Typeless programming in Java 5.0

Martin Plümicke

University of Cooperative Education
Stuttgart

1. September 2006

Overview

Problem

Extensions of the Java 5.0 type–system

- parametrized types, type variables, type terms, wildcards

  e.g.

```
Vector<? extends AbstractList<? super Integer>>
```

Problem

Extensions of the Java 5.0 type–system

- parametrized types, type variables, type terms, wildcards
  e.g.

```
Vector<? extends AbstractList<? super Integer>>
```

Complex typings

- Often it is not obvious, which are the *best* types for methods and variables
- Sometimes principle types in Java 5.0 are *intersection types*, which are not expressible (contradictive of writing re-usable code)

Problem

Extensions of the Java 5.0 type–system

- parametrized types, type variables, type terms, wildcards
  e.g.

```
Vector<? extends AbstractList<? super Integer>>
```

Complex typings

- Often it is not obvious, which are the *best* types for methods and variables
- Sometimes principle types in Java 5.0 are *intersection types*, which are not expressible (contradictive of writing re-usable code)

$\Longrightarrow$ Developing a type–inferenz–systems, which determines principle types

Example: Multiplication of matrices

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

Alternative Typing

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

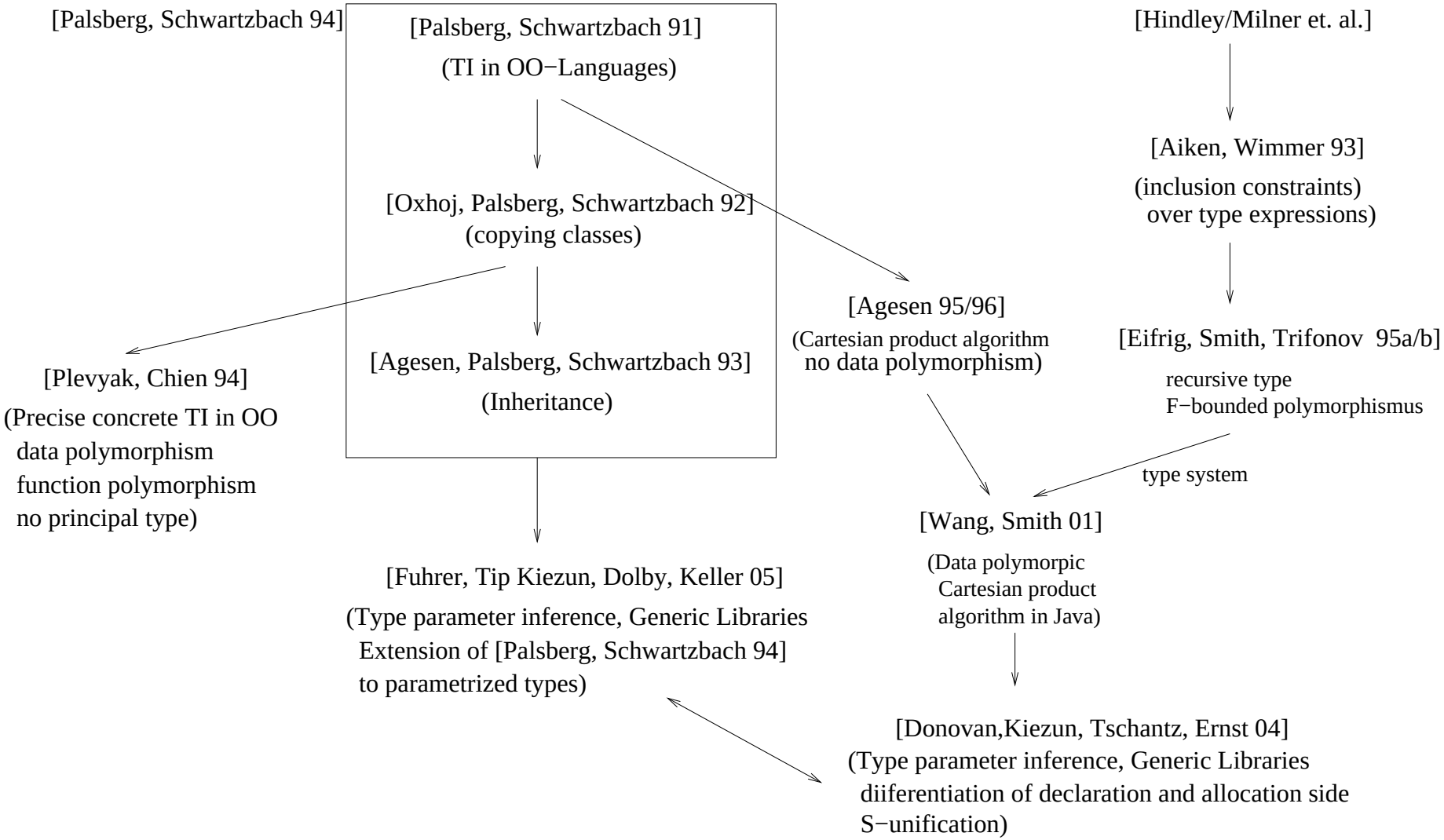
Purpose: Typless

```
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++; }
                    ...
        return ret; }
}
```

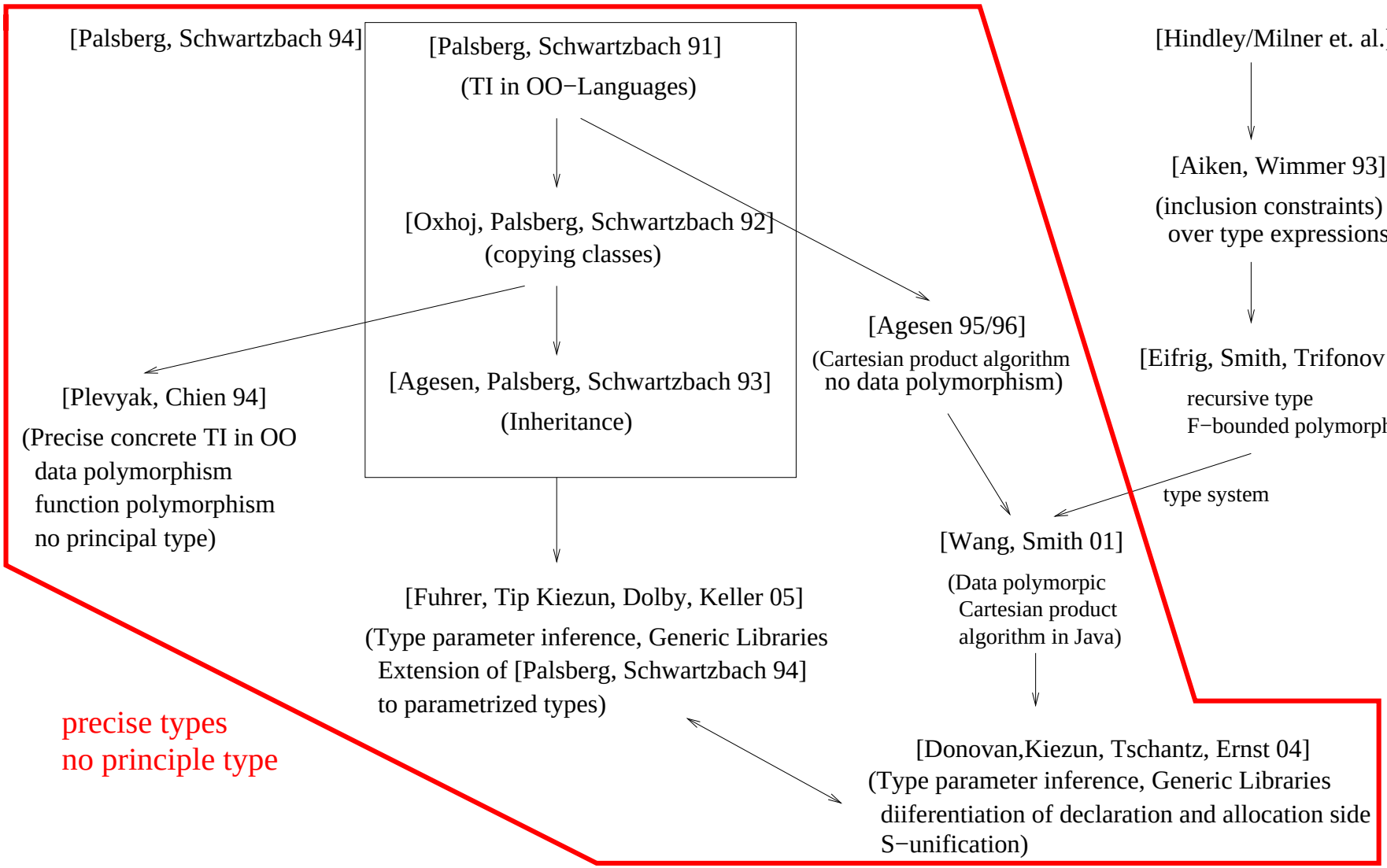
System determines the principle typing(s)

```
mul:  Matrix → Matrix &
      Matrix → Vector<Vector<Integer>> &
      Vector<Vector<Integer>> → Matrix &
      Vector<Vector<Integer>> → Vector<Vector<Integer>>
```

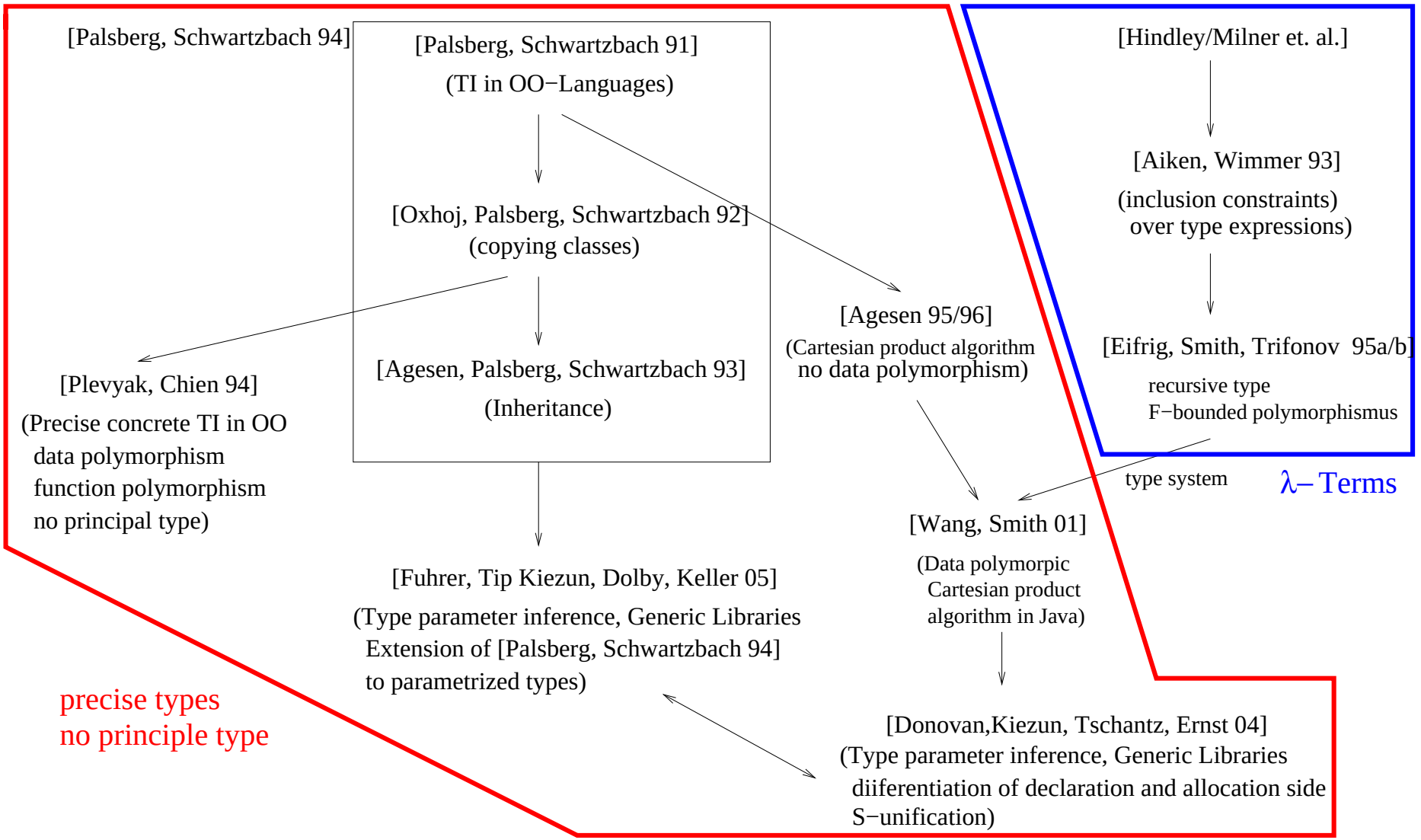
Related work



Related work



Related work



Our approach

[Hindley/Milner et al]

- function type constructor $\longrightarrow$ (no higher–order functions)

Our approach

[Hindley/Milner et al]

- function type constructor $\longrightarrow$ (no higher–order functions)
+ function template $(ty_1 \times \ldots ty_n) \longrightarrow ty_0$ (first–order functions)

Our approach

[Hindley/Milner et al]

- – function type constructor $\longrightarrow$ (no higher–order functions)
- \+ function template $(ty_1 \times \ldots ty_n) \longrightarrow ty_0$ (first–order functions)
- \+ data and function polymorphism (overloading)

The algorithm

Type unification

Subtyping relation for type terms: $\leq^*$

Type Unification problem:

For two type terms θ_1 and θ_2 a substitution σ is demanded such that:

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

Type unifications algorithm[1] I
(base: [Martelli, Montanari 1982])

The type unification algorithm transforms sets of inequations by the following algorithm:

1. Filter all inequations where *exact one* side is a type variable.
2. For each inequation $a \lessdot \theta$ respectively $\overline{\theta} \lessdot a$:
   For each τ with $\tau \leq^* \theta$ respectively $\overline{\theta} \leq^* \tau$
   form a new set of inequation with $a \doteq \tau$
3. Apply the *reduce1*, *reduce2*, *erase*, *swap* und *adapt* to all sets of inequations until nothing is changing.
4. Apply the rule *subst*
5. For all changed sets of inequations *starts again*

[1] [Plümicke 2004, Unif'04, Cork]

Type unifications algorithm II

(reduce1) $$\frac{Eq \cup \{\, C\langle\theta_1, \ldots, \theta_n\rangle \lessdot D\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_{\pi(1)} \doteq \theta'_1, \ldots, \theta_{\pi(n)} \doteq \theta'_n \,\}}$$
where

- $C\langle a_1, \ldots, a_n\rangle \leq^* D\langle a_{\pi(1)}, \ldots, a_{\pi(n)}\rangle$
- $\{\, a_1, \ldots, a_n \,\} \subseteq TV$
- π is a permutation

(reduce2) $$\frac{Eq \cup \{\, C\langle\theta_1, \ldots, \theta_n\rangle \doteq C\langle\theta'_1, \ldots, \theta'_n\rangle \,\}}{Eq \cup \{\, \theta_1 \doteq \theta'_1, \ldots, \theta_n \doteq \theta'_n \,\}}$$

(adapt) $$\frac{Eq \cup \{\, D\langle\theta_1, \ldots, \theta_n\rangle \lessdot D'\langle\theta'_1, \ldots, \theta'_m\rangle \,\}}{Eq \cup \{\, D'\langle\overline{\theta}'_1, \ldots, \overline{\theta}'_m\rangle[a_i \mapsto \theta_i \mid 1 \leqslant i \leqslant n] \lessdot D'\langle\theta'_1, \ldots, \theta'_m\rangle \,\}}$$
where there are $\overline{\theta}'_1, \ldots, \overline{\theta}'_m$ with

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

(erase) $$\frac{Eq \cup \{\, \theta \doteq \theta' \,\}}{Eq} \quad \theta = \theta'$$

(swap) $$\frac{Eq \cup \{\, \theta \doteq a \,\}}{Eq \cup \{\, a \doteq \theta \,\}} \quad a \in TV$$

(subst) $$\frac{Eq \cup \{\, a \doteq \theta \,\}}{Eq[a \mapsto \theta] \cup \{\, a \doteq \theta \,\}}$$
where

- a occurs in Eq but not in θ

Example

Subtyping relation: $\texttt{Matrix<a>} \leq^* \texttt{Vector<Vector<a>>}$

$\texttt{Vector<a>} \leq^* \texttt{AbstractList<a>} \leq^* \texttt{List<a>}$

Application of the algorithm:

$$\{\texttt{Matrix<b>} \lessdot \texttt{Vector<Vector<List<Object>>>}, \texttt{a} \lessdot \texttt{b}\}$$

$$\xRightarrow{(adapt)} \{\{\texttt{Vector<Vector<b>>} \doteq \texttt{Vector<Vector<List<Object>>>}, \texttt{a} \lessdot \texttt{b}\}\},$$

$$\xRightarrow{(reduce2)} \{\{\texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \lessdot \texttt{b}\}\}$$

$$\xRightarrow{(subst)} \{\{\texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \lessdot \texttt{List<Object>}\}\}$$

$$\xRightarrow{(new\ sets)} \{\{\texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{List<Object>}\}, \{\texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{AbstractList<Object>}\}, \{\texttt{b} \doteq \texttt{List<Object>}, \texttt{a} \doteq \texttt{Vector<Object>}\}\}$$

Type-inference-algorithm

Type assumptions: For each absent type in the program a type-placeholder (fresh type variable) is assumed.

Run over the abstract syntax tree: During the run over abstract syntax tree of the coresponding Java class the types are calculated gradually by type unification.

Multiplying the assumptions: If the result of a type unification contains more than one result or if there is data polymorphism, the set of type assumptions is multiplied.

Erase type assumptions: If the type unification fails, the corresponding set of type assumptions is erased.

New method type parameters: At the end remained type-placeholders are replaced by new introduced method type parameters.

Intersection types: At the end each remained set of type assumptions forms one element of the result's intersection type.

Example: Multiplication of matrices: Type assumptions

```
class Matrix extends Vector<Vector<Integer>> {
   {α} mul({β} m) {
      {γ} ret = new Matrix();
      {ϵ} i = 0;
      while(i <size()) {
         {ι} v1 = this.elementAt(i);
         {κ} v2 = new Vector<Integer>();
         {μ} j = 0;
         while(j < v1.size()) {
            {ξ} erg = 0;
            {ρ} k = 0;
            while(k < v1.size()) {
               erg = erg + v1.elementAt(k)
                  m.elementAt(k)).elementAt(j));
               k++; }
               ...
      {ψ} return ret; }}
```

Multiplying the assumptions

```
class Matrix extends Vector<Vector<Integer>> {
   {α,α} mul({β,β} m) {
      {Matrix,Vector<Vector<Integer>>}ret = new Matrix();
      {ϵ,ϵ} i = 0;
      while(i <size()) {
         {ι,ι} v1 = this.elementAt(i);
         {κ,κ} v2 = new Vector<Integer>();
         {μ,μ} j = 0;
         while(j < v1.size()) {
            {ξ,ξ} erg = 0;
            {ρ,ρ} k = 0;
            while(k < v1.size()) {
               erg = erg + v1.elementAt(k)
                 * m.elementAt(k)).elementAt(j));
               k++; }
            v2.addElement(new Integer(erg));
               ...
      {ψ,ψ} return ret; }}
```

End configuration

```
{ Matrix Vector<Vector<Integer>>
  Matrix' Vector<Vector<Integer>> }

    mul({ Matrix                          Matrix
          Vector<Vector<Integer>>' Vector<Vector<Integer>> } m) {

        { Matrix Vector<Vector<Integer>>
          Matrix' Vector<Vector<Integer>> } ret = new Matrix();

        ⋮

        { Matrix Vector<Vector<Integer>>
          Matrix' Vector<Vector<Integer>> } return ret; }}
```

Result:

```
mul:  Matrix → Matrix &
      Matrix → Vector<Vector<Integer>> &
      Vector<Vector<Integer>> → Matrix &
      Vector<Vector<Integer>> → Vector<Vector<Integer>>
```

Principle type

Definition

An intersection type of a method `m` in a class `C`

$$\begin{aligned} m: \quad & (\theta_{1,1} \times \ldots \times \theta_{1,n} \rightarrow \theta_1) \\ & \& \ldots \& \\ & (\theta_{m,1} \times \ldots \times \theta_{m,n}, \rightarrow \theta_m) \end{aligned}$$

is called *principle* if for any correct type annotated method declaration

$$rty\ \texttt{m(}ty1\ \texttt{a1}\ \texttt{,}\ \ldots\ \texttt{,}\ tyn\ \texttt{an)}\ \{\ \ldots\ \ \}$$

there is an element $(\theta_{i,1} \times \ldots \times \theta_{i,n}, \rightarrow \theta_i)$ of the intersection type and there is a substitution σ, such that

$$\sigma(\,\theta_i\,) = rty\,, \sigma(\,\theta_{i,1}\,) = ty1\,, \ldots\,, \sigma(\,\theta_{i,n}\,) = tyn$$

Principle type property

Theorem

If we consider only simple types with unbounded type variables and without wildcards, the type inference algorithm calculates a principle type.

Conclusion and future work

Conclusion

- Type-inference-algorithm for Java 5.0
- Type unification
- Principle type property for Java 5.0 types without bounded type variables and without wildcards

Conclusion and future work

Conclusion

- Type-inference-algorithm for Java 5.0
- Type unification
- Principle type property for Java 5.0 types without bounded type variables and without wildcards

Future work

- Extension to wildcards (type unification algorithm is developed, but not yet implemented)
- Handling of interesection types (adaption of byte-code generation)