUCHT θ IN
DER NAECH-
STEN ZEILE
 σ ?

The next algorithm determines the type of an array access.

Algorithm 5.39 (TRExp for the array access) In this algorithm first the types of the index are determined. Only the if the result type is **int**, the result triples are considered further on. For these triples the types of the array are determined. If the type of the array is an array type then a result type of the whole expression is given.

$$\begin{aligned} \text{TRExp}(\bar{\sigma}, V, \text{ArrayAcc}(e, index)) = \\ \text{let} \\ \{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, index) \\ \text{in} \\ \text{let-foreach } 1 \leq i \leq n \text{ with } ty_i = \text{int}: \\ \{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e) \\ \text{in} \\ \bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \theta, V_{i,j}) \mid ty_{i,j} = \theta \square \} \\ \text{end} \\ \text{end} \end{aligned}$$

Algorithm 5.40 (TRExp for Literals) For the literals of the types **int**, **boolean**, and **char** the corresponding types are the result. For the literal **Null** any type is possible. Therefore a new type variable is generated as its type.

$$\text{TRExp}(\bar{\sigma}, V, \text{IntLiteral}(n)) = \{ (\bar{\sigma}, \text{int}, V) \}$$

$$\text{TRExp}(\bar{\sigma}, V, \text{BoolLiteral}(b)) = \{ (\bar{\sigma}, \text{boolean}, V) \}$$

$$\text{TRExp}(\bar{\sigma}, V, \text{CharLiteral}(c)) = \{ (\bar{\sigma}, \text{char}, V) \}$$

$\text{TRExp}(\bar{\sigma}, V, \text{Null}) = \{ (\bar{\sigma}, \text{fresh}(a), V) \}$

Algorithm 5.41 (TRExp for UnaryMinus) The argument of the `UnaryMinus` must be `int`. Therefore only the result triples of this type are considered.

```

TRExp( $\bar{\sigma}, V, \text{UnaryMinus}(e)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e$ )
  in
    { ( $\sigma_i, \text{int}, V_i$ ) |  $ty_i = \text{int}$  }
  end

```

Algorithm 5.42 (TRExp for Not) The argument of the `Not` must be `boolean`. Therefore only the result triples of this type are considered.

```

TRExp( $\bar{\sigma}, V, \text{Not}(e)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e$ )
  in
    { ( $\sigma_i, \text{boolean}, V_i$ ) |  $ty_i = \text{boolean}$  }
  end

```

Algorithm 5.43 (TRExp for Cast) The `cast`-expression transforms the type of an expression to the given type. Therefore the result type is the given type.

```

TRExp( $\bar{\sigma}, V, \text{Cast}(\theta, e)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e_1$ )
  in
    { ( $\sigma_1, \theta, V_1$ ), ..., ( $\sigma_n, \theta, V_n$ ) }
  end

```

The following algorithms for the addition (5.44), the subtraction (5.45), the multiplication (5.46), the division (5.47), and the modulo function (5.48) are of the same structure. The two arguments must have the type `int`. The result type is then also `int`. All other types are not considered.

Algorithm 5.44 (TRExp for Add)

```

TRExp( $\bar{\sigma}, V, \text{Add}(e_1, e_2)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e_1$ )
  in

```

```

    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
      { ( $\sigma_{i,1}, ty_{i,1}, V_{i,1}$ ), ..., ( $\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}$ ) } = TRExp( $\sigma_i, V_i, e_2$ )
    in
       $\bigcup_{ty_i = \text{int}}$  { ( $\sigma_{i,j}, \text{int}, V_{i,j}$ ) |  $ty_{i,j} = \text{int}$  }
    end
  end

```

Algorithm 5.45 (TRExp for Minus)

```

TRExp( $\bar{\sigma}, V, \text{Minus}(e_1, e_2)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e_1$ )
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
      { ( $\sigma_{i,1}, ty_{i,1}, V_{i,1}$ ), ..., ( $\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}$ ) } = TRExp( $\sigma_i, V_i, e_2$ )
    in
       $\bigcup_{ty_i = \text{int}}$  { ( $\sigma_{i,j}, \text{int}, V_{i,j}$ ) |  $ty_{i,j} = \text{int}$  }
    end
  end

```

Algorithm 5.46 (TRExp for Mul)

```

TRExp( $\bar{\sigma}, V, \text{Mul}(e_1, e_2)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e_1$ )
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
      { ( $\sigma_{i,1}, ty_{i,1}, V_{i,1}$ ), ..., ( $\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}$ ) } = TRExp( $\sigma_i, V_i, e_2$ )
    in
       $\bigcup_{ty_i = \text{int}}$  { ( $\sigma_{i,j}, \text{int}, V_{i,j}$ ) |  $ty_{i,j} = \text{int}$  }
    end
  end

```

Algorithm 5.47 (TRExp for Div)

```

TRExp( $\bar{\sigma}, V, \text{Div}(e_1, e_2)$ ) =
  let
    { ( $\sigma_1, ty_1, V_1$ ), ..., ( $\sigma_n, ty_n, V_n$ ) } = TRExp( $\bar{\sigma}, V, e_1$ )
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
      { ( $\sigma_{i,1}, ty_{i,1}, V_{i,1}$ ), ..., ( $\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}$ ) } = TRExp( $\sigma_i, V_i, e_2$ )
    in

```

```

       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{int}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

Algorithm 5.48 (TRExp for Mod)

```

TRExp( $\bar{\sigma}$ ,  $V$ , Mod( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
       $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
    in
       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{int}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

The following algorithms for Less (5.49), LessEq (5.50), Greater (5.51), and GreaterEq (5.52) are very similar to the above algorithms. Only the result type is not int, but boolean.

Algorithm 5.49 (TRExp for Less)

```

TRExp( $\bar{\sigma}$ ,  $V$ , Less( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
       $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
    in
       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{boolean}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

Algorithm 5.50 (TRExp for LessEq)

```

TRExp( $\bar{\sigma}$ ,  $V$ , LessEq( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
       $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 

```

```

    in
       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{boolean}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

Algorithm 5.51 (TRExp for Greater)

```

TRExp( $\bar{\sigma}$ ,  $V$ , Greater( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
       $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
    in
       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{boolean}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

Algorithm 5.52 (TRExp for GreaterEq)

```

TRExp( $\bar{\sigma}$ ,  $V$ , GreaterEq( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in
    let-foreach  $1 \leq i \leq n$  with  $ty_i = \text{int}$ :
       $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
    in
       $\bigcup_{ty_i = \text{int}} \{ (\sigma_{i,j}, \text{boolean}, V_{i,j}) \mid ty_{i,j} = \text{int} \}$ 
    end
  end
end

```

The next two algorithms Equal (5.53) and NotEqual (5.54) allows as argument type each type. The only condition is, that both arguments have the same type. Therefore the result types of both arguments are unified. If they are successfully unified, the result type is boolean.

Algorithm 5.53 (TRExp for Equal)

```

TRExp( $\bar{\sigma}$ ,  $V$ , Equal( $e_1, e_2$ )) =
  let
     $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
  in

```

```

let-foreach  $1 \leq i \leq n$ :
   $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
let-foreach  $1 \leq i \leq n, 1 \leq j \leq m_i$ :
   $unify_{i,j} = \text{TUnify}_{\leq^*}(ty_{i,j}, ty_{i,j}) \cup \text{TUnify}_{\leq^*}(ty_{i,j}, ty_i)$ 
in
   $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m_i}} \{ (\sigma, \text{boolean}, \text{sub}(\sigma, V_{i,j})) \mid \sigma \in unify_{i,j} \}$ 
end
end

```

Algorithm 5.54 (TRExp for NEq)

```

TRExp( $\bar{\sigma}, V, \text{NEq}(e_1, e_2)$ ) =
let
   $\{ (\sigma_1, ty_1, V_1), \dots, (\sigma_n, ty_n, V_n) \} = \text{TRExp}(\bar{\sigma}, V, e_1)$ 
in
  let-foreach  $1 \leq i \leq n$ :
     $\{ (\sigma_{i,1}, ty_{i,1}, V_{i,1}), \dots, (\sigma_{i,m_i}, ty_{i,m_i}, V_{i,m_i}) \} = \text{TRExp}(\sigma_i, V_i, e_2)$ 
  let-foreach  $1 \leq i \leq n, 1 \leq j \leq m_i$ :
     $unify_{i,j} = \text{TUnify}_{\leq^*}(ty_i, ty_{i,j}) \cup \text{TUnify}_{\leq^*}(ty_{i,j}, ty_i)$ 
  in
     $\bigcup_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m_i}} \{ (\sigma, \text{boolean}, \text{sub}(\sigma, V_{i,j})) \mid \sigma \in unify_{i,j} \}$ 
  end
end

```

Algorithm 5.55 (TUnify) The function **TUnify** is the type unification algorithm, which is presented in section 4.2. Here we describe the adaption of the call of **TUnify** in the type reconstruction algorithm to the presentation in algorithm 4.5.

$$\text{TUnify}_{\leq^*}((ty_{1,1} \dots ty_{1,n}), (ty_{2,1} \dots ty_{2,n}))$$

calls the algorithm 4.5 with the input

$$\{ (ty_{1,1} \leq ty_{2,1}), \dots, (ty_{1,n} \leq ty_{2,n}) \}.$$

The set of type variables TV is given as the type variables which are generate by **fresh** and in **NewTVar**.

$$\text{TUnify}_{mup}(ty_1, ty_2)$$

calls algorithm 4.29.

Es gibt Namenskonflikte zwischen der Funktion **subst** und der Menge der Substitutionen $\text{subst}(T_\Theta(TV))$.
Welchen Sinn haben die res_i – Funktionen?, braucht man sie noch? Wahrscheinlich waren sie dazu da die Typen der linken und rechten Gleichungsseite aufeinander abzustimmen.

Algorithm 5.56 (**sub**) The function **sub**(σ, V) substitutes the type variables of the types of the variables and the methods by σ in the set of type assumptions V .

```

sub( $\sigma, V$ ) =
let
   $V = \{ f_1 : ty_{1,1} \times \dots \times ty_{1,k_1} \rightarrow rty_1, \dots, f_n : ty_{n,1} \times \dots \times ty_{n,k_n} \rightarrow rty_n \}$ 
   $\cup \{ v_1 \mapsto \nu_1, \dots, v_p \mapsto \nu_p \}$ 
in
   $\{ f_1 : \sigma(ty_{1,1}) \times \dots \times \sigma(ty_{1,k_1}) \rightarrow \sigma(rty_1), \dots, f_n : \sigma(ty_{n,1}) \times \dots \times \sigma(ty_{n,k_n}) \rightarrow \sigma(rty_n) \}$ 
   $\cup \{ v_1 \mapsto \sigma(\nu_1), \dots, v_p \mapsto \sigma(\nu_p) \}$ 
end

```

Algorithm 5.57 (**bind**) The function **bind** takes the free variables of a type assumption of a method, and substitute them by the parameters of the respective class.

```

bind( $Para, V$ ) =
let
   $V = \{ me : ty_1 \times \dots \times ty_m \rightarrow rty \} \cup V'$ 
   $Var = \text{TVar}(\{ ty_1, \dots, ty_m, rty \})$ 
   $subst = \{ \sigma \mid \sigma : Var \rightarrow Para \text{ is a function} \}$ 
in
   $\{ me : \bigwedge_{\sigma \in subst} \sigma(ty_1) \times \dots \times \sigma(ty_m) \rightarrow \sigma(rty) \} \cup V'$ 
end

```

Now we consider correctness and completeness of the type reconstruction algorithm in corelation to the type inference system of section 5.4.

Theorem 5.58 (Most principal type property of TRprog) *The type reconstruction algorithm TRprog determines for the methods of a Java 5.0 class representants of the respective most principal types.*

bind
AUSWIRKUN-
GEN BETRA-
CHTEN

Proof: The proof is done in three steps. First we have to prove that each type, which is determined by the type reconstruction algorithm of a method is also derivable by the type inference system (correctness). Second we have to prove that if a type of a method is derivable by the type inference system, the type is also determined by the algorithm (completeness). As it is proved in theorem 5.10, that there is a principal type derivable by the type inference system, in a third step, we have to show that the type reconstruction algorithm do not determine any type which is not principal.

Correctness

For a class declaration $jclass$ $\text{TRprog}(A, jclass)$ is called. Let

$$\begin{aligned} \text{NewTVar}(jclass) = & \text{Class}(\text{ClassName}(\tau), \text{extends}(\tau')) \\ & \text{InstVarDecl}(x_1, \tau_1), \dots, \text{InstVarDecl}(x_n, \tau_n) \\ & \text{Method}(f_1, \theta_1, (v_{1,1} : \theta_{1,1}, \dots, v_{1,m_1} : \theta_{1,k_1}), \text{Block}(B_1)) \\ & \dots, \\ & \text{Method}(f_m, \theta_m, (v_{m,1} : \theta_{m,1}, \dots, v_{m,m_m} : \theta_{m,k_m}), \text{Block}(B_m)) \end{aligned}$$

Furthermore let $\text{TRprog}(A, jclass) = A \cup A_\tau$.

Then it is to prove, that it holds under the assumption that $p \blacktriangleright O$ with $O = \{A_{\text{gen}(\theta)} \mid A_\theta \in A\}$:

$$p \cup \{jclass\} \blacktriangleright O \cup O_{\text{gen}(\tau)} \text{ with } O_{\text{gen}(\tau)} = A_\tau.$$

Let

$$\begin{aligned} & (f_1 : (\theta_{1,1}^1 \times \dots \times \theta_{1,k_1}^1 \rightarrow \theta_1^1) \\ & \quad \wedge \dots \wedge \\ & \quad (\theta_{1,1}^{o_1} \times \dots \times \theta_{1,k_1}^{o_1} \rightarrow \theta_1^{o_1})) \\ & \dots, \\ & (f_m : (\theta_{m,1}^1 \times \dots \times \theta_{m,k_m}^1 \rightarrow \theta_m^1) \\ & \quad \wedge \dots \wedge \\ & \quad (\theta_{m,1}^{o_m} \times \dots \times \theta_{m,k_m}^{o_m} \rightarrow \theta_m^{o_m})) \in A_\tau \end{aligned}$$

Then it is to prove that for each $1 \leq i \leq m$, $1 \leq j \leq o_i$ there is a \overline{O}_τ with

$$\begin{aligned} \overline{O}_\tau = & \{f_i : (\theta_{i,1}^j \times \dots \times \theta_{i,k_i}^j \rightarrow \theta_i^j)\} \\ & \cup \{f_{i'} : (\theta_{i',1}^{j'} \times \dots \times \theta_{i',k_{i'}}^{j'} \rightarrow \theta_{i'}^{j'}) \mid (i' < i) \vee (i < i' \leq m), 1 \leq j' \leq o_{i'}\} \end{aligned}$$

and

$$p \cup jclass \blacktriangleright O \cup \{\text{gen}(\tau) \mapsto \overline{O}_\tau \cup O_{\text{field}}\}$$

where $O_{\text{field}} = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$.

This is equivalent to the following:

With the function calls of TRstart , TRNextMeth , and TRStmt for a block for

$$V_i = \{v_{i,1} : \theta_{i,1}^j, \dots, v_{i,k_i} : \theta_{i,k_i}^j\}$$

there is a substitution σ_i and a type θ_i with

$$(\sigma_i, \theta_i, \overline{O}_\tau \cup O_{\text{field}} \cup V_i) \in \text{TRStmts}([], \overline{O}_\tau \cup O_{\text{field}} \cup V_i, B_i, \overline{O}_\tau \cup O_{\text{field}} \cup V_i)$$

and for $(i' < i) \vee (i < i' \leq m)$ with

$$V_{i'} = \{v_{i',1} : \theta_{i',1}^{j'}, \dots, v_{i',k_{i'}} : \theta_{i',k_{i'}}^{j'}\}$$

there are substitutions $\sigma_{i'}$ and types $\theta_{i'}$ with

$$(\sigma_{i'}, \theta_{i'}, \overline{O}_\tau \cup O_{\text{field}} \cup V_{i'}) \in \text{TRStmts}([], \overline{O}_\tau \cup O_{\text{field}} \cup V_{i'}, B_{i'}, \overline{O}_\tau \cup O_{\text{field}} \cup V_{i'})$$

In this prove we take as arguments for TRStmts the set of type assumptions V_i , respectively $V_{i'}$, which contains the results of the algorithm. We do this as the prove becomes simpler. This is correct as it holds for any set of type assumptions V' :

$$\text{From } (\sigma, \theta, V) = \text{TRStmts}([], V', B, V') \text{ follows } (\sigma, \theta, V) = \text{TRStmts}([], V, B, V)$$

This follows by induction, as in every application the types of a former method can only become less general.

With the **Class** rule it is then to prove that it holds:

$$\begin{aligned} & (O \cup \{\tau \mapsto (\overline{O}_\tau \cup O_{\text{field}})\} \cup \\ & \quad \{\langle \text{local} \rangle \mapsto (O_{\text{field}} \cup V_i)\}, \tau, \tau') \triangleright_{\text{Stmt}} \text{Block}(B_i) : \theta_i, (\theta_i \leq^* \theta_i^j) \end{aligned}$$

and for $(i' < i) \vee (i < i' \leq m)$

$$\begin{aligned} & (O \cup \{\tau \mapsto (\overline{O}_\tau \cup O_{\text{field}})\} \cup \\ & \quad \{\langle \text{local} \rangle \mapsto (O_{\text{field}} \cup V_{i'})\}, \tau, \tau') \triangleright_{\text{Stmt}} \text{Block}(B_{i'}) : \theta_{i'}, (\theta_{i'} \leq^* \theta_{i'}^{j'}) \end{aligned}$$

This prove is done by an induction over the list of block statements:

Induction start: The different blocks contain of one statement: We assume that $B_i = s_i :: []$.

With the function TRStmts there is a \overline{V}_i with $V_i \subseteq \overline{V}_i$ and

$$(\sigma_i, \theta_i, \overline{O}_\tau \cup O_{\text{field}} \cup \overline{V}_i) \in \text{TRStmt}((\sigma_i, \overline{O}_\tau \cup O_{\text{field}} \cup \overline{V}_i, s_i)$$

and for $(i' < i) \vee (i < i' \leq m)$ there are $\overline{V}_{i'}$ with $V_{i'} \subseteq \overline{V}_{i'}$ and

$$(\sigma_{i'}, \theta_{i'}, \overline{O}_\tau \cup O_{\text{field}} \cup \overline{V}_{i'}) \in \text{TRStmt}(\sigma_{i'}, \overline{O}_\tau \cup O_{\text{field}} \cup \overline{V}_{i'}, s_{i'})$$

Then, by the statements rules follows, that it is to prove:

$$\begin{aligned} & (O \cup \{\tau \mapsto (\overline{O}_\tau \cup O_{\text{field}})\} \cup \\ & \quad \{\langle \text{local} \rangle \mapsto (O_{\text{field}} \cup \overline{V}_i)\}, \tau, \tau') \triangleright_{\text{Stmt}} s_i : \theta_i \end{aligned}$$

and for $(i' < i) \vee (i < i' \leq m)$

$$\begin{aligned} & (O \cup \{\tau \mapsto (\overline{O}_\tau \cup O_{\text{field}})\} \cup \\ & \quad \{\langle \text{local} \rangle \mapsto (O_{\text{field}} \cup \overline{V}_{i'})\}, \tau, \tau') \triangleright_{\text{Stmt}} s_{i'} : \theta_{i'} \end{aligned}$$

This prove must be done for all $j \in \{i\} \cup \{i' \mid i' < i \vee i < i' \leq m\}$ by case differentiation for each statement:

- s_j is an expression statement:

First we consider the expression statements **Assign**, **New**, **NewArray**, and **MethodCall**.

It holds, there is $\bar{\theta}$ such that

$$(\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}((\sigma_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j, s_j).$$

Then, $\theta_j = \text{void}$.

With the expression statement rules (figure 5.7), we have to prove such that there is a $\bar{\theta}'$ with

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Expr} s_i : \bar{\theta}'$$

In the later carried out case, $s_j = \text{return}(e)$, we will even prove that $\bar{\theta} = \bar{\theta}'$.

- $s_j = \text{while}(e, \text{Block}(B))$:

In this case holds

$$(\sigma_j, \text{boolean}, \bar{O}_\tau \cup O_{field} \cup V_j) \in \text{TRExp}(\sigma_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j, e)$$

and

$$(\sigma_j, \theta_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRStmts}(\sigma_j, V_j, B, V_j).$$

With the **WhileStmt**-rule (figure 5.5), we have to prove:

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Expr} e : \text{boolean}$$

and

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Stmt} B : \theta_j$$

As said before, we will prove that $e : \text{boolean}$ is derivable in the **return**-case.

The call of **TRStmts** is a recursive call. This means that the result follows by induction.

- $s_j = \text{if}(e, \text{ifbranch}, \text{elsebranch})$:

We must distinguish the different forms of the if-statement (figure 5.6). For all cases holds

$$(\sigma_j, \text{boolean}, \bar{O}_\tau \cup O_{field} \cup V_j) \in \text{TRExp}(\sigma_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j, e).$$

Then, it is to prove:

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Expr} e : \text{boolean}$$

This prove is done in the **return**-case.

It remain to prove the different type derivations for the **then**- and the **else**-branch.

- $\text{ifbranch} = \text{Block}(B_1)$ and $\text{elsebranch} = \epsilon$:

The prove is done analog to the **while**-case.

- $\text{ifbranch} = \text{Block}(B_1)$, $\text{elsebranch} = \text{Block}(B_2)$:

It holds, there are $\bar{\theta}$ and $\bar{\theta}'$ with

$$(\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRStmts}(\sigma_j, V_j, B_1, V_j),$$

$$(\sigma_j, \bar{\theta}', \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRStmts}(\sigma_j, V_j, B_2, V_j).$$

From the **IfStmt** rules (figure 5.6) follows that it is to prove:

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Stmt} B_1 : \bar{\theta}$$

and

$$(O \cup \{\tau \mapsto (\bar{O}_\tau \cup O_{field})\} \cup \{\langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i)\}, \tau, \tau') \triangleright_{Stmt} B_2 : \bar{\theta}'.$$

As **TRStmts** is a recursive call, the result follows by induction.

It remains to consider the different cases, as θ_j is constructed from $\bar{\theta}$ and $\bar{\theta}'$. It is to show that θ_j is constructed equal in the algorithm 5.19 and in the inference rules of figure 5.6:

- $\bar{\theta} \neq \text{void}$ and $\bar{\theta}' \neq \text{void}$: Then, from the call of **TUnify_{mub}** in algorithm 5.19 follows, that it is to show

$$\theta_j \in \text{MUB}(\bar{\theta}, \bar{\theta}').$$

But, from rule **IfStmt_if_el_ty** follows this.

- $\bar{\theta} = \text{void}$ and $\bar{\theta}' = \text{void}$: Then, from algorithm 5.19 follows, that it is to show

$$\theta_j = \text{void}.$$

But from rule **IfStmt_if_ty2** follows this.

- $\bar{\theta} = \text{void}$ and $\bar{\theta}' \neq \text{void}$: Then, from algorithm 5.19 follows, that it is to show

$$\theta_j = \bar{\theta}'.$$

But from rule **IfStmt_el_ty** follows this.

- $\bar{\theta} \neq \text{void}$ and $\bar{\theta}' = \text{void}$: Then, from algorithm 5.19 follows, that it is to show

vgl. Commentary
bei Class-rule

Nochmals Reglen
überprüfen

$$\theta_j = \bar{\bar{\theta}}.$$

But from rule **IfStmt_if_ty2** follows this.

- s_j is a local variable declaration statement:

There are two different possibilities. Either the local variable can be declared with a type or without a type.

In the case that the variable is typed, as well in the algorithm 5.22 as in the inference rule **LoVarDecl2** (figure 5.5) the typed variable is added to the set of type assumptions. This means that the induction can be continued with an extended set of type assumptions. The result type θ_j is unchanged.

If the variable is not explicitly typed $s_j = \text{LocalOrFiledVar}(v)$ (rule **LoVarDecl1**) the result type θ_j is also unchanged. In this case, the set of type assumptions for the inference rules for the continuation of the induction proof must be extended by $v : \theta''$ if the determined type of v during the algorithm is θ'' .

- $s_j = \text{return}(e)$:

It is obvious that algorithm 5.20 is sound if it holds: From

$$(\sigma_j, \theta_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}(\sigma_j, V_j, e)$$

follows

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \}, \tau, \tau') \triangleright_{Expr} e : \theta_j.$$

This proof must also be done by induction.

Induction start: There are different algorithms respectively different rules which represent the induction start.

1. e is a literal expressions:

It is obvious that the results of the functions in algorithm 5.40 correspond to the results of the respective rules in figure 5.9.

2. $e = \text{this}$ or $e = \text{super}$:

It is also obvious that the results of the algorithms 5.35 and 5.36 correspond to the results of the rules **this** and **super** in figure 5.8, respectively.

3. $e = \text{LocalOrFiledVar}(v)$:

As the set of type assumptions contains the typed variable v the result of the algorithm 5.37 corresponds to the results of the rule **LocalOrFiledVar** in figure 5.8.

Induction step: There are also different algorithms respectively different

1. $e = \text{InstVar}(re, v)$

Under the assumption that if

$$(\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}(\sigma_j, V_j, re)$$

it holds

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \}, \tau, \tau') \triangleright_{Expr} re : \bar{\theta},$$

it is to prove that if

$$(\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRInstVar}((\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j), v)$$

it holds:

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \})_{\bar{\theta}} \triangleright_{Id} v : \bar{\bar{\theta}}.$$

This follows, as the type assumptions in the algorithm and in the inference rules are identical and the call of **TUnify**_{≤*} in **TRInstVar** (algorithm 5.38) corresponds to the substitution in the **Ident**-rule in figure 5.4.

HIER WEITERMACHEN

2. $e = \text{MethodCall}(re, f(t_1, \dots, t_n))$

We assume that for

$$(\sigma_j, \bar{\theta}, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}(\sigma_j, V_j, re)$$

holds

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \}, \tau, \tau') \triangleright_{Expr} re : \bar{\theta},$$

and for $1 \leq k \leq n$ with

$$(\sigma_j, \bar{\theta}_k, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}(\sigma_j, V_j, t_k)$$

respectively holds

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \}, \tau, \tau') \triangleright_{Expr} t_k : \bar{\theta}_k.$$

From the definition of **TRtuple** (algorithm 5.27) follows then

$$(\sigma_j, (\bar{\theta}, \bar{\theta}_1, \dots, \bar{\theta}_n), \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRtuple}(\sigma_j, \epsilon, V_j, (re, t_1, \dots, t_n))$$

But this means by rule **MethodCall** in figure 5.8 that it is to prove: if

$$(\sigma_j, \theta_j, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRMCallApp}(\sigma_j, (\bar{\theta}, \bar{\theta}_1, \dots, \bar{\theta}_n), V_j, f)$$

then it holds

$$(\bar{\theta}'_1, \dots, \bar{\theta}'_n, \theta_j) = \text{lub}(\bar{\theta}, f, \bar{\theta}_1, \dots, \bar{\theta}_n)$$

This follows from the property that the type assumptions of the inference rules and the algorithm correspond and in algorithm **TRMCallApp** (algorithm 5.33) at the end the determined results are filtered, such that only the result type of the function symbol with corresponding least arity is given.

3. $e = \text{Assign}(t_1, t_2)$

We assume that for $k = 1, 2$ with

$$(\sigma_j, \bar{\theta}_k, \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRExp}(\sigma_j, V_j, t_k)$$

respectively holds

$$(O \cup \{ \tau \mapsto (\bar{O}_\tau \cup O_{field}) \} \cup \{ \langle \text{local} \rangle \mapsto (O_{field} \cup \bar{V}_i) \}, \tau, \tau') \triangleright_{Expr} t_k : \bar{\theta}_k.$$

From the definition of **TRtuple** (algorithm 5.27) follows then

$$(\sigma_j, (\bar{\theta}_1, \bar{\theta}_2), \bar{O}_\tau \cup O_{field} \cup \bar{V}_j) \in \text{TRtuple}(\sigma_j, \epsilon, V_j, (t_1, t_2))$$

From the **Assign**-rule follows then, that we have to prove:

$$\bar{\theta}_2 \leq^* \bar{\theta}_1.$$

But, this is given, as the result of algorithm 5.29 is determined by **TUnify**.

4. $e = \text{New}(\theta, t_1, \dots, t_n)$

The proof is analog done as in the case of **MethodCall**.

5. $e = \text{NewArray}(\bar{\theta}, t)$

It is obvious that algorithm ??

Induction step: rule **Block_1** and **Block_2** and algorithm 5.17

Completeness

Most principal type property

Furthermore, it is to show that the types of the methods in A_τ are most principal.

For the most principal property we have to prove that for each $\bar{O}_{gen(\tau)}$ with $\bar{O}_{gen(\tau)} \triangleright Id$

$f_{\tau,i}^{(m_{\tau,i})} : \nu_1 \times \dots \times \nu_{m_{\tau,i}} \rightarrow \nu$ there is a $1 \leq j' \leq o_{(f_{\tau,i})}$ with

■

With theorem 5.10 and theorem 5.14 follows

Corollary 5.59 *The type reconstruction algorithm **TRprog** is correct and complete in correlation to the type inference system of 5.4.*

5.5.2 ANDERER SECTION NAME Type reconstruction example

In this section we present an example for the type reconstruction algorithm. We execute the algorithm for the same Java 5.0 program, the **mul** method of the class **Matrix** (fig. 5.15), as in section 5.4.2.

```
class Matrix extends Vector<Vector<Integer>> {

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

Figure 5.15: The Java 5.0 class **Matrix**

The abstract syntax **absMatrix** of this example is given in section 5.4.2.

For the algorithm we need the set of type assumptions $A = \{ A_{\forall a. \text{Vector}<a>}, A_{\text{Integer}} \}$ with

$$A_{\forall a. \text{Vector}<a>} = \{ \text{elementAt} : \text{int} \rightarrow a, \text{addElement} : a \rightarrow \text{void}, \text{size} : \rightarrow \text{int} \}$$

and

$$A_{\text{Integer}} = \{ \langle \text{init} \rangle \text{Integer} : \text{int} \rightarrow \text{Integer}, \text{intValue} : \rightarrow \text{int} \}.$$

The type of the actual class *act_cl* is given as **Matrix** and the finite closure **FC**(\leq) is given as $\{\mathbf{Matrix}\langle a \rangle \leq * \mathbf{Vector}\langle \mathbf{Vector}\langle a \rangle \rangle\}$.

Now we can start and call the **TRprog**. The run of **TRprog** is presented such that each function gets its own number. Only an end-recursive function call gets no new number.

(1) **TRprog**(*A*, **absMatrix**)

First, we apply the function **NewTVar** to **absMatrix**, which instances new type variables for each parameter type and each return type. In this example we denote the fresh type variables by the greek letters $\alpha_1, \alpha_2, \dots$

```
NewTVar( absMatrix ) =
  class(
    ClassName( Matrix ),
    extends( Vector<Vector<Integer>> ),
    Method( mul,  $\alpha_1$ , ( m :  $\alpha_2$  ), Block( Bl1 ) ) )
```

As the class **Matrix** contains no fields it holds:

$$V_{fields_methods} = \{ \mathbf{mul} : \alpha_2 \rightarrow \alpha_1 \}.$$

The set of the parameters of method **mul** is given as

$$V_1 = \{ \mathbf{m} : \alpha_2 \}.$$

Then the function **TRstart** is called.

(1.1) **TRstart**(Block(*Bl*₁), *V*₁, *V*_{fields_methods})

In function **TRstart** nearly nothing is done, as **TRstart** controls the type reconstruction of the different blocks of the respective methods in the class. In the class **Matrix** only the methods **mul** exists.

TRstart calls by **TRNextMeth** the function **TRstmt**. In **TRstmt** only the **Block** constructor is removed and the function **TRstmts** is called.

In the following let $V_{1.1} = V_{fields_methods} \cup V_1$.

(1.1.1) **TRstmts**($[], V_{1.1}, Bl_1, V_{1.1}$)

In **TRstmts** for the first statement **TRstmt** is called again.

(1.1.1.1) **TRstmt**($[], V_{1.1}, \mathbf{LocalVarDecl}(\mathbf{ret})$)

The result is $([], \mathbf{void}, V_{1.1} \cup \{ \mathbf{ret} : \alpha_3 \})$.

In the following we denote

$$V_{1.1.1} = V_{1.1} \cup \{ \mathbf{ret} : \alpha_3 \}.$$

(1.1.1) **TRstmts**($[], V_{1.1.1}, \mathbf{Assign}(\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

Then, **TRstmt** for the next statement is called.

(1.1.1.2) **TRstmt**($[], V_{1.1.1}, \mathbf{Assign}(\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

In **TRstmt** first **TRExp** is called.

(1.1.1.2.1) **TRExp**($[], V_{1.1.1}, \mathbf{Assign}(\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

In **TRExp** the function **TRtuple** is called.

(1.1.1.2.1.1) **TRtuple**($[], \epsilon, V_{1.1.1}, (\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

In **TRtuple** first **TRmultiply** is called for **LocalOrFieldVar**(**ret**). Then, in **TRmultiply** the function **TRExp** is called.

(1.1.1.2.1.1.1) **TRExp**($[], V_{1.1.1}, \mathbf{LocalOrFieldVar}(\mathbf{ret})$)

As $V_{1.1.1} = V \setminus (\mathbf{ret}^{(0)}) \cup \{ \mathbf{ret} : \alpha_3 \}$, the result is $\{ ([], \alpha_3, V_{1.1.1}) \}$.

(1.1.1.2.1.1.1) **TRtuple**($[], \epsilon, V_{1.1.1}, (\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

In **TRtuple** the **TRtuple** is called again with the second argument.

(1.1.1.2.1.1.2) **TRtuple**($[], \alpha_3, V_{1.1.1}, (\mathbf{New}(\mathbf{Matrix}, ()))$)

In **TRtuple** again **TRExp** is called by **TRmultiply**.

(1.1.1.2.1.1.2.1) **TRExp**($[], V_{1.1.1}, \mathbf{New}(\mathbf{Matrix}, ()))$)

The result is $\{ ([], \mathbf{Matrix}, V_{1.1.1}) \}$.

(1.1.1.2.1.1.2) **TRtuple**($[], \alpha_3, V_{1.1.1}, (\mathbf{New}(\mathbf{Matrix}, ()))$)

The result is built by **TRmultiply**: $\{ ([], (\alpha_3 \mathbf{Matrix}), V_{1.1.1}) \}$.

(1.1.1.2.1.1) **TRtuple**($[], \epsilon, V_{1.1.1}, (\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

As there is only one result, nothing happens and the result is also: $\{ ([], (\alpha_3 \mathbf{Matrix}), V_{1.1.1}) \}$.

(1.1.1.2.1) **TRExp**($[], V_{1.1.1}, \mathbf{Assign}(\mathbf{LocalOrFieldVar}(\mathbf{ret}), \mathbf{New}(\mathbf{Matrix}, ()))$)

The next step in **TRExp** is the type unification:

$unify = \mathbf{TUnify}_{\leq^*}(\mathbf{Matrix}, \alpha_3) = \{ [\alpha_3 \mapsto \mathbf{Matrix}], [\alpha_3 \mapsto \mathbf{Vector}\langle \mathbf{Vector}\langle \mathbf{Integer} \rangle \rangle] \}$

ACHTUNG: SIND INDIZIES VON *V* und σ SO OK?????

In the following let

$$\sigma_{1.1.1.2.1}^1 = [\alpha_3 \mapsto \mathbf{Matrix}]$$

and

$$\sigma_{1.1.1.2.1}^2 = [\alpha_3 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle].$$

and let

$$V_{1.1.1.2.1}^1 = V_{1.1.1.1} \setminus \{\text{ret} : \alpha_3\} \cup \{\text{ret} : \text{Matrix}\}$$

and

$$V_{1.1.1.2.1}^2 = V_{1.1.1.1} \setminus \{\text{ret} : \alpha_3\} \cup \{\text{ret} : \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle\}$$

Then the result is $\{(\sigma_{1.1.1.2.1}^1, \text{Matrix}, V_{1.1.1.2.1}^1), (\sigma_{1.1.1.2.1}^2, \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle, V_{1.1.1.2.1}^2)\}$.

(1.1.1.2) $\text{TRStmt}([], V_{1.1.1.1}, \text{Assign}(\text{LocalOrFieldVar}(\text{ret}), \text{New}(\text{Matrix}, ())))$

The result is $\{(\sigma_{1.1.1.2.1}^1, \text{void}, V_{1.1.1.2.1}^1), (\sigma_{1.1.1.2.1}^2, \text{void}, V_{1.1.1.2.1}^2)\}$.

(1.1.1.3) $\text{TRStmts}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1, \text{LocalVarDecl}(i) :: \text{stmts}, V_{1.1})$

TRStmt is called for the next statement.

(1.1.1.3.1) $\text{TRStmt}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1, \text{LocalVarDecl}(i))$

The result is $(\sigma_{1.1.1.2.1}^1, \text{void}, V_{1.1.1.2.1}^1 \cup \{i : \alpha_4\})$.

(1.1.1.3) $\text{TRStmts}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1, \text{Assign}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(0)) :: \text{stmts}, V_{1.1})$

With the result TRStmts was called again for the next statement. Then TRStmt is called.

(1.1.1.3.2) $\text{TRStmt}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1 \cup \{i : \alpha_4\}, \text{Assign}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(0)))$

Let in the following

$$\sigma_{1.1.1.3.2}^1 = \sigma_{1.1.1.2.1}^1 \cup [\alpha_4 \mapsto \text{int}]$$

and

$$V_{1.1.1.3.2}^1 = V_{1.1.1.2.1}^1 \cup \{i : \text{int}\}.$$

The result is $\{(\sigma_{1.1.1.3.2}^1, \text{void}, V_{1.1.1.3.2}^1)\}$.

(1.1.1.3) $\text{TRStmts}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1, \text{WhileStmt}(\dots) :: \text{stmts}, V_{1.1})$

With the result TRStmts was called again from TRStmts for the next statement. Then TRStmt is called.

(1.1.1.3.3) $\text{TRStmt}(\sigma_{1.1.1.3.2}^1, V_{1.1.1.3.2}^1, \text{WhileStmt}(\dots))$

The result is given as (cp. appendix B):

$$\{(\sigma_{1.1.1.3.3}^1, \text{void}, V_{1.1.1.3.3}^1), (\sigma_{1.1.1.3.3}^2, \text{void}, V_{1.1.1.3.3}^2)\},$$

where

$$\begin{aligned} \sigma_{1.1.1.3.3}^1 &= [\alpha_{15} \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_{13} \mapsto \text{Integer}, \alpha_{11} \mapsto \text{Integer}, \\ &\quad \alpha_{10} \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_2 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle, \alpha_8 \mapsto \text{int}, \\ &\quad \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle, \\ &\quad \alpha_6 \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_7 \mapsto \text{int}] \\ \sigma_{1.1.1.3.3}^2 &= [\alpha_{16} \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_{14} \mapsto \text{Integer}, \alpha_{12} \mapsto \text{Integer}, \alpha_2 \mapsto \text{Matrix}, \\ &\quad \alpha_{10} \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}, \\ &\quad \alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_6 \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_7 \mapsto \text{int}] \\ V_{1.1.1.3.3}^1 &= \{\text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector}\langle \text{Integer} \rangle, v2 : \text{Vector}\langle \text{Integer} \rangle, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle \rightarrow \alpha_1, \\ &\quad m : \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle\} \\ V_{1.1.1.3.3}^2 &= \{\text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector}\langle \text{Integer} \rangle, v2 : \text{Vector}\langle \text{Integer} \rangle, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Matrix} \rightarrow \alpha_1, m : \text{Matrix}\}. \end{aligned}$$

(1.1.1.3.4) $\text{TRStmts}(\sigma_{1.1.1.3.3}^1, V_{1.1.1.3.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

With the result TRStmts was called again from TRStmts for the next statement. Then TRStmt is called.

(1.1.1.3.4.1) $\text{TRExp}(\sigma_{1.1.1.3.3}^1, V_{1.1.1.3.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret})))$

In TRStmt for Return the function TRExp was called.

The result is given as:

$$\{(\sigma_{1.1.1.3.3}^1, \text{Matrix}, V_{1.1.1.3.3}^1)\}.$$

(1.1.1.3.4) $\text{TRStmts}(\sigma_{1.1.1.3.3}^1, V_{1.1.1.3.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

The result is given as:

$$\{(\sigma_{1.1.1.3.3}^1, \text{Matrix}, V_{1.1.1.3.3}^1 \setminus \{\text{ret} : \text{Matrix}, i : \text{int}\})\}.$$

(1.1.1.3.5) $\text{TRStmts}(\sigma_{1.1.1.3.3}^2, V_{1.1.1.3.3}^2, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

Analogous to the first case the result is given as:

$$\{(\sigma_{1.1.1.3.3}^2, \text{Matrix}, V_{1.1.1.3.3}^2 \setminus \{\text{ret} : \text{Matrix}, i : \text{int}\})\}.$$

HIER WEITERMACHEN

(1.1.1.4) $\text{TRStmts}(\sigma_{1.1.1.2.1}^1, V_{1.1.1.2.1}^1, \text{LocalVarDecl}(i) :: \text{stmts}, V_{1.1})$

TRStmt is called for the next statement.

(1.1.1.4.1) $\text{TRStmt}(\sigma_{1.1.1.2.1}^2, V_{1.1.1.2.1}^2, \text{LocalVarDecl}(i))$

The result is $(\sigma_{1.1.1.2.1}^2, \text{void}, V_{1.1.1.2.1}^2 \cup \{i : \alpha_{17}\})$.

(1.1.1.4) $\text{TRStmts}(\sigma_{1.1.1.2.1}^2, V_{1.1.1.2.1}^2, \text{Assign}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(0)) :: \text{stmts}, V_{1.1})$

With the result TRStmts was called again for the next statement. Then TRStmt is called.

(1.1.1.4.2) $\text{TRStmt}(\sigma_{1.1.1.2.1}^2, V_{1.1.1.2.1}^2 \cup \{i : \alpha_{17}\}, \text{Assign}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(0)))$

Let in the following

$$\sigma_{1.1.1.4.2}^2 = \sigma_{1.1.1.2.1}^1 \cup [\alpha_{17} \mapsto \text{int}]$$

and

$$V_{1.1.1.4.2}^2 = V_{1.1.1.2.1}^2 \cup \{i : \text{int}\}.$$

The result is $\{(\sigma_{1.1.1.4.2}^2, \text{void}, V_{1.1.1.4.2}^2)\}$.

(1.1.1.4) $\text{TRStmts}(\sigma_{1.1.1.2.1}^2, V_{1.1.1.2.1}^2, \text{WhileStmt}(\dots) :: \text{stmts}, V_{1.1})$

With the result TRStmts was called again from TRStmts for the next statement. Then TRStmt is called.

(1.1.1.4.3) $\text{TRStmt}(\sigma_{1.1.1.4.2}^2, V_{1.1.1.4.2}^2, \text{WhileStmt}(\dots))$

Analogous to the first case, which is shown in appendix B) the result differs only in the fresh variable names and in the type for **ret**, which is here **Vector<Vector<Integer>>** instead of **Matrix**.

$$\{(\sigma_{1.1.1.4.3}^1, \text{void}, V_{1.1.1.4.3}^1), (\sigma_{1.1.1.4.3}^2, \text{void}, V_{1.1.1.4.3}^2)\},$$

where

ALLE VARIABLEN UMBENNENEN BIS AUF $\alpha_1, \alpha_2, \alpha_3$ UMBENNENEN
ERSTE NEUE FREIE FRISCHE VARIABLE IST α_{18} .

$$\begin{aligned} \sigma_{1.1.1.4.3}^1 &= [\alpha_{28} \mapsto \text{Vector<Integer>}, \alpha_{26} \mapsto \text{Integer}, \alpha_{24} \mapsto \text{Integer}, \\ &\quad \alpha_{23} \mapsto \text{Vector<Integer>}, \alpha_2 \mapsto \text{Vector<Vector<Integer>>}, \alpha_{21} \mapsto \text{int}, \\ &\quad \alpha_{22} \mapsto \text{int}, \alpha_3 \mapsto \text{Vector<Vector<Integer>>}, \alpha_{17} \mapsto \text{int}, \\ &\quad \alpha_{18} \mapsto \text{Vector<Integer>}, \alpha_{19} \mapsto \text{Vector<Integer>}, \alpha_{20} \mapsto \text{int}] \\ \sigma_{1.1.1.4.3}^2 &= [\alpha_{29} \mapsto \text{Vector<Integer>}, \alpha_{27} \mapsto \text{Integer}, \alpha_{25} \mapsto \text{Integer}, \alpha_2 \mapsto \text{Matrix}, \\ &\quad \alpha_{23} \mapsto \text{Vector<Integer>}, \alpha_{21} \mapsto \text{int}, \alpha_{22} \mapsto \text{int}, \\ &\quad \alpha_3 \mapsto \text{Vector<Vector<Integer>>}, \alpha_{17} \mapsto \text{int}, \alpha_{18} \mapsto \text{Vector<Integer>}, \\ &\quad \alpha_{19} \mapsto \text{Vector<Integer>}, \alpha_{20} \mapsto \text{int}] \\ V_{1.1.1.4.3}^1 &= \{\text{ret} : \text{Vector<Vector<Integer>>}, i : \text{int}, v1 : \text{Vector<Integer>}, \\ &\quad v2 : \text{Vector<Integer>}, j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \\ &\quad \text{mul} : \text{Vector<Vector<Integer>>} \rightarrow \alpha_1, m : \text{Vector<Vector<Integer>>}\} \\ V_{1.1.1.4.3}^2 &= \{\text{ret} : \text{Vector<Vector<Integer>>}, i : \text{int}, v1 : \text{Vector<Integer>}, \\ &\quad v2 : \text{Vector<Integer>}, j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Matrix} \rightarrow \alpha_1, \\ &\quad m : \text{Matrix}\}. \end{aligned}$$

(1.1.1.4.4) $\text{TRStmts}(\sigma_{1.1.1.4.3}^1, V_{1.1.1.4.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

With the result TRStmts was called again from TRStmts for the next statement. Then TRStmt is called.

(1.1.1.4.4.1) $\text{TRExp}(\sigma_{1.1.1.4.3}^1, V_{1.1.1.4.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret})))$

In TRStmt for **Return** the function TRExp was called.

The result is given as:

$$\{(\sigma_{1.1.1.4.3}^1, \text{Vector<Vector<Integer>>}, V_{1.1.1.4.3}^1)\}.$$

(1.1.1.4.4) $\text{TRStmts}(\sigma_{1.1.1.4.3}^1, V_{1.1.1.4.3}^1, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

The result is given as:

$$\{(\sigma_{1.1.1.4.3}^1, \text{Vector<Vector<Integer>>}, V_{1.1.1.4.3}^1 \setminus \{\text{ret} : \text{Vector<Vector<Integer>>}, i : \text{int}\})\}.$$

(1.1.1.4.5) $\text{TRStmts}(\sigma_{1.1.1.4.3}^2, V_{1.1.1.4.3}^2, \text{Return}(\text{LocalOrFieldVar}(\text{ret}))) :: [], V_{1.1})$

Analogous to the first case the result is given as:

$$\{(\sigma_{1.1.1.4.3}^2, \text{Vector<Vector<Integer>>}, V_{1.1.1.4.3}^2 \setminus \{\text{ret} : \text{Vector<Vector<Integer>>}, i : \text{int}\})\}.$$

(1.1) $\text{TRstart}(\text{Block}(Bl_1), V_1, V_{\text{fields_methods}})$

In (1.1.1.4.3) two different types von **ret** were reconstructed. Theses two traces were both divided again. The result are four different triples with respective reconstructed types.

In the following let

$$V_{1.1.1.3.3}^{1'} = V_{1.1.1.3.3}^1 \setminus \{\text{ret} : \text{Matrix}, i : \text{int}\},$$

$$V_{1.1.1.3.3}^{2'} = V_{1.1.1.3.3}^2 \setminus \{\text{ret} : \text{Matrix}, i : \text{int}\},$$

$$V_{1.1.1.4.3}^{1'} = V_{1.1.1.4.3}^1 \setminus \{\text{ret} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, i : \text{int}\},$$

and

$$V_{1.1.1.4.3}^{2'} = V_{1.1.1.4.3}^2 \setminus \{\text{ret} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, i : \text{int}\}.$$

Then, the result

$$\begin{aligned} & \{ (\sigma_{1.1.1.3.3}^1, \text{Matrix}, V_{1.1.1.3.3}^{1'}), \\ & (\sigma_{1.1.1.3.3}^2, \text{Matrix}, V_{1.1.1.3.3}^{2'}), \\ & (\sigma_{1.1.1.4.3}^1, \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, V_{1.1.1.4.3}^{1'}), \\ & (\sigma_{1.1.1.4.3}^2, \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, V_{1.1.1.4.3}^{2'}) \} \end{aligned}$$

is returned to TRstart .

Then, in TRstart the following is done:

- $\theta_1 = \text{RetType}(\text{mul}, V_{1.1.1.3.3}^{1'}) = \alpha_1$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\text{Matrix}, \alpha_1)$
 $= \{ [\alpha_1 \mapsto \text{Matrix}], [\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle] \}$
- $\theta_2 = \text{RetType}(\text{mul}, V_{1.1.1.3.3}^{2'}) = \alpha_1$
- $\text{unify}_2 = \text{TUnify}_{\leq^*}(\text{Matrix}, \alpha_1)$
 $= \{ [\alpha_1 \mapsto \text{Matrix}], [\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle] \}$
- $\theta_3 = \text{RetType}(\text{mul}, V_{1.1.1.4.3}^{1'}) = \alpha_1$
- $\text{unify}_3 = \text{TUnify}_{\leq^*}(\text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, \alpha_1)$
 $= \{ [\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle] \}$
- $\theta_4 = \text{RetType}(\text{mul}, V_{1.1.1.4.3}^{2'}) = \alpha_1$
- $\text{unify}_4 = \text{TUnify}_{\leq^*}(\text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle, \alpha_1)$
 $= \{ [\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle] \}$

$$\begin{aligned} - \text{ret}_1 = & \{ ([\alpha_1 \mapsto \text{Matrix}], \\ & V_{1.1.1.3.3}^{1'} \setminus \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \alpha_1\} \\ & \cup \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \text{Matrix}\}) \\ & ([\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle], \\ & V_{1.1.1.3.3}^{1'} \setminus \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \alpha_1\} \\ & \cup \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle]) \\ & ([\alpha_1 \mapsto \text{Matrix}], \\ & V_{1.1.1.3.3}^{2'} \setminus \{\text{mul} : \text{Matrix} \rightarrow \alpha_1\} \cup \{\text{mul} : \text{Matrix} \rightarrow \text{Matrix}\}) \\ & ([\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle], \\ & V_{1.1.1.3.3}^{2'} \setminus \{\text{mul} : \text{Matrix} \rightarrow \alpha_1\} \\ & \cup \{\text{mul} : \text{Matrix} \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle]) \\ & ([\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle], \\ & V_{1.1.1.4.3}^{1'} \setminus \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \alpha_1\} \\ & \cup \{\text{mul} : \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle]) \\ & ([\alpha_1 \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle], \\ & V_{1.1.1.4.3}^{2'} \setminus \{\text{mul} : \text{Matrix} \rightarrow \alpha_1\} \\ & \cup \{\text{mul} : \text{Matrix} \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle]) \} \end{aligned}$$

The result is then given as ret_1 .

(1) $\text{TRprog}(A, \text{absMatrix})$

In TRprog finally, the intersection type of the different reconstructed type is built:

$$\begin{aligned} A_{\text{Matrix}} = & \{ \text{mul}^{(1)} : (\text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \text{Matrix}) \\ & \wedge (\text{Matrix} \rightarrow \text{Matrix}) \\ & \wedge (\text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle) \\ & \wedge (\text{Matrix} \rightarrow \text{Vector}\langle \text{Vector}\langle \text{Integer}\rangle\rangle) \} \end{aligned}$$

The result is then given as:

$$\{ A_{\forall a. \text{Vector}\langle a \rangle}, A_{\text{Integer}}, A_{\text{Matrix}} \}.$$

HIER WEITERMACHEN

NOCHMALS UEBERPRUEFEN, OB NICHT BY TUNIFY-AUFRUFEN ALTE ZUORDNUNGEN VON ALLTEN UNIFIKATOREN RAUSFALLEN

5.6 Implementation

- Wie detailliert soll eine Implementierung beschrieben werden?

5.7 Comparison of our approach to other existing approaches

In this section we compare our approach to the existing approaches presented in section 5.2. In the first part we give a comparison to the flow analysis approach. Then we show how our approach is more comprehensive than the refactoring approaches, which infers parameters of raw declared types. Finally we compare it to the milner-like approaches.

5.7.1 Flow analysis approaches

The approaches in [PS91, OPS92, APS93, PS94] respectively in [PC94, Age95, WS01] base on data flow analysis to construct the constraints. Finally the constraint system is solved. In contrast our approach, like the Hindley–Milner approach, computes the types by assuming types for the methods, which are transformed to the result by unifying during a tree trace of the program’s abstract syntax tree.

Besides the fundamental differences of the type inference algorithms, the flow analysis approaches differ in the goal. While we try to find a most principle type, here a mostly concret type is demanded. The approaches are developed for code optimization. In contrast our approach is developed for reusing application code. These two approaches are contrary.

The following example shows the differences. We give the example in Java-like syntax.

```
interface Comparable<T> {
    int compareTo(T o);
}

class Max {
    max(x, y) {
        if (x.compareTo(y) > 0) return x
        else return y;
    }
}

class Main {
    main () {
        ...
        max(1.1, 1.2);
        ...
    }
}
```

```
        max(1,1);
        ...
    }
}
```

The result of our type inference algorithm would be

$$\langle T \rangle \text{ max} : \text{Comparable}\langle T \rangle \times \text{Comparable}\langle T \rangle \rightarrow \text{Comparable}\langle T \rangle.$$

In `main` in the first call of `max` the parameter `T` would be instantiated by `Float` and in the second call it would be instantiated by `Integer`, as `Float` implements `Comparable<Float>` as well as `Integer` implements `Comparable<Integer>`.

In [PS94] the type

$$\langle T \rangle \text{ max} : \text{Comparable}\langle T \rangle \times \text{Comparable}\langle T \rangle \rightarrow \text{Comparable}\langle T \rangle$$

is considered as unprecise. Therefore, the type for `max` is determined not until `max` is called. This leads to two discriminated types for

$$\text{max} : \text{Comparable}\langle \text{Float} \rangle \times \text{Comparable}\langle \text{Float} \rangle \rightarrow \text{Comparable}\langle \text{Float} \rangle$$

and

$$\text{Comparable}\langle \text{Integer} \rangle \times \text{Comparable}\langle \text{Integer} \rangle \rightarrow \text{Comparable}\langle \text{Integer} \rangle.$$

These different type informations are important for the code optimization as the `compareTo` methods of floats and integers are completely different.

Subsumed the approaches determine not most principle types, like our approach, but they determine mostly concret types.

5.7.2 Converting raw types to parametrized types

If we consider the algorithms in [DKTE04, FTK⁺05], we see that they base on the algorithms of flow analysis approaches. This means that the algorithms are basically different to our type inference algorithm, as described in section 5.7.1.

Furthermore, there is another fundamental difference. There is no statement, no variable, no argument, and no function result without any type information. The type informations are only not precise. This leads to the difference that these algorithms need not to consider data- and function polymorphism, which means that the problem, which is solved by our algorithm, could be considered as an extension of the problem inferring type parameters.

S-unification

In [DKTE04] in the algorithm determining the types of the allocation sides there is one affinity to our approach. The S-unification is similar to our type unification algorithm (cp. section 4.2). Both algorithms unify sub-types. But, there are three main differences.

First in the S-unification algorithm type variables can only occur in the side of the greater type. This means that only constraints of the form $type \leq^* typvar$ (lower bounds) can be derived. The second difference is, that in the S-unification algorithm the constraints are preserved during the rest of the calculation, while in our algorithm the constraints are immediately solved. The preserving of the constraints has the advantage that only for the lower bounds of the solutions the algorithm is recursively called, while in our algorithm for all solutions the algorithm is recursively called. If we would also call the algorithm only for the lower bounds, we would possibly loose some solution as, following from the first difference, in our approach also upper bounds are allowed. Additionally, we need not do resolution as our algorithm gives real results, while the S-unification gives constraints (lower bounds) of the results. The third difference is, if we have determined one constraint for a type variable and solved subsequently, we substitute all other occurrences of the type variable by the respective solution, such that later on never another constraint could be determined for this type variable. This means the reunification of the S-unification algorithm is unnecessary in our algorithm.

5.7.3 Cecil

The main differences to our type system are the existence of functions objects of the form $\lambda(v_1 : T_1, \dots, v_n : T_n) : T_r\{B\}$ and the lack of intersection types to describe overloading. These intersection types are here not necessary, as only type-checking and not type-inference is the goal.

A further difference is the existence of a subsumption rule, which determines that if an expression has a type it has also all supertypes of this type. If there would be a type-checking rule for the assignment, this would let to an unsound type-system⁴, as the following example shows.

Example 5.60 *Let the following class declaration be given*

```
class A {
    B f;
}

class B extends A { ... }
```

Then, `newA().f = newA()`; is valid, although `newA().f` has the type `B` and the left hand side of an assignment must be greater than the right hand side. The reason is that `newA().f` has also the type `A`, as it holds $B \leq A$.

5.7.4 Milner-like approaches

The idea of our approach is, that we took the the Hindley–Milner approach, which gives a type inference algorithm for the λ -calculus. We erase in our approach the function

⁴In [DEK99] the soundness of Java (with generics) is defined as preserving types up to the subclass/subinterface relation during program execution.

constructor \rightarrow , as in Java 5.0 there are no λ -terms. Instead of the function constructor we need function types $\theta_1 \times \dots \times \theta_n \rightarrow \theta_0$ (cp. definition 5.1) for the type description of methods. As the arguments of methods in Java 5.0 have either base types or simple types the types $\theta_0 \dots \theta_n$ are no function types, which means that the type system has no higher-order function types.

Furthermore we add to the type system data-polymorphism and function-polymorphism, which means that ad-hoc polymorphism respective overloading is introduced. We do this by multiplying the set of type assumptions in case of data- or function-polymorphism. This leads finally to the property that the result can be an intersection type of function types.

Now, we explain by means of the main example in [EST95a], here given in the language I-LOOP, the different possibilities of object-oriented languages with and without higher-order functions. In [EST95a] two classes `View` and `GView` are defined. `GView` is a subclass of `View`. The idea is that a view is dependant of other views, which is realized by a field `dep` in `View`. Furthermore there is a method `doall`, which is parametrized by an arbitrary function. The method `doall` applies the arbitrary function to itself and to all dependant views.

First we present the example as a Java 5.0 program without typings:

```
class View {
    View dep;

    doall (fu) {
        fu.f(this);
        if (dep == null) return new Empty();
        else dep.doall(fu);
    }

    setDep(v) { dep = v; }
}

class GView extends View {

    draw () { return new Empty(); }
}

interface Function {
    Empty f(View v);
}

class FuDraw implements Function {
    f(ob) { return ob.draw(); }
}

class Empty { }
```

```

class Main {
    public static void main(String[] args) {
        v1 = new View();
        g1 = new GView();
        g2 = new GView();
        g3 = new GView();
        g1.setDep(v1);
        g2.setDep(g3);
        FuDraw fudraw = new FuDraw();
        g2.doall(fudraw);
    }
}

```

The parameter function of `doall`, which should be applied to the view and to all dependant views is hidden in the interface `Function`. This means that for each application of the method `doall` an own class has to be written, which implements the interface `Function`. In the given case the method `draw` of themselves should be invoked. Therefore the class `FuDraw` is implemented.

The following implementation is done in `PIZZA`, which contains higher-order functions and λ -terms.

```

class View {
    View dep;

    Empty doall ((View)->Empty f) {
        f(this);
        if (dep == null) return new Empty();
        else dep.doall(f);

        //never reached
        return new Empty();
    }

    void setDep(View v) { dep = v; }
}

class GView extends View {

    Empty draw () { return new Empty(); }
}

class Empty { }

class Main {
    public static void main(String[] args) {
        View v1 = new View();

```

```

        GView g1 = new GView();
        GView g2 = new GView();
        GView g3 = new GView();
        g1.setDep(v1);
        g2.setDep(g3);
        g2.doall(fun(GView ob)->Empty { return ob.draw(); });
    }
}

```

In this implementation the parameter `f` of the method `doall` has a higher-order type $(\text{View}) \rightarrow \text{Empty}$. This means that the method calls of `doall` expect a function as argument. In the last line of `Main` such an argument is given by the λ -term `fun(GView ob) -> Empty { return ob.draw(); }`. The hiding of the parameter function in an interface and its implementation, respectively, is not longer necessary.

Unfortunately, both implementations are not type correct. In the `Java 5.0` implementation the type-inference algorithm infers for the method `f` in the class `FuDraw` the function type $\text{GView} \rightarrow \text{Empty}$. But then `FuDraw` is not an implementation of the interface `Function`, as for `f` in `Function` the type $\text{View} \rightarrow \text{Empty}$ is demanded. The `PIZZA` implementation has an analogous problem. The λ -term `fun(GView ob)->Empty { return ob.draw(); }` has not the demanded type $(\text{View}) \rightarrow \text{Empty}$ of the argument of `doall`.

In [EST95a] this example is used to explain the two philosophies *subclasses are subtyping* respective *inheritance is not subtyping* [CHC90]. If the field `dep` in the subclass `GView` would have the type `GView` then for the argument `f` of `doall` the type $(\text{GView}) \rightarrow \text{Empty}$ would be inferred. Then the method call in `Main` would be type correct.

Nevertheless, in [EST95a] in the constraints no inconsistency could be found. In the languages `L-LOOP` the types of fields are also inferred, which means following the philosophy *inheritance is not subtyping* that for the field `dep` in the subclass `GView` the type `GView` could be inferred. As `g2` and `g3`, respectively, are instances of `GView`, such typing would be correct, although in `Java 5.0` and in `PIZZA`, respectively, the program is not type correct. The main problems of this type system are that the types are larger and less easily readable for programmers than the types of the Hindley–Milner type system.

5.7.5 OCAML

In this section we give first a comparison of the approaches of `OCAML/SML` and `Java 5.0`, generally. Then, we consider the differences of subtyping, specially.

General difference

The main difference between the approaches of the type systems are that `SML` (Hindley–Milner) allows higher-order types for the functions explicitly, where in `Java 5.0` only first-order types for methods are allowed. Implicitly, higher-order types are allowed by hiding in objects. The following example shows this

Example 5.61 We consider the function `map` which applies another function to each element of a list, respectively to each element of a vector.

In OCAML `map` is defined as:

NOCH TESTEN

```
let rec map f = function
| [] -> []
| x :: xs -> (f x) :: (map f xs)
```

The type of `map` is $(\alpha \rightarrow \beta) \rightarrow \alpha \text{list} \rightarrow \beta \text{list}$

In Java 5.0 an interface has to be declared, such that the `map` function can be defined in this way.

```
interface function<a,b> {
    b f (a x);
}
```

Then, the method `map` can be declared by:

```
Vector<b> map (function<a,b> fu, Vector<a> v) {
    Vector<b> ret = new Vector<b>();
    for(int i=0; i<v.size(); i++) {
        ret.add(fu.f(v.get(i)));
    }
    return ret;
}
```

The type of `map` is a first-order type $\text{function} < a, b > \times \text{Vector} < a > \rightarrow \text{Vector} < b >$.

The main problem of this “higher-order type hiding” is, that for each instantiated function no its own class must be declared. For example if a `Vector` of `Integers` is transformed in `Floats`, the following class `trans` must be declared:

```
class trans implements function<Integer,Float> {
    public Float f (Integer i) {
        return new Float(i);
    }
}
```

If we consider these different approaches, the OCAML approach seems more flexible. The flexibility implicates that full automatic type inference is impossible. In [Lei83] is proved that in a higher-order type system with subtyping and overloading the type inference problem is not decidable.

Subtyping

In OCAML subtyping is never implicit. This means that there are coercions necessary.

This means that the implementation of the matrix multiplication in OCAML leads not to an intersection type.

```
class matrix a =
object (this)
inherit [int vector] vector a
method mul (m : matrix) =
    let ret = new matrix (new vector 0) in
    for i = 0 to this#size do
        let v1 = this#elementAt i in
        let v2 = new vector 0 in
        for j = 0 to v1#size do
            let erg = ref 0 in
            for k = 0 to v1#size do
                erg := !erg + ((v1#elementAt k) * ((m#elementAt k)#elementAt j));
                v2#add !erg
            done;
            ret#add v2;
        done;
    done;
    ret
end;;
```

Furthermore the type inference system is not able to determine a real type for the method `mul`. The following error message is given:

```
(< elementAt : int -> < elementAt : int -> int; .. >; .. > as 'a) -> matrix
where 'a is unbound
```

This means that the system determines the record, of the type signature, but is not able to map to this record to a class. If the argument is typed by `matrix` then the following result is determined:

```
class matrix :
int vector ->
object
    val mutable l : int vector array
    method add : int vector -> unit
    method elementAt : int -> int vector
    method mul : matrix -> matrix
    method size : int
end
```

5.7.6 ImplicitPoly–Tiger

The ideas which are given in [App02] are the nearest to the ideas of our type inference system. In **ImplicitPoly–Tiger** in contrast to our approach are only no intersection types introduced, which means that the user declared functions are not overloaded. This leads to the fact that the type reconstruction can be traced back to the ordinary unitary unification.

Appendix A

Abstract syntax

Appendix B

Type reconstruction example

HIER WEITERMACHEN
IN TRStmts AM ENDE DIE HINZUGEFUEGTEN LOKALEN VARIABLEN WIEDER
ENTFERNEN UND NUMMERIERUNG UEBERPRUEFEN. ENDREKURSIVE AUFRUFE
ERHALTEN KEINE NEUE NUMMER.

4. ARGUMENT IN TRStmts EINTRAGEN

In this section we complete the type reconstruction example from section 5.5.2. The Java 5.0 program, which types are reconstructed is given as:

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

The corresponding abstract syntax is given in section 5.4.2.

Only the emphasized `while`-loop is considered in this appendix. The framework is considered in section 5.5.2. From this follows for the type unifier

$$\sigma = [\alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}],$$

for the set of local type assumptions

$$V = \{\text{mul} : \alpha_2 \rightarrow \alpha_1, m : \alpha_2, \text{ret} : \text{Matrix}, i : \text{int}\}$$

and for the set of type assumptions of other classes

$$A_{\text{Vector}\langle a \rangle} = \{\text{elementAt} : \text{int} \rightarrow a, \text{addElement} : a \rightarrow \text{void}, \text{size} : \rightarrow \text{int}\}$$

and

$$A_{\text{Integer}} = \{\text{<init>Integer} : \text{int} \rightarrow \text{Integer}, \text{intValue} : \rightarrow \text{int}\}.$$

Now we reconstruct the types for the `while`-loop:

$$(1) \text{ TRStmt}(\sigma, V, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$$

First, for the condition expression the function `TRExp` is called.

$$(1.1) \text{ TRExp}(\sigma, V, \text{Less}(\text{LocalOrFieldVar}(i), \text{MethodCall}(\text{this}, \text{size}(\dots))))$$

In the `TRExp` algorithm for `Less` first `TRExp` is called again for the first argument.

$$(1.1.1) \text{ TRExp}(\sigma, V, \text{LocalOrFieldVar}(i))$$

The result is $\{(\sigma, \text{int}, V)\}$.

$$(1.1) \text{ TRExp}(\sigma, V, \text{Less}(\text{LocalOrFieldVar}(i), \text{MethodCall}(\text{this}, \text{size}(\dots))))$$

Then, `TRExp` is called for the second argument.

$$(1.1.2) \text{ TRExp}(\sigma, V, \text{MethodCall}(\text{this}, \text{size}(\dots)))$$

First, `TRtuple` is called with the receiver `this`. The result of this call is $\{(\sigma, \text{Matrix}, V)\}$. Then, the function `TRMCallApp` is called.

(1.1.2.1) $\text{TRMCallApp}(\sigma, (\text{Matrix}), V, \text{size}())$

- $A_{\forall a. \text{Vector}\langle a \rangle} = \{ \text{elementAt} : \text{int} \rightarrow a, \text{addElement} : a \rightarrow \text{void}, \text{size} : \text{int} \}$
- $(\bar{\theta}_{1,0}, \rightarrow \bar{\theta}_1) = (\text{Vector}\langle \beta_1 \rangle, \rightarrow \text{int})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\text{Matrix}, \text{Vector}\langle \beta_1 \rangle) = [\beta \mapsto \text{Vector}\langle \text{Integer} \rangle]$
- $\text{res}_{\text{Vector}} = \{ (\sigma, \text{int}, V) \}$
- $\text{case}_1 = \{ (\sigma, \text{int}, V) \}$
- $\text{case}_2 = \emptyset$

This means that the result is $\{ (\sigma, \text{int}, V) \}$.

(1.1) $\text{TRExp}(\sigma, V, \text{Less}(\text{LocalOrFieldVar}(i), \text{MethodCall}(\text{this}, \text{size}())))$

As the return types of both arguments are `int` and respectively only one result is given, the result of this function call is $\{ (\sigma, \text{boolean}, V) \}$.

(1) $\text{TRStmt}(\sigma, V, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$

After the type reconstruction of the condition the TRStmts is called for the statements of the Block.

(1.2) $\text{TRStmts}(\sigma, V, \text{LocalVarDecl}(v1) :: \text{Assign}(\dots) :: \text{stmts}, V)$

The result of $\text{TRStmt}(\sigma, V, \text{LocalVarDecl}(v1))$ is $\{ (\sigma, \text{void}, V \cup \{ v1 : \alpha_5 \}) \}$.

In the following let

$$V_{1.2} = V \cup \{ v1 : \alpha_5 \}.$$

With the result TRStmts was called again for the next statement. Then TRStmt is called.

(1.2.1) $\text{TRStmt}(\sigma, V_{1.2}, \text{Assign}(\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

In TRStmt the function TRExp is called.

(1.2.1.1) $\text{TRExp}(\sigma, V_{1.2}, \text{Assign}(\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

In TRExp the function TRtuple is called.

(1.2.1.1.1) $\text{TRtuple}(\sigma, \epsilon, V_{1.2}, (\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

The result of $\text{TRmultiply}(\sigma, \epsilon, V_{1.2}, \text{LocalOrFieldVar}(v1))$ is $\{ (\sigma, \alpha_5, V_{1.2}) \}$.

With this result the function TRtuple is called again for the second expression. In TRtuple the function TRExp is called from TRmultiply .

(1.2.1.1.1.1) $\text{TRExp}(\sigma, V_{1.2}, \text{MethodCall}(\text{this}, \text{elementAt}(\text{LocalOrFieldVar}(i))))$

In TRExp first TRtuple is called. The result of TRtuple is $\{ (\sigma, (\text{Matrix int}), V_{1.2}) \}$. Then, the function TRMCallApp is called.

(1.2.1.1.1.1.1) $\text{TRMCallApp}(\sigma, (\text{Matrix int}), V_{1.2}, \text{elementAt}(\text{LocalOrFieldVar}(i)))$

- $A_{\forall a. \text{Vector}\langle a \rangle} = \{ \text{elementAt} : \text{int} \rightarrow a, \text{addElement} : a \rightarrow \text{void}, \text{size} : \text{int} \}$
- $(\bar{\theta}_{j,0}, \bar{\theta}_{j,1} \rightarrow \bar{\theta}_j) = (\text{Vector}\langle \beta_2 \rangle, \text{int} \rightarrow \text{Vector}\langle \text{Integer} \rangle)$
- $\text{unify}_j = \text{TUnify}_{\leq^*}((\text{Matrix}, \text{int}), (\text{Vector}\langle \beta_2 \rangle, \text{int})) = \{ [\beta_2 \mapsto \text{Vector}\langle \text{Integer} \rangle] \}$
- $\text{res}_{\text{Vector}} = \{ (\sigma, \text{Vector}\langle \text{Integer} \rangle, V_{1.2}) \}$
- $\text{case}_1 = \{ (\sigma, \text{Vector}\langle \text{Integer} \rangle, V) \}$
- $\text{case}_2 = \emptyset$

This means that the result is $\{ (\sigma, \text{Vector}\langle \text{Integer} \rangle, V_{1.2}) \}$.

(1.2.1.1.1) $\text{TRtuple}(\sigma, \epsilon, V_{1.2}, (\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

The result is $\{ (\sigma, (\alpha_5 \text{Vector}\langle \text{Integer} \rangle), V_{1.2}) \}$.

(1.2.1.1) $\text{TRExp}(\sigma, V_{1.2}, \text{Assign}(\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

- $\text{unify} = \text{TUnify}_{\leq^*}(\text{Vector}\langle \text{Integer} \rangle, \alpha_5) = \{ [\alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle] \}$
- $\text{sub}([\alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle] \circ \sigma, V_{1.2}) = V_{1.2} \setminus \{ v1 : \alpha_5 \} \cup \{ v1 : \text{Vector}\langle \text{Integer} \rangle \}$

In the following let

$$\sigma_{1.2.1.1} = [\alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle] \circ \sigma$$

and

$$V_{1.2.1.1} = V_{1.2} \setminus \{ v1 : \alpha_5 \} \cup \{ v1 : \text{Vector}\langle \text{Integer} \rangle \}.$$

Then, the result is given as $\{ \sigma_{1.2.1.1}, \text{Vector}\langle \text{Integer} \rangle, V_{1.2.1.1} \}$.

(1.2.1) $\text{TRStmt}(\sigma, V_{1.2}, \text{Assign}(\text{LocalOrFieldVar}(v1), \text{MethodCall}(\text{this}, \dots)))$

The result is $\{ \sigma_{1.2.1.1}, \text{void}, V_{1.2.1.1} \}$.

(1.3) $\text{TRStmts}(\sigma_{1.2.1.1}, V_{1.2.1.1}, \text{LocalVarDecl}(v2) :: \text{Assign}(\dots) :: \text{stmts}, V)$

With the result TRStmts was called again for the next statement.

The result of $\text{TRStmt}(\sigma_{1.2.1.1}, V_{1.2.1.1}, \text{LocalVarDecl}(v2))$ is $\{ (\sigma_{1.2.1.1}, \text{void}, V_{1.2.1.1} \cup \{ v2 : \alpha_6 \}) \}$.

In the following let

$$V_{1.3} = V_{1.2.1.1} \cup \{ v2 : \alpha_6 \}.$$

(1.4) $\text{TRStmts}(\sigma_{1.2.1.1}, V_{1.3}, \text{Assign}(\text{LocalOrFieldVar}(v2), \dots) :: \text{stmts}, V)$

With the result TRStmts was called again for the next statement.
The result of

$\text{TRStmt}(\sigma_{1.2.1.1}, V_{1.3}, \text{Assign}(\text{LocalOrFieldVar}(v2), \text{New}(\text{Vector<Integer>}(), \dots)))$

is

$$\{ (\sigma_{1.2.1.1} \cup [\alpha_6 \mapsto \text{Vector<Integer>}], \text{void}, V_{1.3} \setminus \{v2 : \alpha_6\} \cup \{v2 : \text{Vector<Integer>}\}) \}$$

With this result TRStmts is called twice.

In the following let

$$\sigma_{1.4} = \sigma_{1.2.1.1} \cup [\alpha_6 \mapsto \text{Vector<Integer>}],$$

and

$$V_{1.4} = V_{1.3} \setminus \{v2 : \alpha_6\} \cup \{v2 : \text{Vector<Integer>}\}.$$

(1.4.1) $\text{TRStmts}(\sigma_{1.4}, V_{1.4}, \text{LocalVarDecl}(j) :: \text{Assign}(\text{LocalOrFieldVar}(j), \dots) :: \text{stmts}, V)$

The result of the calling of TRStmts respectively TRStmt for the LocalVarDecl - and the Assign -statement is $\{ (\sigma_{1.4} \cup [\alpha_7 \mapsto \text{int}], \text{void}, V_{1.4} \cup \{j : \text{int}\}) \}$
In the following let

$$\sigma_{1.4.1} = \sigma_{1.4} \cup [\alpha_7 \mapsto \text{int}],$$

and

$$V_{1.4.1} = V_{1.4} \cup \{j : \text{int}\}.$$

(1.4.1.1) $\text{TRStmt}(\sigma_{1.4.1}, V_{1.4.1}, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$

With the result TRStmt was called for the next statement by an again calling of TRStmts .
The result is (cp. section B.1):

$$\begin{aligned} \sigma_{1.4.1.1}^1 &= [\alpha_{13} \mapsto \text{Integer}, \alpha_{11} \mapsto \text{Integer}, \alpha_{10} \mapsto \text{Vector<Integer>}, \\ &\quad \alpha_2 \mapsto \text{Vector<Vector<Integer>>}, \alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \\ &\quad \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector<Integer>}, \alpha_6 \mapsto \text{Vector<Integer>}, \alpha_7 \mapsto \text{int}] \\ \sigma_{1.4.1.1}^2 &= [\alpha_{14} \mapsto \text{Integer}, \alpha_{12} \mapsto \text{Integer}, \alpha_2 \mapsto \text{Matrix}, \alpha_{10} \mapsto \text{Vector<Integer>}, \\ &\quad \alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector<Integer>}, \\ &\quad \alpha_6 \mapsto \text{Vector<Integer>}, \alpha_7 \mapsto \text{int}] \\ V_{1.4.1.1}^1 &= \{ \text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector<Integer>}, v2 : \text{Vector<Integer>}, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Vector<Vector<Integer>>} \rightarrow \alpha_1, \\ &\quad m : \text{Vector<Vector<Integer>>} \} \\ V_{1.4.1.1}^2 &= \{ \text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector<Integer>}, v2 : \text{Vector<Integer>}, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Matrix} \rightarrow \alpha_1, m : \text{Matrix} \}. \end{aligned}$$

The type reconstruction of the WhileStmt is considered in section B.1.

(1.4.1.1.1) $\text{TRStmt}(\sigma_{1.4.1.1}^1, V_{1.4.1.1}^1, \text{MethodCall}(\text{LocalOrFieldVar}(\text{ret}), \text{addElement}(\text{LocalOrFieldVar}(v2))))$

The application of TRExp , TRtuple , and TRMCallApp leads to the result:

$$\{ ([\alpha_{15} \mapsto \text{Vector<Integer>}] \circ \sigma_{1.4.1.1}^1, \text{void}, V_{1.4.1.1}^1) \}.$$

(1.4.1.1.2) $\text{TRStmt}(\sigma_{1.4.1.1}^2, V_{1.4.1.1}^2, \text{MethodCall}(\text{LocalOrFieldVar}(\text{ret}), \text{addElement}(\text{LocalOrFieldVar}(v2))))$

Analogous the result is given as:

$$\{ ([\alpha_{16} \mapsto \text{Vector<Integer>}] \circ \sigma_{1.4.1.1}^2, \text{void}, V_{1.4.1.1}^2) \}.$$

(1) $\text{TRStmt}(\sigma, V, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$

As the call of TRStmts for

$\text{Assign}(\text{LocalOrFieldVar}(i), \text{Add}(\text{LocalOrFieldVar}(i), \text{IntLiteral}(1)))$

only removes the type assumptions for the local variables, the result is given as

$$\{ ([\alpha_{15} \mapsto \text{Vector<Integer>}] \circ \sigma_{1.4.1.1}^1, \text{void}, V_{1.4.1.1}^1 \setminus \{v1 : \text{Vector<Integer>}, v2 : \text{Vector<Integer>}, j : \text{int}\}), \\ ([\alpha_{14} \mapsto \text{Vector<Integer>}] \circ \sigma_{1.4.1.1}^2, \text{void}, V_{1.4.1.1}^2 \setminus \{v1 : \text{Vector<Integer>}, v2 : \text{Vector<Integer>}, j : \text{int}\}) \}$$

B.1 Type reconstruction of the middle while-loop

In this section we complete the type reconstruction example from appendix B. The Java 5.0 program, which types are reconstructed is given as:

```
class Matrix extends Vector<Vector<Integer>> {

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

```

        k++;
    }
    v2.addElement(new Integer(erg));
    j++;
}
ret.addElement(v2);
i++;
}
return ret;
}

```

The corresponding abstract syntax is given in section 5.4.2.

Only the emphasized middle `while`-loop is considered in this section. The framework is considered in appendix refsec:whilefirst. From this follows for the type unifier

$$\sigma = [\alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_6 \mapsto \text{Vector}\langle \text{Integer} \rangle, \alpha_7 \mapsto \text{int}] (= \sigma_{1.4.1}),$$

for the set of local type assumptions

$$V = \{\text{mul} : \alpha_2 \rightarrow \alpha_1, \text{m} : \alpha_2, \text{ret} : \text{Matrix}, \text{i} : \text{int}, \text{v1} : \text{Vector}\langle \text{Integer} \rangle, \text{v2} : \text{Vector}\langle \text{Integer} \rangle, \text{j} : \text{int}\} (= V_{1.4.1})$$

and for the set of type assumptions of other classes

$$A_{\forall a. \text{Vector}\langle a \rangle} = \{\text{elementAt} : \text{int} \rightarrow a, \text{addElement} : a \rightarrow \text{void}, \text{size} : \rightarrow \text{int}\}$$

and

$$A_{\text{Integer}} = \{\langle \text{init} \rangle_{\text{Integer}} : \text{int} \rightarrow \text{Integer}, \text{intValue} : \rightarrow \text{int}\}.$$

Now we reconstruct the types for the middle `while`-loop:

$$(1) \text{ TRStmt}(\sigma, V, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$$

The types for the expression

- $\text{Less}(\text{LocalOrFieldVar}(j), \text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{size}()))$

and the statements

- $\text{LocalVarDecl}(\text{erg})$
- $\text{Assign}(\text{LocalOrFieldVar}(\text{erg}), \text{IntLiteral}(0))$
- $\text{LocalVarDecl}(k)$

- $\text{Assign}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(0))$

are reconstructed analogously as in appendix B.

The result is $\{(\sigma \cup [\alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}], \text{void}, V \cup \{\text{erg} : \text{int}, k : \text{int}\})\}$.

In the following let

$$\sigma_1 = [\alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}] \circ \sigma,$$

and

$$V_1 = V \cup \{\text{erg} : \text{int}, k : \text{int}\}$$

Then TRStmt for the next `while`-loop is called.

$$(1.1) \text{ TRStmt}(\sigma_1, V_1, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$$

The result of

$$\text{TRExp}(\sigma_1, V_1, \text{Less}(\text{LocalOrFieldVar}(k), \text{MethodCall}(\text{LocalOrFieldVar}(v1), \text{size}())))$$

is $\{(\sigma_1, \text{boolean}, V_1)\}$.

$$(1.1.1) \text{ TRStmt}(\sigma_1, V_1, \text{Assign}(\text{LocalOrFieldVar}(\text{erg}), \dots))$$

With the result TRStmts was called again for the next statement. Then TRStmt is called. In the function TRStmt for the `Assign`-statement, the function TRExp is called. In TRExp the function TRtuple is called.

$$(1.1.1.1) \text{ TRtuple}(\sigma_1, V_1, (\text{LocalOrFieldVar}(\text{erg}), e_2))$$

where $e_2 = \text{Add}(\text{LocalOrFieldVar}(\text{erg}),$

$$\begin{aligned} & \text{Mul}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(v1), \\ & \quad \text{elementAt}(\text{LocalOrFieldVar}(k)) \text{ }), \\ & \quad \text{intValue}()) \text{ }), \\ & \text{MethodCall}(\text{MethodCall}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(m), \\ & \quad \text{elementAt}(\text{LocalOrFieldVar}(k)) \text{ }), \\ & \quad \text{elementAt}(\text{LocalOrFieldVar}(j)) \text{ }), \\ & \quad \text{intValue}()) \text{ }) \end{aligned}$$

The result of $\text{TRmultiply}(\sigma_1, V_1, \text{LocalOrFieldVar}(\text{erg}))$ is $\{(\sigma_1, \text{int}, V_1)\}$.

With this result the function TRtuple is called again. In TRtuple the function TRExp is called by the function TRmultiply .

(1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e_2)$

The result of $\text{TRExp}(\sigma_1, V_1, \text{LocalOrFieldVar}(\text{erg}))$ is $\{(\sigma_1, \text{int}, V_1)\}$
Then TRExp is called again.

(1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, \text{Mul}(\text{MethodCall}(\dots), \text{MethodCall}(\dots)))$

The result of first argument

$\text{TRExp}(\sigma_1, V_1, \text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(v_1),$
 $\text{elementAt}(\text{LocalOrFieldVar}(k))$),
 $\text{intValue}())$)

is $\{(\sigma_1, \text{int}, V_1)\}$

For the second argument TRExp is called again.

(1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e)$

where $e = \text{MethodCall}(\text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(m),$
 $\text{elementAt}(\text{LocalOrFieldVar}(k))$),
 $\text{elementAt}(\text{LocalOrFieldVar}(j))$),
 $\text{intValue}())$)

With TRtuple and TRmultiply the function TRExp is called.

(1.1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e')$

where $e' = \text{MethodCall}(\text{MethodCall}(\text{LocalOrFieldVar}(m),$
 $\text{elementAt}(\text{LocalOrFieldVar}(k))$),
 $\text{elementAt}(\text{LocalOrFieldVar}(j))$))

In TRExp the function TRtuple is called.

LAYOUT STIMMT NOCH NICHT

(1.1.1.1.1.1.1.1.1.1) $\text{TRtuple}((\sigma_1, \epsilon, V_1), e'')$

with $e'' = (\text{MethodCall}(\text{LocalOrFieldVar}(m), \text{elementAt}(\text{LocalOrFieldVar}(k))),$
 $\text{elementAt}(\text{LocalOrFieldVar}(j))$)

TRExp is called again from TRmultiply for the first expression.

(1.1.1.1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, \text{MethodCall}(\text{LocalOrFieldVar}(m),$
 $\text{elementAt}(\text{LocalOrFieldVar}(k))$))

First, in TRExp the function TRtuple is called for the receiver and the argument expression of the Method -expression.

(1.1.1.1.1.1.1.1.1.1.1.1) $\text{TRtuple}((\sigma_1, \epsilon, V_1), (\text{LocalOrFieldVar}(m) \text{ LocalOrFieldVar}(k)))$

By the functions TRmultiply and TRExp follows the result $\{((\sigma_1, (\alpha_2 \text{ int}), V_1))\}$.

(1.1.1.1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, \text{MethodCall}(\text{LocalOrFieldVar}(m),$
 $\text{elementAt}(\text{LocalOrFieldVar}(k))$))

Second, with the result $\{((\sigma_1, (\alpha_2 \text{ int}), V_1))\}$ the function TRMCallApp is called.

(1.1.1.1.1.1.1.1.1.1.2) $\text{TRMCallApp}((\sigma_1, (\alpha_2 \text{ int}), V_1), \text{elementAt}(\text{LocalOrFieldVar}(k)))$

- $A_{\forall a. \text{Vector}\langle a \rangle} = \{ \text{elementAt} : \text{int} \rightarrow a$
 $\text{addElement} : a \rightarrow \text{void}$
 $\text{size} : \rightarrow \text{int} \}$
- $\sigma_{\text{new}}^1 = []$
- $(\bar{\theta}_{1,0}, \bar{\theta}_{1,1} \rightarrow \bar{\theta}_1) = \text{fresh}(\text{Vector}\langle a \rangle, \text{int} \rightarrow a) = (\text{Vector}\langle \alpha_{10} \rangle, \text{int} \rightarrow \alpha_{10})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}((\alpha_2 \text{ int}), (\text{Vector}\langle \alpha_{10} \rangle, \text{int}))$
 $= \{ [\alpha_2 \mapsto \text{Vector}\langle \alpha_{10} \rangle], [\alpha_2 \mapsto \text{Matrix}, \alpha_{10} \mapsto \text{Vector}\langle \text{Integer} \rangle] \}$
- $\text{substset}_1 = \{ ([\alpha_2 \mapsto \text{Vector}\langle \alpha_{10} \rangle] \circ \sigma_1,$
 $\alpha_{10},$
 $V_1 \setminus \{ \text{mul} : \alpha_2 \rightarrow \alpha_1, m : \alpha_2 \}$
 $\cup \{ \text{mul} : \text{Vector}\langle \alpha_{10} \rangle \rightarrow \alpha_1, m : \text{Vector}\langle \alpha_{10} \rangle \}$,
 $([\alpha_2 \mapsto \text{Matrix}, \alpha_{10} \mapsto \text{Vector}\langle \text{Integer} \rangle] \circ \sigma_1,$
 $\text{Vector}\langle \text{Integer} \rangle,$
 $V_1 \setminus \{ \text{mul} : \alpha_2 \rightarrow \alpha_1, m : \alpha_2 \} \cup \{ \text{mul} : \text{Matrix} \rightarrow \alpha_1, m : \text{Matrix} \}) \}$
 $= \text{ret}_{\text{Vector}}$
 $= \text{case}_1$
- $\text{case}_2 = \emptyset$

In the following let

$$\begin{aligned} \sigma_{\text{TRMCallApp}}^1 &= [\alpha_2 \mapsto \text{Vector}\langle \alpha_{10} \rangle] \circ \sigma_1 \\ \sigma_{\text{TRMCallApp}}^2 &= [\alpha_2 \mapsto \text{Matrix}, \alpha_{10} \mapsto \text{Vector}\langle \text{Integer} \rangle] \circ \sigma_1 \\ V_{\text{TRMCallApp}}^1 &= V_1 \setminus \{ \text{mul} : \alpha_2 \rightarrow \alpha_1, m : \alpha_2 \} \\ &\quad \cup \{ \text{mul} : \text{Vector}\langle \alpha_{10} \rangle \rightarrow \alpha_1, m : \text{Vector}\langle \alpha_{10} \rangle \} \\ V_{\text{TRMCallApp}}^2 &= V_1 \setminus \{ \text{mul} : \alpha_2 \rightarrow \alpha_1, m : \alpha_2 \} \cup \{ \text{mul} : \text{Matrix} \rightarrow \alpha_1, m : \text{Matrix} \}. \end{aligned}$$

The result is then

$$\{ (\sigma_{\text{TRMCallApp}}^1, \alpha_{10}, V_{\text{TRMCallApp}}^1), (\sigma_{\text{TRMCallApp}}^2, \text{Vector}\langle \text{Integer} \rangle, V_{\text{TRMCallApp}}^2) \}.$$

(1.1.1.1.1.1.1.1.1) $\text{TRtuple}(\sigma_1, \epsilon, V_1, e'')$

Now, the function TRtuple is called twice, for each triple of the result respectively.

(1.1.1.1.1.1.1.1.2) $\text{TRtuple}(\sigma_{\text{TRMCallApp}}^1, \alpha_{10}, V_{\text{TRMCallApp}}^1, \text{elementAt}(\text{LocalOrFieldVar}(j)))$

With TRmultiply and TRExp the result is given as

$$\{(\sigma_{\text{TRMCallApp}}^1, (\alpha_{10} \text{ int}), V_{\text{TRMCallApp}}^1)\}.$$

(1.1.1.1.1.1.1.1.3) $\text{TRtuple}(\sigma_{\text{TRMCallApp}}^2, \text{Vector<Integer>}, V_{\text{TRMCallApp}}^2, \text{elementAt}(\text{LocalOrFieldVar}(j)))$

With TRmultiply and TRExp the result is given as

$$\{(\sigma_{\text{TRMCallApp}}^2, (\text{Vector<Integer> int}), V_{\text{TRMCallApp}}^2)\}.$$

(1.1.1.1.1.1.1.1.1) $\text{TRtuple}(\sigma_1, \epsilon, V_1, e'')$

Then the result is given as

$$\{(\sigma_{\text{TRMCallApp}}^1, (\alpha_{10} \text{ int}), V_{\text{TRMCallApp}}^1, (\sigma_{\text{TRMCallApp}}^2, (\text{Vector<Integer> int}), V_{\text{TRMCallApp}}^2))\}.$$

(1.1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e')$

For each triple of this result, the function TRMCallApp is called.

(1.1.1.1.1.1.1.1.2) $\text{TRMCallApp}(\sigma_{\text{TRMCallApp}}^1, (\alpha_{10} \text{ int}), V_{\text{TRMCallApp}}^1, \text{elementAt}(\text{LocalOrFieldVar}(j)))$

$$\begin{aligned} - A_{\forall a. \text{Vector}<a>} &= \{ \text{elementAt} : \text{int} \rightarrow a \\ &\quad \text{addElement} : a \rightarrow \text{void} \\ &\quad \text{size} : \rightarrow \text{int} \} \\ - (\bar{\theta}_{1,0}, \bar{\theta}_{1,1} \rightarrow \bar{\theta}_1) &= \text{fresh}(\text{Vector}<a>, \text{int} \rightarrow a) = (\text{Vector}<\alpha_{11}>, \text{int} \rightarrow \alpha_{11}) \\ - \text{unify}_1 &= \text{TUnify}_{\leq^*}((\alpha_{10} \text{ int}), (\text{Vector}<\alpha_{11}>, \text{int})) \\ &= \{ [\alpha_{10} \mapsto \text{Vector}<\alpha_{11}>], [\alpha_{10} \mapsto \text{Matrix}, \alpha_{11} \mapsto \text{Vector}<\text{Integer}>] \} \\ - \text{subset}_1 &= \{ ([\alpha_{10} \mapsto \text{Vector}<\alpha_{11}>] \circ \sigma_{\text{TRMCallApp}}^1, \\ &\quad \alpha_{11}, \\ &\quad V_{\text{TRMCallApp}}^1 \setminus \{ \text{mul} : \text{Vector}<\alpha_{10}> \rightarrow \alpha_1, m : \text{Vector}<\alpha_{10}> \} \\ &\quad \cup \{ \text{mul} : \text{Vector}<\text{Vector}<\alpha_{11}>> \rightarrow \alpha_1, m : \text{Vector}<\text{Vector}<\alpha_{11}>> \} \}, \\ &\quad ([\alpha_{10} \mapsto \text{Matrix}, \alpha_{11} \mapsto \text{Vector}<\text{Integer}>] \circ \sigma_{\text{TRMCallApp}}^1, \\ &\quad \text{Vector}<\text{Integer}>, \\ &\quad V_{\text{TRMCallApp}}^1 \setminus \{ \text{mul} : \text{Vector}<\alpha_{10}> \rightarrow \alpha_1, m : \text{Vector}<\alpha_{10}> \} \\ &\quad \cup \{ \text{mul} : \text{Vector}<\text{Matrix}> \rightarrow \alpha_1, m : \text{Vector}<\text{Matrix}> \} \} \} \\ &= \text{ret}_{\text{Vector}} \\ &= \text{case}_1 \end{aligned}$$

– $\text{case}_2 = \emptyset$

In the following let

$$\begin{aligned} \sigma_{1.1.1.1.1.1.1.1.2}^1 &= [\alpha_{10} \mapsto \text{Vector}<\alpha_{11}>] \circ \sigma_{\text{TRMCallApp}}^1 \\ \sigma_{1.1.1.1.1.1.1.1.2}^2 &= [\alpha_{10} \mapsto \text{Matrix}, \alpha_{11} \mapsto \text{Vector}<\text{Integer}>] \circ \sigma_{\text{TRMCallApp}}^1 \\ V_{1.1.1.1.1.1.1.1.2}^1 &= V_{\text{TRMCallApp}}^1 \setminus \{ \text{mul} : \text{Vector}<\alpha_{10}> \rightarrow \alpha_1, m : \text{Vector}<\alpha_{10}> \} \\ &\quad \cup \{ \text{mul} : \text{Vector}<\text{Vector}<\alpha_{11}>> \rightarrow \alpha_1, m : \text{Vector}<\text{Vector}<\alpha_{11}>> \} \\ V_{1.1.1.1.1.1.1.1.2}^2 &= V_{\text{TRMCallApp}}^1 \setminus \{ \text{mul} : \text{Vector}<\alpha_{10}> \rightarrow \alpha_1, m : \text{Vector}<\alpha_{10}> \} \\ &\quad \cup \{ \text{mul} : \text{Vector}<\text{Matrix}> \rightarrow \alpha_1, m : \text{Vector}<\text{Matrix}> \}. \end{aligned}$$

Then, the result is given as

$$\{(\sigma_{1.1.1.1.1.1.1.1.2}^1, \alpha_{11}, V_{1.1.1.1.1.1.1.1.2}^1), (\sigma_{1.1.1.1.1.1.1.1.2}^2, \text{Vector}<\text{Integer}>, V_{1.1.1.1.1.1.1.1.2}^2)\}.$$

(1.1.1.1.1.1.1.1.3) $\text{TRMCallApp}(\sigma_{\text{TRMCallApp}}^2, (\text{Vector}<\text{Integer}> \text{int}), V_{\text{TRMCallApp}}^2, \text{elementAt}(\text{LocalOrFieldVar}(j)))$

$$\begin{aligned} - A_{\forall a. \text{Vector}<a>} &= \{ \text{elementAt} : \text{int} \rightarrow a \\ &\quad \text{addElement} : a \rightarrow \text{void} \\ &\quad \text{size} : \rightarrow \text{int} \} \\ - (\bar{\theta}_{1,0}, \bar{\theta}_{1,1} \rightarrow \bar{\theta}_1) &= \text{fresh}(\text{Vector}<a>, \text{int} \rightarrow a) = (\text{Vector}<\alpha_{12}>, \text{int} \rightarrow \alpha_{12}) \\ - \text{unify}_1 &= \text{TUnify}_{\leq^*}((\text{Vector}<\text{Integer}> \text{int}), (\text{Vector}<\alpha_{12}>, \text{int})) \\ &= \{ [\alpha_{12} \mapsto \text{Integer}] \} \\ - \text{subset}_1 &= \{ ([\alpha_{12} \mapsto \text{Integer}] \circ \sigma_{\text{TRMCallApp}}^2, \\ &\quad \text{Integer}, \\ &\quad V_{\text{TRMCallApp}}^2) \} \\ &= \text{ret}_{\text{Vector}} \\ &= \text{case}_1 \\ - \text{case}_2 &= \emptyset \end{aligned}$$

In the following let

$$\begin{aligned} \sigma_{1.1.1.1.1.1.1.1.3} &= [\alpha_{12} \mapsto \text{Integer}] \circ \sigma_{\text{TRMCallApp}}^2 \\ V_{1.1.1.1.1.1.1.1.3} &= V_{\text{TRMCallApp}}^2 \end{aligned}$$

Then, the result is given as:

$$\{(\sigma_{1.1.1.1.1.1.1.1.3}, \text{Integer}, V_{1.1.1.1.1.1.1.1.3})\}.$$

(1.1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e')$

The result is given as:

$$\begin{aligned} &\{(\sigma_{1.1.1.1.1.1.1.1.2}^1, \alpha_{11}, V_{1.1.1.1.1.1.1.1.2}^1), \\ &\quad (\sigma_{1.1.1.1.1.1.1.1.2}^2, \text{Vector}<\text{Integer}>, V_{1.1.1.1.1.1.1.1.2}^2), \\ &\quad (\sigma_{1.1.1.1.1.1.1.1.3}, \text{Integer}, V_{1.1.1.1.1.1.1.1.3})\}. \end{aligned}$$

(1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e)$ For each triple of this result, the function TRMCallApp is called.(1.1.1.1.1.1.1.2) $\text{TRMCallApp}((\sigma_{1.1.1.1.1.1.1.2}^1, \alpha_{11}, V_{1.1.1.1.1.1.1.2}^1), \text{intValue}())$

- $A_{\text{Integer}} = \{ \langle \text{init} \rangle_{\text{Integer}} : \text{int} \rightarrow \text{Integer} \mid \text{intValue} : \rightarrow \text{int} \}.$
- $(\bar{\theta}_{1,0}, \bar{\theta}_1) = (\text{Integer}, \text{int})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\alpha_{11}, \text{Integer}) = \{ [\alpha_{11}, \mapsto \text{Integer}] \}$
- $\text{subset}_1 = \{ ([\alpha_{11} \mapsto \text{Integer}] \circ \sigma_{1.1.1.1.1.1.1.2}^1, \text{int}, V_{1.1.1.1.1.1.1.2}^1 \setminus \{ \text{mul} : \text{Vector} \langle \text{Vector} \langle \alpha_{11} \rangle \rangle \rightarrow \alpha_1, m : \text{Vector} \langle \text{Vector} \langle \alpha_{11} \rangle \rangle \} \cup \{ \text{mul} : \text{Vector} \langle \text{Vector} \langle \text{Integer} \rangle \rangle \rightarrow \alpha_1, m : \text{Vector} \langle \text{Vector} \langle \text{Integer} \rangle \rangle \} \} \}$
 $= \text{ret}_{\text{Integer}}$
 $= \text{case}_1$
- $\text{case}_2 = \emptyset$

In the following let

$$\begin{aligned} \sigma_{1.1.1.1.1.1.1.2} &= [\alpha_{11} \mapsto \text{Integer}] \circ \sigma_{1.1.1.1.1.1.1.2}^1 \\ V_{1.1.1.1.1.1.1.2} &= V_{1.1.1.1.1.1.1.2}^1 \setminus \{ \text{mul} : \text{Vector} \langle \text{Vector} \langle \alpha_{11} \rangle \rangle \rightarrow \alpha_1, m : \text{Vector} \langle \text{Vector} \langle \alpha_{11} \rangle \rangle \} \\ &\quad \cup \{ \text{mul} : \text{Vector} \langle \text{Vector} \langle \text{Integer} \rangle \rangle \rightarrow \alpha_1, m : \text{Vector} \langle \text{Vector} \langle \text{Integer} \rangle \rangle \} \end{aligned}$$

Then, the result is given as:

$$\{ (\sigma_{1.1.1.1.1.1.1.2}, \text{int}, V_{1.1.1.1.1.1.2}) \}.$$

(1.1.1.1.1.1.1.3) $\text{TRMCallApp}((\sigma_{1.1.1.1.1.1.1.2}^2, \text{Vector} \langle \text{Integer} \rangle, V_{1.1.1.1.1.1.1.2}^2), \text{intValue}())$

- $A_{\text{Integer}} = \{ \langle \text{init} \rangle_{\text{Integer}} : \text{int} \rightarrow \text{Integer} \mid \text{intValue} : \rightarrow \text{int} \}.$
- $(\bar{\theta}_{1,0}, \bar{\theta}_1) = (\text{Integer}, \text{int})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\text{Vector} \langle \text{Integer} \rangle, \text{Integer}) = \emptyset$
- $\text{subset}_1 = \emptyset = \text{ret}_{\text{Integer}}$
- $\text{case}_1 = \emptyset$
- $\text{case}_2 = \emptyset$

This means that the result is \emptyset .(1.1.1.1.1.1.1.4) $\text{TRMCallApp}((\sigma_{1.1.1.1.1.1.1.3}, \text{Integer}, V_{1.1.1.1.1.1.1.3}), \text{intValue}())$

- $A_{\text{Integer}} = \{ \langle \text{init} \rangle_{\text{Integer}} : \text{int} \rightarrow \text{Integer} \mid \text{intValue} : \rightarrow \text{int} \}.$
- $(\bar{\theta}_{1,0}, \bar{\theta}_1) = (\text{Integer}, \text{int})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\text{Integer}, \text{Integer}) = \{ [] \}$
- $\text{subset}_1 = \{ (\sigma_{1.1.1.1.1.1.1.3}, \text{int}, V_{1.1.1.1.1.1.1.3}^1) \}$
 $= \text{ret}_{\text{Integer}}$
 $= \text{case}_1$
- $\text{case}_2 = \emptyset$

Then, the result is given as:

$$\{ (\sigma_{1.1.1.1.1.1.1.3}, \text{int}, V_{1.1.1.1.1.1.1.3}) \}.$$

(1.1.1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e)$ the result is then given as the union of the results of the function calls of TRMCallApp :

$$\{ (\sigma_{1.1.1.1.1.1.1.2}, \text{int}, V_{1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.1.3}, \text{int}, V_{1.1.1.1.1.1.3}) \}.$$

(1.1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, \text{Mul}(\text{MethodCall}(\dots), \text{MethodCall}(\dots)))$ As the result of first argument was $\{ (\sigma_1, \text{int}, V_1) \}$, the result of the Mul application is

$$\{ (\sigma_{1.1.1.1.1.1.2}, \text{int}, V_{1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.3}, \text{int}, V_{1.1.1.1.1.1.3}) \}.$$

(1.1.1.1.1) $\text{TRExp}(\sigma_1, V_1, e_2)$ Analogous, follows that the result of the Add -application is also:

$$\{ (\sigma_{1.1.1.1.1.1.2}, \text{int}, V_{1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.3}, \text{int}, V_{1.1.1.1.1.1.3}) \}.$$

(1.1.1.1) $\text{TRtuple}(\sigma_1, V_1, (\text{LocalOrFieldVar}(\text{erg}), e_2))$ The result is determined by the function TRmultiply :

$$\{ (\sigma_{1.1.1.1.1.1.2}, (\text{int int}), V_{1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.3}, (\text{int int}), V_{1.1.1.1.1.1.3}) \}.$$

As in TRExp for the Assign -expression the types int and int are unified, in the result the unifier and the set of type assumptions is not changed:

$$\{ (\sigma_{1.1.1.1.1.1.2}, \text{void}, V_{1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.3}, \text{void}, V_{1.1.1.1.1.1.3}) \}.$$

(1.1.2) $\text{TRStmt}(\sigma_{1.1.1.1.1.1.1.2}, V_{1.1.1.1.1.1.1.2}, \text{Assign}(\text{LocalOrFieldVar}(k), \text{Add}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(1))))$

With the first result TRStmts was called again for the next statement. Then TRStmt is called.

As $(k : \text{int}) \in V_{1.1.1.1.1.1.1.2}$ the result is:

$$\{(\sigma_{1.1.1.1.1.1.1.2}, \text{void}, V_{1.1.1.1.1.1.1.2})\}.$$

(1.1.3) $\text{TRStmt}(\sigma_{1.1.1.1.1.1.1.2}, V_{1.1.1.1.1.1.1.2}, \text{Assign}(\text{LocalOrFieldVar}(k), \text{Add}(\text{LocalOrFieldVar}(k), \text{IntLiteral}(1))))$

With the second result TRStmts was called again for the same statement. Then TRStmt is called.

The result is analogous:

$$\{(\sigma_{1.1.1.1.1.1.1.3}, \text{void}, V_{1.1.1.1.1.1.1.3})\}.$$

(1.1) $\text{TRStmt}(\sigma_1, V_1, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$

This means that the result is again:

$$\{(\sigma_{1.1.1.1.1.1.1.2}, \text{void}, V_{1.1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.1.3}, \text{void}, V_{1.1.1.1.1.1.1.3})\}.$$

(1.2) $\text{TRStmt}(\sigma_{1.1.1.1.1.1.1.2}, V_{1.1.1.1.1.1.1.2}, \text{MethodCall}(\text{LocalOrFieldVar}(v2), \text{addElement}(\text{New}(\text{Integer}, (\text{LocalOrFieldVar}(\text{erg}))))))$

First TRtuple is called for $\text{LocalOrFieldVar}(v2)$ and $\text{New}(\text{Integer}, (\text{LocalOrFieldVar}(\text{erg})))$. The result is:

$$\{([\alpha_{13} \mapsto \text{Integer}] \circ \sigma_{1.1.1.1.1.1.1.2}, (\text{Vector}<\text{Integer}> \text{Integer}), V_{1.1.1.1.1.1.1.2})\}.$$

Then for this result TRMCallApp is called. The result is:

$$\{(\sigma_{1.1.1.1.1.1.1.2}, \text{void}, V_{1.1.1.1.1.1.1.2})\}.$$

(1.3) $\text{TRStmt}(\sigma_{1.1.1.1.1.1.1.3}, V_{1.1.1.1.1.1.1.3}, \text{MethodCall}(\text{LocalOrFieldVar}(v2), \text{addElement}(\text{New}(\text{Integer}, (\text{LocalOrFieldVar}(\text{erg}))))))$

Analogous the result is:

$$\{([\alpha_{14} \mapsto \text{Integer}] \circ \sigma_{1.1.1.1.1.1.1.3}, (\text{Vector}<\text{Integer}> \text{Integer}), V_{1.1.1.1.1.1.1.3})\}.$$

(1) $\text{TRStmt}(\sigma, V, \text{WhileStmt}(\text{Less}(\dots), \text{Block}(\dots)))$

As nothing happens during the type reconstruction of

$$\text{Assign}(\text{LocalOrFieldVar}(j), \text{Add}(\text{LocalOrFieldVar}(j), \text{IntLiteral}(1)))$$

the result of the type reconstruction of the middle while-loop is:

$$\{(\sigma_{1.1.1.1.1.1.1.2}, \text{void}, V_{1.1.1.1.1.1.1.2}), (\sigma_{1.1.1.1.1.1.1.3}, \text{void}, V_{1.1.1.1.1.1.1.3})\}.$$

with

$$\begin{aligned} \sigma_{1.1.1.1.1.1.1.2} &= [\alpha_{13} \mapsto \text{Integer}, \alpha_{11} \mapsto \text{Integer}, \alpha_{10} \mapsto \text{Vector}<\text{Integer}>, \\ &\quad \alpha_2 \mapsto \text{Vector}<\text{Vector}<\text{Integer}>>, \alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \\ &\quad \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector}<\text{Integer}>, \alpha_6 \mapsto \text{Vector}<\text{Integer}>, \alpha_7 \mapsto \text{int}] \\ \sigma_{1.1.1.1.1.1.1.3} &= [\alpha_{14} \mapsto \text{Integer}, \alpha_{12} \mapsto \text{Integer}, \alpha_2 \mapsto \text{Matrix}, \alpha_{10} \mapsto \text{Vector}<\text{Integer}>, \\ &\quad \alpha_8 \mapsto \text{int}, \alpha_9 \mapsto \text{int}, \alpha_3 \mapsto \text{Matrix}, \alpha_4 \mapsto \text{int}, \alpha_5 \mapsto \text{Vector}<\text{Integer}>, \\ &\quad \alpha_6 \mapsto \text{Vector}<\text{Integer}>, \alpha_7 \mapsto \text{int}] \\ V_{1.1.1.1.1.1.1.2} &= \{\text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector}<\text{Integer}>, v2 : \text{Vector}<\text{Integer}>, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Vector}<\text{Vector}<\text{Integer}>> \rightarrow \alpha_1, \\ &\quad m : \text{Vector}<\text{Vector}<\text{Integer}>>\} \\ V_{1.1.1.1.1.1.1.3} &= \{\text{ret} : \text{Matrix}, i : \text{int}, v1 : \text{Vector}<\text{Integer}>, v2 : \text{Vector}<\text{Integer}>, \\ &\quad j : \text{int}, \text{erg} : \text{int}, k : \text{int}, \text{mul} : \text{Matrix} \rightarrow \alpha_1, m : \text{Matrix}\}. \end{aligned}$$

Appendix C

Type reconstruction example factorial

In this section we give another example, which shows how inherited fields and recursive method calls are handled. The Java 5.0 program, which types are reconstructed is given as:

```
class Int {
    int i;

    Int (int i) {
        this.i = i;
    }

    Int mul (Int J) {
        return new Int(j.i * this.i);
    }
}

class factorial extends Int {

    fac() {
        if (i==1) return new Int(1);
        else {
            return this.mul(new factorial(i-1).fac());
        }
    }
}
```

The corresponding abstract syntax for the class `factorial` is given as:

```
absfactorial =
class( ClassName( factorial ), extends( Int ),
    Method( <init>_factorial, factorial, (i : int),
        Block( super( LocalOrFieldVar(i) ) ) )
    Method( fac, ∅, () ),
```

```
Block(
    IfStmt(
        Equal( InstVar( this, i ), IntLiteral(1) ),
        Block( Return( New( Int, (IntLiteral(1)) ) ) ),
        Block( Return( MethodCall(
            this,
            mul( MethodCall(
                New( factorial,
                    Minus( InstVar( this, i ),
                        IntLiteral(1) ) ),
                fac() ) ) ) ) ) ) ) ) ) )
```

For the type reconstruction of the method `fac` of the class `factorial` we need a set of type assumptions $A = \{ A_{\text{Int}} \}$ with

$$A_{\text{Int}} = \{ i : \text{int} \\ \quad \text{<init>}_{\text{Int}} : \text{int} \rightarrow \text{Int} \\ \quad \text{mul} : \text{Int} \rightarrow \text{Int} \}$$

The type of the actual class `act.cl` is given as `factorial` and the finite closure $\text{FC}(\leq)$ is given as $\{ \text{factorial} \leq^* \text{Int} \}$.

Now we can start and call the `TRprog`.

(1) `TRprog(A, absfactorial)`

First, we apply the function `NewTVar` to `absfactorial`, which instances new type variables for each parameter type and each return type. In this example we denote the fresh type variables by the greek letters $\alpha_1, \alpha_2, \dots$, as above.

The result is given as:

```
NewTVar( absfactorial ) =
class( ClassName( factorial ), extends( Int ),
    Method( <init>_factorial, factorial, (i : int), Block( B1 ) )
    Method( fac, α1, (), Block( B2 ) ) )
```

It holds:

$$V_{\text{fields_methods}} = \{ \text{<init>}_{\text{factorial}} : \text{int} \rightarrow \text{factorial}, \text{fac} : \rightarrow \alpha_1 \}.$$

The sets of the parameters of the methods given as

$$\begin{aligned} V_1 &= \{ i : \text{int} \} . \\ V_2 &= \emptyset \end{aligned}$$

Then, the function `TRstart` is called.

(1.1) $\text{TRstart}(\text{Block}(Bl_1), \text{Block}(Bl_2), (V_1, V_2), V_{fields_methods})$

As the constructor $\langle \text{init} \rangle_{\text{factorial}}$ consists no type variable, we consider only the type reconstruction of the method fac .

TRstart calls by TRNextMeth the function TRstmt . In TRstmt only the Block constructor is removed and the function TRstmts is called.

(1.1.1) $\text{TRstmts}([], V_{fields_methods}, Bl_2)$

In TRstmts for the first statement TRstmt is called again.

(1.1.1.1) $\text{TRstmt}([], V_{fields_methods}, \text{IfStmt}(e_0, e_1, e_2))$

where

$e_0 = \text{Equal}(\text{InstVar}(\text{this}, i), \text{IntLiteral}(1))$,
 $e_1 = \text{Block}(\text{Return}(\text{New}(\text{Int}, (\text{IntLiteral}(1)))))$, and
 $e_2 = \text{Block}(\text{Return}(\text{MethodCall}(\text{this},$
 $\quad \text{mul}(\text{MethodCall}(\text{New}(\text{factorial},$
 $\quad \quad \text{Minus}(\text{InstVar}(\text{this}, i),$
 $\quad \quad \quad \text{IntLiteral}(1))$),
 $\quad \quad \text{fac}())$))

First TRExp is called for e_0 .

(1.1.1.1.1) $\text{TRExp}([], V_{fields_methods}, e_0)$

In TRExp for Equal the function TRExp is called again for both arguments.

(1.1.1.1.1.1) $\text{TRExp}([], V_{fields_methods}, \text{InstVar}(\text{this}, i))$

It holds

$\text{TRExp}([], V_{fields_methods}, \text{this}) = \{([], \text{factorial}, V_{fields_methods})\}$.

With this result TRInstVar is called.

(1.1.1.1.1.1.1) $\text{TRInstVar}([], \text{factorial}, V_{fields_methods}, i)$

- $C = \text{Int}$
- $(i^{(0)} : \text{int}) \in A_{\text{Int}}$
- $\theta_0 = \text{factorial}$
- $\sigma_{\text{factorial}} = \text{TUnify}_{\leq^*}(\text{factorial}, \text{Int}) = \{[]\}$

The result is then $\{([], \text{int}, V_{fields_methods})\}$.

(1.1.1.1.1) $\text{TRExp}([], V_{fields_methods}, e_0)$

With this result TRExp is called for the second argument $\text{IntLiteral}(1)$. The result is also given as $\{([], \text{int}, V_{fields_methods})\}$.

From this follows that the result of the whole function call is then given as

$\{([], \text{boolean}, V_{fields_methods})\}$.

(1.1.1.1) $\text{TRstmt}([], V_{fields_methods}, \text{IfStmt}(e_0, e_1, e_2))$

With this result of the condition e_0 the function TRstmt is called from TRstmts for the *then*-branch e_1 .

(1.1.1.1.2) $\text{TRstmt}([], V_{fields_methods}, \text{Return}(\text{New}(\text{Int}, (\text{IntLiteral}(1))))$

The result is

$\{([], \text{Int}, V_{fields_methods})\}$.

(1.1.1.1) $\text{TRstmt}([], V_{fields_methods}, \text{IfStmt}(e_0, e_1, e_2))$

With this result of the *then*-branch e_1 the function TRstmt is called again from TRstmts for the *else*-branch e_2 .

(1.1.1.1.3) $\text{TRstmt}([], V_{fields_methods}, e'_2)$

where

$e'_2 = \text{Return}(\text{MethodCall}(\text{this},$
 $\quad \text{mul}(\text{MethodCall}(\text{New}(\text{factorial},$
 $\quad \quad \text{Minus}(\text{InstVar}(\text{this}, i),$
 $\quad \quad \quad \text{IntLiteral}(1))$),
 $\quad \quad \text{fac}())$))

Then TRExp is called.

(1.1.1.1.3.1) $\text{TRExp}([], V_{fields_methods}, e''_2)$

where

$e''_2 = \text{MethodCall}(\text{this},$
 $\quad \text{mul}(\text{MethodCall}(\text{New}(\text{factorial},$
 $\quad \quad \text{Minus}(\text{InstVar}(\text{this}, i),$
 $\quad \quad \quad \text{IntLiteral}(1))$),
 $\quad \quad \text{fac}())$))

First TRtuple is called for the receiver and the argument of the method call mul .

(1.1.1.1.3.1.1) $\text{TRtuple}([], \epsilon, V_{fields_methods}),$
 $(\text{this}, \text{MethodCall}(\text{New}(\text{factorial},$
 $\text{Minus}(\text{InstVar}(\text{this}, i),$
 $\text{IntLiteral}(1))),$
 $\text{fac}()))$

First, for the first expression of the tuple TRmultiply is called. The result is given as $\{([], \text{factorial}, V_{fields_methods})\}$. Then, TRmultiply is called for the second expression. From TRmultiply the function TRExp is called.

(1.1.1.1.3.1.1.1) $\text{TRExp}([], V_{fields_methods}),$
 $\text{MethodCall}(\text{New}(\text{factorial},$
 $\text{Minus}(\text{InstVar}(\text{this}, i),$
 $\text{IntLiteral}(1))),$
 $\text{fac}()))$

Now, TRtuple is called again for the receiver of fac :

$\text{TRtuple}([], \epsilon, V_{fields_methods}), (\text{New}(\text{factorial},$
 $\text{Minus}(\text{InstVar}(\text{this}, i),$
 $\text{IntLiteral}(1))))$.

With the result

$\{([], \text{factorial}, V_{fields_methods})\}$

the function TRMCallApp is called.

(1.1.1.1.3.1.1.1.1) $\text{TRMCallApp}([], \text{factorial}, V_{fields_methods}), \text{fac}()$
 $- \text{case}_1 = \emptyset$ (as fac is no method of an already typed class)
 $- V = V \setminus \{\text{fac}^{(0)}\} \cup \{\text{fac}^{(0)} \rightarrow \alpha_1\}$
 $- \text{unify} = \text{TUnify}_{\leq^*}(\text{factorial}, \text{factorial}) = \{[]\}$
 $- \text{case}_2 = \{([], \alpha_1, V_{fields_methods})\}$

The result is given as

$\{([], \alpha_1, V_{fields_methods})\}$.

(1.1.1.1.3.1.1) $\text{TRtuple}([], \epsilon, V_{fields_methods}),$
 $(\text{this}, \text{MethodCall}(\text{New}(\text{factorial},$
 $\text{Minus}(\text{InstVar}(\text{this}, i),$
 $\text{IntLiteral}(1))),$
 $\text{fac}()))$

The result is given as

$\{([], \text{factorial } \alpha_1, V_{fields_methods})\}$.

(1.1.1.1.3.1) $\text{TRstmt}([], V_{fields_methods}, e_2'')$

With this result the function TRMCallApp is called.

(1.1.1.1.3.1.2) $\text{TRMCallApp}([], \text{factorial } \alpha_1, V_{fields_methods}), e_2'''$

where $e_2''' = \text{mul}(\text{MethodCall}(\text{New}(\text{factorial},$
 $\text{Minus}(\text{InstVar}(\text{this}, i),$
 $\text{IntLiteral}(1))))$

- $\text{mul}^{(1)} : \text{Int} \rightarrow \text{Int} \in A_{\text{Int}}$
- $(\bar{\theta}_{1,0}, \bar{\theta}_{1,1} \rightarrow \bar{\theta}_1) = (\text{Int}, \text{Int} \rightarrow \text{Int})$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}((\text{factorial}, \alpha_1), (\text{Int}, \text{Int})) = \{[\alpha_1 \mapsto \text{Int}], [\alpha_1 \mapsto \text{factorial}]\}$
- $\text{subset}_1 = \{([\alpha_1 \mapsto \text{Int}], \text{Int}, V_{fields_methods} \setminus \{\text{fac} \rightarrow \alpha_1\} \cup \{\text{fac} \rightarrow \text{Int}\}),$
 $([\alpha_1 \mapsto \text{factorial}], \text{factorial}, V_{fields_methods} \setminus \{\text{fac} \rightarrow \alpha_1\} \cup \{\text{fac} \rightarrow \text{factorial}\})\}$
- $\text{case}_1 = \text{ret}_{\text{Int}} = \text{subset}_1$
- $\text{case}_2 = \emptyset$ (as mul is no recursive call)

This means that the result is given as

$\{([\alpha_1 \mapsto \text{Int}], \text{Int}, V_{1.1.1.1.3.1}^1), ([\alpha_1 \mapsto \text{factorial}], \text{Int}, V_{1.1.1.1.3.1}^2)\}$.

where

$V_{1.1.1.1.3.1}^1 = V_{fields_methods} \setminus \{\text{fac} \rightarrow \alpha_1\} \cup \{\text{fac} \rightarrow \text{Int}\}$
 $V_{1.1.1.1.3.1}^2 = V_{fields_methods} \setminus \{\text{fac} \rightarrow \alpha_1\} \cup \{\text{fac} \rightarrow \text{factorial}\}$

(1.1.1.1) $\text{TRstmt}([], V_{fields_methods}, \text{IfStmt}(e_0, e_1, e_2))$

The result of the *then*-branch is given as

$\{(\sigma_1, ty_1, V_1)\} = \{([], \text{Int}, V_{fields_methods})\}$.

The result of the *else*-branch is given as

$\{(\sigma_{1,1}, ty_{1,1}, V_{1,1}), (\sigma_{1,2}, ty_{1,2}, V_{1,2})\} = \{([\alpha_1 \mapsto \text{Int}], \text{Int}, V_{1.1.1.1.3.1}^1),$
 $([\alpha_1 \mapsto \text{factorial}], \text{Int}, V_{1.1.1.1.3.1}^2)\}$.

This means

- $\text{unify}_{1,1}^1 = \{[]\}$
- $\text{unify}_{1,1}^2 = \{[]\}$
- $\text{unify}_{1,2}^1 = \{[]\}$
- $\text{unify}_{1,2}^2 = \{[]\}$

From this follows the result

$\{([], \text{Int}, V_{1.1.1.1.3.1}^1), ([], \text{Int}, V_{1.1.1.1.3.1}^2)\}$.

(1.1.2) $\text{TRStmts}([], V_{fields_methods}, Bl_2)$

This means that the result is also:

$$\{ ([], \text{Int}, V_{1.1.1.1.3.1}^1), ([], \text{Int}, V_{1.1.1.1.3.1}^2) \}.$$

(1.1) $\text{TRstart}(\text{Block}(Bl_1), \text{Block}(Bl_2), (V_1, V_2), V_{fields_methods})$

In TRstart the following is done

- $\theta_1 = \text{RetType}(\text{fac}, V_{1.1.1.1.3.1}^1) = \text{Int}$
- $\theta_2 = \text{RetType}(\text{fac}, V_{1.1.1.1.3.1}^2) = \text{factorial}$
- $\text{unify}_1 = \text{TUnify}_{\leq^*}(\text{Int}, \text{Int}) = \{ [] \}$
- $\text{unify}_2 = \text{TUnify}_{\leq^*}(\text{Int}, \text{factorial}) = \emptyset$

From this follows the result

$$\{ ([], V_{1.1.1.1.3.1}^1) \}.$$

(1) $\text{TRprog}(A, \text{absfactorial})$

In TRprog finally, the intersection type of the different reconstructed type is built:

$$A_{\text{factorial}} = \{ \begin{array}{ll} \langle \text{init} \rangle_{\text{factorial}}^{(1)} & : \text{int} \rightarrow \text{factorial}, \\ \text{fac}^{(0)} & : \rightarrow \text{Int} \end{array} \}.$$

The result is then given as:

$$\{ A_{\text{Int}}, A_{\text{factorial}} \}.$$

Summerized let us consider again the handling of inherited fields and recursive method calls.

The handling of inherited fields is shown in (1.1.1.1.1.1.1). The field `i` (`InstVar(this, i)`) is inherited from the class `Int`. During the type reconstruction all classes are determined, which have an field `i`. In our case only the class `Int` is determined. Then by type unification the receiver and the determined classes are unified. In our case `factorial` (the actual class) and `Int` are unified.

In (1.1.1.1.3.1.1.1.1) we showed the handling of recursive method calls. In our example the method `fac` is called recursive. In the function `TRMCallApp` first all classes are determined, which contains methods with the given name and the given number of arguments. In our case there is no class with a method `fac`. Only in the set of type assumptions $V_{fields_methods}$, which contains the type assumptions of the actual class, there is an assumption for `fac`. Which means that this method call is a recursive call.

Bibliography

- [Age95] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.
- [App02] Andrew W. Appel. *Modern compiler implementation in Java*, chapter 16, pages 354–386. Cambridge University Press, 2002.
- [APS93] O. Agesen, J. Palsberg, and M. Schwartzbach. Type Inference of Self: Analysis of objects with dynamic and multiple inheritance. *ECOOP’93, Seventh European Conference on Object-Oriented Programming*, pages 329–349, 1993.
- [Aus04] Calvin Austin. J2SE 1.5 in a nutshell. <http://java.sun.com/developer/technicalArticles/releases/j2se15>, February 2004.
- [AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.
- [Bar84] Hendrik P. Barendregt. *The lambda calculus : Its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, 1984.
- [BBM94] C. Beierle, S. Bottcher, and G. Meyer. Report of the logic programming language protos-1, 1994.
- [BCK⁺01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding Genericity to the Java Programming Language: Participant Draft Specification. <http://java.sun.com>, 2001.
- [Bei95] Christoph Beierle. Type inferencing for polymorphic order-sorted logic programs. In *International Conference on Logic Programming*, pages 765–779, 1995.
- [BH98] Bauer and Höllerer. *Übersetzung objektorientierter Programmiersprachen*. Springer-Verlag, 1998. (in german).

- [BM94] Christoph Beierle and Gregor Meyer. Run-time type computations in the warren abstract machine. *Logic Programming*, (18):123–148, 1994.
- [BMS80] Rod M. Burstall, Dave B. MacQueen, and Don T. Sannella. Hope—an experimental applicative language. Technical Report CSR-62-80, University of Edinburgh, Dept. of Computer Science, 1980.
- [BOSW98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. *GJ: Extending the Java Programming Language with type parameters*, 1998.
- [BOSW98b] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. *GJ Specification*, 1998.
- [BOSW98c] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.
- [BS01] Franz Baader and Wayne Snyder. Unification Theory. In Robinson and Voronkov [RV01], chapter 8, pages 447–533.
- [Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.
- [CCH⁺89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM Press, 1989.
- [CHC90] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135. ACM Press, 1990.
- [Chi93] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, 1993.
- [CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proceedings ACM Conference on Lisp and Functional Programming*, pages 170–181, June 1992.
- [CKPZ93] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent aggregates language report 2.0. Technical report, University of Illinois at Urbana-Champaign, Department of Computer Science, 1993. <http://www-csag.ucsd.edu/projects/concert/ca.html>.
- [CMP00] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d’applications avec Objective Caml*. O’Reilly, avril 2000. in french, translation <http://caml.inria.fr/oreilly-book>.

- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):470–522, 1985.
- [DEK99] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.
- [DHH81] Ingalls Daniel H. H. Design Principles Behind Smalltalk. *BYTE Magazine*, August 1981. http://users.ipa.net/~dwighth/smalltalk/byte-aug81/design_principles_behind_smalltalk.html.
- [DKTE04] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 15–34, New York, NY, USA, 2004. ACM Press.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.
- [eAB⁺02] Simon Peyton Jones [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 language and libraries, the revised report, December 2002.
- [EST95a] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 169–184, New York, NY, USA, 1995. ACM Press.
- [EST95b] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Type Inference for Recursively Constrained Types and its Application to Object Oriented Programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.
- [FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Proceedings 2nd European Symposium on Programming (ESOP '88)*, pages 94–114, 1988.
- [FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.
- [Fre94] Tim Freeman. *Refinement Types of ML*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1994.
- [FTK⁺05] Robert Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005.

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java language specification*. The Java series. Addison-Wesley, 1996.
- [GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The JavaTM Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.
- [GM89] J. A. Goguen and J. Meseguer. Order-sorted algebras I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Technical report, SRI International, July 1989.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [Hen88] Fritz Henglein. Type inference and semi-unification. In *Proceedings ACM Conference on Lisp and Functional Programming*, New York, July 1988.
- [Hen93] Fritz Henglein. Type inference with Polymorphic Recursion. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 15(2), pages 253–289, April 1993.
- [Her30] J. Herbrand. Recherches sur la Théorie de la Démonstration (thesis). In W. Goldfarb, editor, *Logical Writings*. Cambridge, 1930.
- [Hin69] R. Hindley. The principle type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.* 146, pages 29–60, December 1969.
- [HT92] Patricia M. Hill and Rodney W. Topor. A Semantics for Typed Logic Programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.
- [IPW99] 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.
- [Jen84] R. D. Jenks. A primer: 11 keys to new Scratchpad. In J. Fitch, editor, *Proceedings Symposium on Symbolic and Algebraic Computation (EUROSAM 84)*, volume 147 of *LNCS*, pages 123–147, Cambridge, England, July 1984. Springer-Verlag.
- [Jon93] Mark P. Jones. A system of constructor classes: Overloading and implicate higher-order polymorphism. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 52–61, Copenhagen, Denmark, June 1993.

- [Jon94] Mark P. Jones. Gofer, September 1994.
- [JS92] R. D. Jenks and R. S. Sutor. *AXIOM: The Scientific Computation System*. Springer-Verlag, New Yourk, 1992.
- [JW74] Kathleen Jensen and Niklaus Wirth. *PASCAL. User Manual and Report*. Lecture notes in computer science, Bd. 18(S10). Springer, Berlin, 1974.
- [JWM85] Kathleen Jensen, Niklaus Wirth, and Andrew B. Mickel. *Pascal user manual and report : rev. for the ISO Pascal standard*. Springer, New York, 3rd edition, 1985. revised by Andrew B. Mickel and James F. Miner.
- [Kae88] Stefan Kaes. Parametric overloading in polymorphic programming languages. In H. Ganzinger, editor, *Proceedings 2nd European Symposium on Programming (ESOP '88)*, volume 300 of *Lecture Notes in Computer Science*, pages 131–144, Nancy, France, March 1988. Springer-Verlag.
- [Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proceedings ACM Conference on Lisp and Functional Programming Languages*, pages 193–204, June 1992.
- [KPS94] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall software series. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.
- [KTU90] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. In *Proceedings 22nd Annual ACM Symposium on Theory of Computation (STOC)*, pages 468–476, Baltimore, Maryland, May 1990.
- [KTU93] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the Presence of Polymorphic Recursion. *ACM Transactions on Programming Languages and Systems*, 15(2), April 1993.
- [Lei83] Daniel Leivant. Polymorphic type inference. *Proc. 10th Symposium on Principles of Programming Languages 1982*, 1983.
- [Lit98] Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *OOPSLA '98 Conference Proceedings*, volume 33(10), pages 388–411, 1998.
- [LY98] Oukseh Lee and Kwangkeun Yi. Proofs About A Folklore Let-Polymorphic Type Inference Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1998. to appear.

- [Mey92] B. Meyer. *Eiffel: The language*. Prentice Hall International, 1992.
- [MGS89] J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. *Journal of Symbolic Computation*, 8:383–413, 1989.
- [Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 17:348–378, 1978.
- [Mil97] Robin Milner. *The definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.
- [Mit84] John C. Mitchell. Coercion and type inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185. ACM Press, 1984.
- [Mit91] John C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.
- [MM76] A. Martelli and U. Montanari. Unification in Linear Time and Space, A structured Presentation. Technical Report B76–16, University of Pisa, 1976.
- [MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
- [MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, Mass., 1990.
- [Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Int. Conf. on Programming*, volume LNCS 167, 1984.
- [NS91] T. Nipkow and G. Snelting. Type classes and overloading resolving via order-sorted unification. In Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [Ode02] Martin Odersky. Inferred type instantiation for GJ. Note sent to the types mailing list, January 2002.
- [Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.
- [OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming*, LNCS 615:329–349, July 1992.
- [ORW00] Martin Odersky, Enno Runne, and Philip Wadler. Two Ways to Bake Your Pizza – Translating Parameterised Types into Java. *Proceedings of a Dagstuhl Seminar, Springer Lecture Notes in Computer Science*, 1766:114–132, 2000.

- [OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.
- [OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.
- [OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Conference Record of Conference on Functional Programming Languages and Computer Architecture*, pages 135–146, La Jolla, California, June 1995. ACM Press.
- [OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, 2001.
- [PC94] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340. ACM Press, 1994.
- [Plü99a] Martin Plümicke. OBJ-P *The Polymorphic Extension of OBJ-3*. PhD thesis, University of Tuebingen, WSI-99-4, 1999.
- [Plü99b] Martin Plümicke. Polymorphism in OBJ-P. In *Perspectives of System Informatics, Proceedings*, volume LNCS 1755 of *Lecture Notes of Computer Science*, pages 148–153. Springer-Verlag, 1999.
- [PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, October 1991.
- [PS92] Jens Palsberg and Michael I. Schwartzbach. Safety analysis versus type inference for partial types. *Information Processing Letters*, 43:175–180, 1992.
- [PS94] Jens Palsberg and Michael I. Schwartzbach. *Object-oriented type systems*. John Wiley & Sons, 1994.
- [PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
- [Rey85] J C Reynolds. Three approaches to type structure. In *Proc. of the international joint conference on theory and practice of software development (TAPSOFT) Berlin, March 25-29, 1985 on Mathematical foundations of software development, Vol. 1: Colloquium on trees in algebra and programming (CAAP'85)*, pages 97–138. Springer-Verlag New York, Inc., 1985.

- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, January 1965.
- [RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers B.V., 2001.
- [Sch91] Michael I. Schwartzbach. *Type inference with inequalities*. Springer, New York, 1991.
- [Sie89] J. Siekmann. Unification Theory. *J. Symbolic Computation*, 7:207–274, 1989.
- [Smo88a] Gert Smolka. Logic programming with polymorphically order-sorted types. *Proc. First International Workshop on Algebraic and Logic Programming*, Springer-Verlag, LNCS 343:53–70, 1988. Gaussig, GDR.
- [Smo88b] Gert Smolka. TEL (version 0.9), report and user manual. Technical Report 87-17, FB Informatik, Universität Kaiserslautern, 1988. SEKI-report.
- [Smo89] Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.
- [SNGM89] Gert Smolka, Werner Nutt, Joseph A. Goguen, and José Meseguer. Order-sorted equational computation. In Hassan Ait-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2*, chapter 10, pages 297–367. Academic Press, 1989.
- [SS89] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNCS*. Springer-Verlag, 1989.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1991.
- [TEPH05] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In Philip Wadler, editor, *Proceedings of FOOL 12*, Long Beach, California, USA, January 2005. ACM, School of Informatics, University of Edinburgh.
- [Thi93] Peter Thiemann. An Overview of the SODA system. In *3rd Int. Conf. on Algebraic Methodology and Software Technology*, pages 185–192, 1993.
- [Thi94] Peter Thiemann. *Grundlagen der funktionalen Programmierung*, chapter 11.2. Teubner, 1994.
- [Tur86] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings Functional Programming Languages and Computer Architecture*, Nancy, volume 201 of *Lecture notes in computer science*, pages 1–16. Springer-Verlag, 1986.

- [US91] David Ungar and R. B. Smith. SELF: The power of simplicity. *LISP and Symbolic Computation*, 4(3):187–205, July 1991.
- [VS91] Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ml typability with overloading. In Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [Wal90] U. Waldmann. Unitary unification in Order-sorted Signatures. Technical Report Forschungsbericht 298, Universität Dortmund, 1989 (Revised Version 1990).
- [Wan87] Mitchell Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.
- [WB89] Philip Wadler and Stephan Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Conf. Record of the 16th ACM Annual Symposium on Principles of Programming Languages*, pages 60–76, 1989.
- [Web93] Andreas Weber. *Type Systems for Computer Algebra*. PhD thesis, University of Tübingen, 1993.
- [Wir88] Niklaus Wirth. *Programming in Modula 2*. Texts and monographs in computer science. Springer, Berlin ; Heidelberg, 4. edition, 1988.
- [WS01] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. *ECOOP, Lecture Notes in Computer Science*, 2072:99–117, 2001.