

Typeless programming in Java 5.0

Martin Plümicke
University of Cooperative Education Stuttgart
Department of Information Technology
Florianstraße 15, D-72160 Horb
m.pluemicke@ba-horb.de

Jörg Bäuerle
AWM Media
International IT Department
P.O. Box 4006, Worthing BN13 1AP, UK
Joerg.Baeuerle@gmx.net

ABSTRACT

With the introduction of Java 5.0 [9] the type system has been extended by parameterized types, type variables, type terms, and wildcards. As a result very complex types can arise. The term

`Vector<Vector<AbstractList<Integer>>>`

is for example a correct type in Java 5.0.

Considering all that, it is often rather difficult for a programmer to recognize whether such a complex type is the correct one for a given method or not. Furthermore there are methods whose principle types would be intersection types. But intersection types are not implemented in Java 5.0. This means that Java 5.0 methods often don't have the principle type which is contradictive to the OOP-Principle of writing re-usable code.

This has caused us to develop a Java 5.0 type inference system which assists the programmer by calculating types automatically. This type inference system allows us, to declare method parameters and local variables without type annotations. The type inference algorithm calculates the appropriate and principle types.

We implement the algorithm in Java using the observer design pattern.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.2 [Software Engineering]: Design Tools and Techniques—*modules and interfaces*; D.3.3 [Programming Languages]: Language Constructs and Features—*data types and structures*

General Terms

Algorithms, Theory

Keywords

Code generation, language design, program design and im-

```
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++; }  
                v2.addElement(new Integer(erg));  
                j++; }  
            ret.addElement(v2);  
            i++; }  
        return ret; }  
}
```

Figure 1: The class Matrix

plementation, type inference, type system

1. INTRODUCTION

In this paper we present a type inference algorithm for a core language of Java 5.0. Type inference means that we can implement Java 5.0 programs without type annotations of method parameters and return types and of local variables. The type inference system calculates them automatically.

Let us consider an example. The class `Matrix` (fig. 1) extends `Vector<Vector<Integer>>`. `Matrix` has the method `mul`, which implements the multiplication of matrices. The parameters, the return type, and the local variables are explicitly typed (underlined in fig. 1). If we consider the type annotations more accurately, we will observe that there is more than one possibility to give correct type annotations. The return type, the type annotation of `m`, and the type annotation of `ret` are not unambiguous. The type `Vector<Vector<Integer>>` would also be a correct type annotation. This means, considering the type of the method `mul`, that it has beside the type `Matrix` \rightarrow `Matrix`, the type `Vector<Vector<Integer>>` \rightarrow `Vector<Vector<Integer>>`, and all mixtures. The fact that a method has more than one type is called intersection type. In Java 5.0 no annotations of intersection types are allowed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPPJ 2006 August 30 - September 1, 2006, Mannheim, Germany.
Copyright 2006 ACM ...\$5.00.

The idea of Java 5.0 type inference is to omit these type annotations. The system automatically calculates the principle intersection type of the methods.

The type inference discipline arose in functional programming languages (e.g. Haskell [10] or SML [14]). A basic paper for type inference including an algorithm and the definition of principle types is given by [3]. A method's principle type is a type, where all correct types of the method are derivable from. In section 4.3 we give a detailed definition.

Many papers on type inference have been published. Some papers consider type inference in object-oriented languages. In [15, 11] type inference systems for a language oriented at Smalltalk [8] are given. In this approach the idea is to collect type constraints, build a uniform set, and solve them by a fixed point derivation. In [5] similar as in [1] types are defined as a pair of a conventional type and a set of type constraints. Finally, the constraints are satisfied, if possible. These type systems differ from the Java 5.0 type system.

In [4] and [6] a refactoring is presented that replaces raw references to generic library classes with parametrized references. For this the type parameters of the raw type annotations must be inferred. The main difference to our approach is, that in our approach no type annotation, must be given and the algorithm infers the whole type.

The type system of polymorphically order-sorted types, which is considered for a logical language [18] and for a functional object-oriented language OBJ-P [16], is very similar to the Java 5.0 type system. Therefore our approach is oriented at polymorphically order-sorted types.

The paper is organized as follows. In the second section we formally describe the Java 5.0 type system. In the third section, we give a unification algorithm for a subset of the Java 5.0 types. The unification algorithm is the base of the type inference algorithm. In the fourth section we present the type inference algorithm itself, illustrate the algorithm by an example, and consider the properties of the algorithm. Finally in the sixth section we present the implementation. We have done the implementation in Java using the observer design pattern. We close the paper with a conclusion and an outlook.

2. JAVA 5.0 TYPE SYSTEM

As a base for the type inference algorithm we have to give a formal definition of the Java 5.0 type system. First we introduce the simple types. Fields, method parameters, return types of methods, and local variables are annotated by simple types. Then, from the *extends* relation we derive the subtyping relation. Furthermore, we define a relation *finite closure*, which is necessary for the unification algorithm (section 3). Finally, we introduce function types, which represent the types of methods.

The simple types are built as a set of terms over a finite rank alphabet of the class/interface names, a set of unbounded respectively bounded type variables, and unbounded respectively bounded wildcards.

Definition 1. Let JC be a set of declared Java 5.0 classes and $\Theta = (\Theta^{(n)})_{n \in \mathbb{N}}$ the finite rank alphabet of class/interface names indexed by its respective number of parameters. Let TV be a set of unbounded type variables. Furthermore, let $T_\Theta(TV)$ be the set of (type) terms over Θ and TV .

Then, the set of *simple types* $S\text{Type}_\Theta(BTV)$ is the smallest set satisfying the following conditions:

- $T_\Theta(TV) \subseteq S\text{Type}_\Theta(BTV)$
- For each intersection of simple types $ty = ty_1 \& \dots \& ty_n$

$$BTV^{(ty)} \subseteq S\text{Type}_\Theta(BTV)$$

where $BTV^{(ty)}$ contains all type variables, bounded by ty . A bounded type variable is denoted by $a|_{ty}$.

- For $\theta_i \in S\text{Type}_\Theta(BTV)$
 - $\cup \{?\}$
 - $\cup \{? \text{ extends } \tau \mid \tau \in S\text{Type}_\Theta(BTV)\}$
 - $\cup \{? \text{ super } \tau \mid \tau \in S\text{Type}_\Theta(BTV)\}$
- and $C \in \Theta^{(n)} : C < \theta_1, \dots, \theta_n > \in S\text{Type}_\Theta(BTV)$.

The inheritance hierarchy consists of two different relations: The “*extends* relation” (in sign $<$) 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 2. Let JC be a set of declared Java 5.0 classes, $S\text{Type}_\Theta(BTV)$ the corresponding set of simple types, and \leq the corresponding extends relation. The *subtyping relation* \leq^* is given as the smallest ordering satisfying the following conditions:

- if $(\theta, \theta') \in S\text{Type}_\Theta(BTV) \times S\text{Type}_\Theta(BTV)$ is an element of the reflexive and transitive closure of \leq 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:
 - $\sigma_1(a) = \sigma_2(a)$
 - $\sigma_1(a) = \theta$ and $\sigma_2(a) = ?$
 - $\sigma_1(a) = \theta$ and $\sigma_2(a) = ? \text{ extends } \theta'$ and $\theta \leq^* \theta'$
 - $\sigma_1(a) = ? \text{ super } \theta$ and $\sigma_2(a) = \theta'$ and $\theta \leq^* \theta'$
- $a \leq^* \theta'$ for $a \in BTV^{(\theta_1 \& \dots \& \theta_n)}$ where $\exists \theta_i : \theta_i \leq^* \theta'$.

It is surprising that the 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 (cp. [9]).

Furthermore, we declare an ordering on the set of type terms which we call the finite closure of the *extends* relation. This ordering is necessary for the type unification algorithm (section 3).

Definition 3. The *finite closure* $\mathbf{FC}(\leq)$ is the reflexive and transitive closure of pairs in the *extends* relation $C(a_1 \dots a_n) \leq D(\theta_1, \dots, \theta_m)$, where the a_i are type variables and the θ_i are real type terms.

Now we give an example to illustrate the abstract definitions.

Example 1. Let the following Java 5.0 program be given.

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

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

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

$\text{(reduce1)} \quad \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \leq D \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_{\pi(1)} \doteq \theta'_1, \dots, \theta_{\pi(n)} \doteq \theta'_n \}}$ <p>where</p> <ul style="list-style-type: none"> • $C \langle a_1, \dots, a_n \rangle \leq^* D \langle a_{\pi(1)}, \dots, a_{\pi(n)} \rangle$ • $\{ a_1, \dots, a_n \} \subseteq TV$ • π is a permutation 	$\text{(reduce2)} \quad \frac{Eq \cup \{ C \langle \theta_1, \dots, \theta_n \rangle \doteq C \langle \theta'_1, \dots, \theta'_n \rangle \}}{Eq \cup \{ \theta_1 \doteq \theta'_1, \dots, \theta_n \doteq \theta'_n \}}$ $\text{(erase)} \quad \frac{Eq \cup \{ \theta \doteq \theta' \}}{Eq} \quad \theta = \theta'$ $\text{(swap)} \quad \frac{Eq \cup \{ \theta \doteq a \}}{Eq \cup \{ a \doteq \theta \}} \quad \theta \notin TV, a \in TV$
$\text{(adapt)} \quad \frac{Eq \cup \{ D \langle \theta_1, \dots, \theta_n \rangle \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}{Eq \cup \{ D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle [a_i \mapsto \theta_i \mid 1 \leq i \leq n] \leq D' \langle \theta'_1, \dots, \theta'_m \rangle \}}$ <p>where there are $\bar{\theta}'_1, \dots, \bar{\theta}'_m$ with</p> <ul style="list-style-type: none"> • $(D \langle a_1, \dots, a_n \rangle \leq^* D' \langle \bar{\theta}'_1, \dots, \bar{\theta}'_m \rangle) \in \mathbf{FC}(\leq)$ 	$\text{(subst)} \quad \frac{Eq \cup \{ a \doteq \theta \}}{Eq[a \mapsto \theta] \cup \{ a \doteq \theta \}}$ <p>where</p> <ul style="list-style-type: none"> • a occurs in Eq but not in θ

Figure 2: Java 5.0 type unification

The following type term pairs are elements of the subtyping relation \leq^* :

$$\begin{aligned} \text{Vector} \langle \text{Vector} \langle a \rangle \rangle &\leq^* \text{Vector} \langle \text{Vector} \langle a \rangle \rangle, \\ \text{Vector} \langle \text{Vector} \langle a \rangle \rangle &\leq^* \text{AbstractList} \langle \text{Vector} \langle a \rangle \rangle, \\ \text{Matrix} \langle a \rangle &\leq^* \text{Vector} \langle \text{Vector} \langle a \rangle \rangle, \\ \text{Matrix} \langle a \rangle &\leq^* \text{AbstractList} \langle \text{Vector} \langle a \rangle \rangle. \end{aligned}$$

But $\text{Vector} \langle \text{Vector} \langle a \rangle \rangle \not\leq^* \text{Vector} \langle \text{AbstractList} \langle a \rangle \rangle$ which follows from the soundness of the Java 5.0 type system.

The finite closure $\mathbf{FC}(\leq)$ is given as the reflexive and transitive closure of $\{ \text{Matrix} \langle a \rangle \leq^* \text{Vector} \langle \text{Vector} \langle a \rangle \rangle \leq^* \text{AbstractList} \langle \text{Vector} \langle a \rangle \rangle \leq^* \text{List} \langle \text{Vector} \langle a \rangle \rangle \}$.

We complete the Java 5.0 type system with the following definition.

Definition 4. Let $\text{SType}_\Theta(BTV)$ be a set of simple types of given Java 5.0 classes. The respective set of *Java 5.0 types* $\text{Type}(\text{SType}_\Theta(BTV))$ is the smallest set with the following properties:

1. For the considered **base types** holds
 $\{ \text{Integer}, \text{Boolean}, \text{Char} \} \subseteq \text{Type}(\text{SType}_\Theta(BTV)).$
2. $\text{SType}_\Theta(BTV) \subseteq \text{Type}(\text{SType}_\Theta(BTV))$
(simple type).
3. If for $0 \leq i \leq n$: $\theta_i \in (\text{SType}_\Theta(BTV) \cup \text{basetype})$ then $\theta_1 \times \dots \times \theta_n \rightarrow \theta_0 \in \text{Type}(\text{SType}_\Theta(BTV))$ **(function type)**.
4. If $ty_1, ty_2 \in \text{Type}(\text{SType}_\Theta(BTV))$, then $ty_1 \& ty_2 \in \text{Type}(\text{SType}_\Theta(BTV))$ **(intersection type)**.

The *base types* and the *simple types* describe types of fields, types of methods' parameters, return types of methods, and types of local variables. Finally the types of the methods are given as intersections of *function types*. The intersections are necessary to describe the principle type of a method. If a method has an intersection type, this means that more than one type is inferable for the code.

We do not consider *raw types* as they are only necessary to use older Java programs (Version ≤ 1.4). The behavior of raw types can be simulated by using the corresponding parameterized types, where all arguments are instantiated by **Object**.

3. TYPE UNIFICATION

The basis of the type inference algorithm is the type unification. The *type unification problem* is given as: For two type terms θ_1, θ_2 a substitution is demanded, such that

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

σ is called a unifier of θ_1 and θ_2 . In the following we denote $\theta \leq \theta'$ for two type terms, which should be type unified.

Our type unification algorithm is based on the unification algorithm of Martelli and Montanari [13]. 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 type terms smaller than θ and greater than θ respectively are correct substitutions for a . This means that there are multiple unifiers.

Now, we give the type unification algorithm. We restrict the type terms to terms without bounded type variables and without wildcards. We denoted this subset of $\text{SType}_\Theta(BTV)$ in definition 1 by $T_\Theta(TV)$.

The algorithm itself is given in seven steps:

1. For each pair $a \leq \theta$ a set of pairs is built, which contains for all subtypes $\bar{\theta}$ of θ the pair $a \doteq \bar{\theta}$.
2. For each pair $\theta \leq a$ a set of pairs is built, which contains for all supertypes θ' of θ the pair $a \doteq \theta'$.
3. The cartesian product of the sets from step 1 and 2 is built.
4. Repeated application of the rules *reduce1*, *reduce2*, *erase*, *swap*, and *adapt* (fig. 2) to each set of type term pairs.
5. Application of the rule *subst* (fig. 2) to each set of type term pairs.
6. For all changed sets of type terms start again with step 1.
7. Summarize all results

For more details about the Java 5.0 type unification see in [17].

Example 2. Let the subtyping relation and the finite closure be given as in example 1.

<i>Source</i>	$:=$	$(class \mid interface)^*$
<i>class</i>	$:=$	$Class(stype, [extends(stype),] [implements(stype+),] IVarDecl^*, Method^*)$
<i>interface</i>	$:=$	$interface(stype, [extends(stype),] MHeader^*)$
<i>IVarDecl</i>	$:=$	$InstVarDecl(stype, var)$
<i>MHeader</i>	$:=$	$MethodHeader(mname, stype, (var, stype)^*)$
<i>Method</i>	$:=$	$Method(mname, [stype], (var[stype])^*, block)$
<i>block</i>	$:=$	$Block(stmt^*)$
<i>stmt</i>	$:=$	$block \mid Return(expr) \mid while(expr, block) \mid LocalVarDecl(var[stype]) \mid If(expr, block[, block])$
		$\mid stmtexpr$
<i>stmtexpr</i>	$:=$	$Assign(var, expr) \mid New(stype, expr^*) \mid NewArray(stype, expr) \mid MethodCall([expr,]f, expr^*)$
<i>expr</i>	$:=$	$stmtexpr \mid this \mid super \mid LocalOrFieldVar(var) \mid InstVar(expr, var) \mid ArrayAcc(expr, expr)$
		$\mid Add(expr, expr) \mid Minus(expr, expr) \mid Mul(expr, expr) \mid Div(expr, expr) \mid Mod(expr, expr)$
		$\mid Not(expr) \mid And(expr, expr) \mid Or(expr, expr)$

Figure 3: The Java 5.0 core language

We apply the unification algorithm to

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

In the first three steps nothing happens.

In step 4 we get by the *adapt* rule

$\{ \{ Vector<Vector> \\ \doteq Vector<Vector<List<Object>>>, \\ a < b \} \},$

as $(Matrix<a> \leq^* Vector<Vector<a>>) \in FC(\leq)$. Then the *reduce2* rule leads to: $\{ \{ b \doteq List<Object>, a < b \} \}$. In step 5 the *subst* rule leads to

$\{ \{ b \doteq List<Object>, \\ a < List<Object> \} \}$

With the again application of the first three steps we get finally:

$\{ \{ b \doteq List<Object>, a \doteq List<Object> \}, \\ \{ b \doteq List<Object>, a \doteq AbstractList<Object> \}, \\ \{ b \doteq List<Object>, a \doteq Vector<Object> \} \}$

This means that this type unification has three unifiers as its results.

4. TYPE INFERENCE

The language for our type inference algorithm is given in figure 3. It is an abstract representation of a core of Java 5.0. The input of the type inference algorithm is a set of abstract syntax trees representing Java 5.0 classes, where the parameters, return types, and local variables of the methods are not necessarily type annotated (underlined in figure 3). The type inference algorithm infers the absent type annotations. The result of the algorithm contains for each method an intersection of function types and the corresponding typings for the local variables. The intersection of function types of the methods describes the different possible typings of its parameters and its return types.

4.1 The Algorithm

The basic idea of the algorithm is that each expression, each statement and each block is typed by simple types and that each method is typed by function types. These types are determined step by step during the run of the algorithm.

Type assumptions: First, we assume for each expression, for each statement, and for each block a type variable as a type-placeholder. The types of the methods are assumed as function types, which consists also of type-placeholders for each argument and the return type.

Run over the abstract syntax tree: During a run over the abstract syntax tree, the types of each expression, each statement, and each block are determined. This is done step by step. At each position of the abstract syntax tree there are type assumptions of the expressions, the statements, and the blocks, respectively, and there are conditions for these types given by the Java 5.0 type system. The type assumptions are unified by type unification as the respective type-conditions define.

For example if we determine the type of $Assign(a, expr)$ ($a = expr$), we have type assumptions ty_a and ty_{expr} for a respective $expr$. The type-condition for $Assign$ defines, that it holds $ty_{expr} \leq^* ty_a$. This means that the type unification algorithm is applied to $\{ ty_{expr} < ty_a \}$. After that the resulting unifiers are applied to the respective type assumptions.

There are rules for each Java 5.0 construct, which define the type-conditions for the corresponding types.

Multiplying the assumptions: In two cases the set of type assumptions is multiplied:

- If the result of the type unification contains more than one unifier, for each unifier a new set of type assumptions is generated.
- If during a method call there are different receivers, which can invoke the method, for each receiver a new set of type assumptions is generated.

In both cases the algorithm is continued on both sets of type assumptions.

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

New method type parameters: If at the end, there are type-placeholders contained in type assumptions, these type-placeholders are replaced by new introduced method type parameters.

Intersection types: If at the end, there is more than one set of type assumptions for a method, this method has an intersection type, which is then generated.

4.2 Type Inference Example

We consider again the *matrices* example from the introduction. We apply the algorithm to the corresponding abstract syntax tree of the class `Matrix` (fig. 1), where the underlined type annotations are erased. In the first step **type assumptions** all expressions, statements, and the block are typed by type-placeholders. In the following we consider some steps of the **run over the abstract syntax tree**.

First, the `New`-statement `New(Matrix,())` (in concrete syntax: `new Matrix();`) gets the type `Matrix` as its type assumption.

Then, the statement `Assign(ret, New(Matrix,()))` (`ret = new Matrix();`) should be typed. For this the type assumption of `ret` is also necessary. The type assumption of `ret` is the type-placeholder β . The type-condition for `Assign`-statements, leads to the condition `Matrix` \leq β . The type unification gives two unifiers: $\{\beta \mapsto \text{Matrix}\}$ and $\{\beta \mapsto \text{Vector}\langle \text{Vector}\langle \text{Integer} \rangle \rangle\}$. This means that the algorithm's step, **multiplying the assumptions**, is applied and we get two sets of type assumptions. In the first one the type of `ret` is assumed as `Matrix` and in the second one it is assumed as `Vector` \langle `Vector` \langle `Integer` $\rangle\rangle$.

Next, we consider the type calculation of the statement `MethodCall(MethodCall(m, elementAt, k), elementAt, j)` (`m.elementAt(k).elementAt(j)`) in the innermost while-loop. First, the type of `MethodCall(m, elementAt, k)` is determined. The type assumption of `m` is the type-placeholder α . Now all types are considered, which can invoke a method `elementAt`. In our context it is `Vector` \langle `T` \rangle with `elementAt` : `Vector` \langle `T` \rangle \rightarrow `T`. The type-condition of the `methodcall`-rule defines that for a new type-placeholder δ , it holds $\alpha \leq \text{Vector}\langle\delta\rangle$. There are two unifiers: $\{\alpha \mapsto \text{Vector}\langle\delta\rangle\}$ and $\{\alpha \mapsto \text{Matrix}, \delta \mapsto \text{Vector}\langle \text{Integer} \rangle\}$. This means that we get, by the algorithm's step **multiplying the assumptions**, two different type assumptions δ and `Vector` \langle `Integer` \rangle for `MethodCall(m, elementAt, k)`. This expression is simultaneously the receiver of the second method call. This means that on the one hand for δ all types are considered, which can invoke a method `elementAt`. This is again `Vector` \langle `T` \rangle . On the other hand it is determined whether `Vector` \langle `Integer` \rangle can invoke `elementAt`. This is also possible. This means following the type-condition of the `methodcall`-rule, that for a new type-placeholder ϵ it must hold $\delta \leq \text{Vector}\langle\epsilon\rangle$ and for a further type-placeholder ϵ' it must hold `Vector` \langle `Integer` $\rangle \leq \text{Vector}\langle\epsilon'\rangle$. The first unification again gives the unifiers $\{\delta \mapsto \text{Vector}\langle\epsilon\rangle\}$ and $\{\delta \mapsto \text{Matrix}, \epsilon \mapsto \text{Vector}\langle \text{Integer} \rangle\}$. The second unification gives $\{\epsilon' \mapsto \text{Integer}\}$. This means, that we get for `MethodCall(MethodCall(m, elementAt, k), elementAt, j)` : $(*)$ three type assumptions `Integer`, ϵ , and `Vector` \langle `Integer` \rangle and for the parameter `m` we get the type assumptions `Vector` \langle `Vector` \langle ϵ $\rangle\rangle$, `Vector` \langle `Matrix` \rangle , and `Matrix`.

Then the type for `Add(..., (*))` is determined. The type-condition for the addition demands that its arguments are subtypes of `Integer`. The means, that it must holds `Integer` \leq `Integer`, $\epsilon \leq$ `Integer`, and `Vector` \langle `Integer` $\rangle \leq$ `Integer`. It is obvious, that the last unification fails. This means that the algorithm's step **erase type assumptions** is applied and the corresponding set of type assumptions is

erased. From this, it follows, that for `m` there remains two adapted type assumptions `Vector` \langle `Vector` \langle `Integer` $\rangle\rangle$ and `Matrix`.

During the rest of the running nothing happens to the type assumptions of the parameter `m`.

The statement `Return(ret)` (`return ret;`) determines the return type of this presently considered method `mul`. As the type assumptions of the local variable `ret` are `Matrix` and `Vector` \langle `Vector` \langle `Integer` $\rangle\rangle$, these two are the type assumptions for the return type.

In the last step of the algorithm, **intersection types**, this leads to the following inferred type for `mul`:

```
mul : Vector<Vector<Integer>>  $\rightarrow$  Matrix
    & Matrix  $\rightarrow$  Matrix
    & Vector<Vector<Integer>>
                                    $\mapsto$  Vector<Vector<Integer>>
    & Matrix  $\mapsto$  Vector<Vector<Integer>>
```

This is the result which we expected in the first section.

4.3 Principle Type Property

First, we will give a definition of a principle Java 5.0 type. The definition is a generalization of the corresponding definition in functional programming languages [3].

Definition 5. An intersection type of a method `m` in a class `C`

$$m : (\theta_{1,1} \times \dots \times \theta_{1,n} \rightarrow \theta_1) \\ \& \dots \& \\ (\theta_{m,1} \times \dots \times \theta_{m,n} \rightarrow \theta_m)$$

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

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

there is an element $(\theta_{1,1} \times \dots \times \theta_{1,n} \rightarrow \theta_1)$ of the intersection type and there is a substitution σ , such that

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

THEOREM 1. *If we consider only simple types with unbounded type variables and without wildcards, the type inference algorithm calculates a principle type.*

4.4 Resolving Intersection Types

In conventional Java no intersection types for methods are allowed. This means that the compiler cannot translate them. Therefore, we need an approach to deal with intersection types after they are inferred. There are three possibilities.

The first one is to present the user with all inferred types and the user has to select one of them. Subsequently code is generated for that type. At the moment we have implemented this approach.

Another approach would be to generate code for each element of the intersection type. This approach would have the advantage, that later on all inferred types for the method would be usable. The disadvantage is, that the same code appears several times in the byte-code file.

The third idea is to generate the code for each method only once. However, for each type of the intersection type an entry in the constant-pool is built. This means that the same executable code is referenced by different entries in the constant-pool. For this approach we have to do further investigations.

5. IMPLEMENTATION

In order to present a proof of concept, we have integrated the type inference system in a **Java** compiler. The compiler itself has been implemented in **Java** by using the tools JLex [2] and jay [12].

In its analysis phase the compiler parses a **Java** program and creates an *Abstract Syntax Tree* (subsequently called *AST*). This *AST* forms, together with some other basic data structures, the input data for the *Type Inference Algorithm* (subsequently called *TIA*) described in section 4. The *TIA* calculates the missing type annotations and performs a general type checking. In doing so, it replaces the common semantic check.

5.1 Basic Data Structures

5.1.1 TypePlaceholder

All the types of the **Java** program to be compiled are represented in the *AST* by instances of subclasses of the abstract class *Type*. In order to allow programmers to omit type annotations for method declarations and local variable declarations, we have to extend the type hierarchy by introducing another subclass of *Type*.

This subclass is an auxiliary data structure for the *TIA*. Its instances act as placeholders for the individual declaration types. The class is therefore called *TypePlaceholder*.

5.1.2 TypeAssumption

The implementation of the type assumptions described in section 4 is realized by the abstract class *TypeAssumption*. This class is, besides the *AST*, the main data structure for the *TIA*. It basically maps an identifier onto an assumed type by storing a *String* instance and a *Type* instance.

At the beginning of the *TIA* an initial set of *TypeAssumption* instances is created for all field declarations (field variables and methods). During the processing of the *TIA* when more and more knowledge about the types is gained, this set is extended by adding new *TypeAssumption* instances or by modifying old ones.

5.1.3 Substitution

While *TypeAssumption* is the implementation for mapping an identifier onto a type, the class *Substitution* maps a type placeholder onto a calculated type. A *Substitution* instance stores a *TypePlaceholder* reference and the corresponding *Type* instance which the placeholder will be replaced with. Normally for each unifier provided by the unification algorithm (see section 3) a *Substitution* instance is created.

Through the method *Substitution.execute()* the type replacement for this particular placeholder in the *AST* can be invoked (see section 5.2).

5.2 Substitution Based Approach

The *TIA* theoretically described in section 4 follows an approach which is mainly based on type assumptions. The algorithm cyclically collects type information and unification results in order to extend and specify existing sets of type assumptions.

Type substitutions however play a minor role in this approach and are only used as an auxiliary means. The unifiers provided by the unification algorithm are usually discarded after their type substitution has been applied on the sets of

type assumptions.

The *TIA*'s output data structure consists of multiple sets of type assumptions which represent a theoretical image of the **Java** program's type configuration. Type unifiers or type substitutions are not part of the output data structure.

This assumption based approach is very difficult to implement as such a set of type assumptions as a whole cannot be applied to the *AST*. The type information must be broken down into small, executable instructions. These instructions are identical to the type substitutions, though. For the implementation it is crucial to add all type substitutions as *Substitution* instances to the *TIA*'s output data!

The implementation still uses, according to the *TIA*'s specification, type assumptions for calculating types, but in terms of modifying the *AST*, type substitutions are more important.

Therefore the implemented *TIA* returns a data structure consisting of multiple tuples of *TypeAssumption* sets and *Substitution* sets. Each tuple represents a possible type configuration for the **Java** program.

5.3 Applying the Output Data

The question facing our implementation is, how to apply such a set of type substitutions returned by the *TIA* to the *AST*.

The most obvious solution would be to use the set of type substitutions as input data for a second algorithm which is responsible for applying them to the *AST*. Considering that the whole *AST* would have to be traversed again, the performance of this solution would not be very good.

Therefore we choose a totally different approach which is based on the *Observer Design Pattern* [7]. According to this design pattern, many *observers* (also called *listeners*) register themselves at an object they are interested in, so that they can be notified about its state changes.

In our case the observers are all the *AST* components which store a *TypePlaceholder* object. Such a component registers itself as an observer at the *TypePlaceholder* whose state changes it is interested in. The state changes are the type replacements and substitutions respectively. Such an observer is represented by the interface *ITypeReplacementListener* and is notified by a method call to its interface method *replaceType(ReplaceTypeEvent e)*. The observer can then replace its *TypePlaceholder* with the new type it receives via the *ReplaceTypeEvent*.

Each *TypePlaceholder* stores its *ITypeReplacementListeners* in a *Vector* called *m_ReplacementListeners* and notifies all registered observers when the method *fireReplaceTypeEvent()* is called (see figure 4).

As all observers are stored in a field variable of *TypePlaceholder*, it is important that all observers who are interested in a type placeholder A register at the same *TypePlaceholder* instance representing A. This means that within the *AST* there must not be more than one instance for the type placeholder A.

In order to achieve this, we prohibit the creation of *TypePlaceholder* objects by defining its constructor private. Instead we provide the static *Factory Method* [7] *TypePlaceholder.fresh()* which creates a new *TypePlaceholder* object and stores it in a central registry. An existing *TypePlaceholder* can be retrieved from the registry by the static method *TypePlaceholder.getInstance(String name)*.

So the following happens after the *TIA* has finished. The

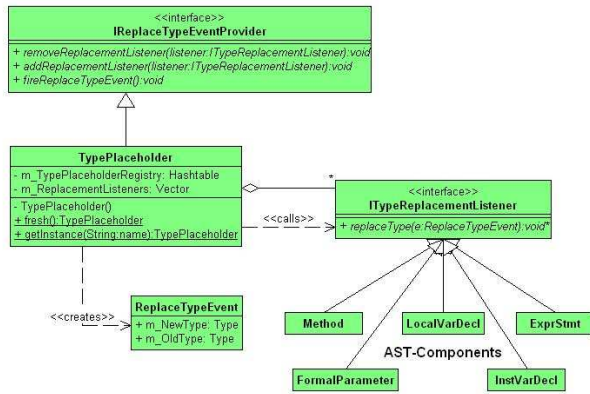


Figure 4: Implemented Observer Pattern

programmer has to choose between the possible type configuration returned by the TIA. On each *Substitution* object of the selected configurations the method *execute()* is called. This method requests the unique *TypePlaceholder* instance of its type placeholder from the registry and passes its calculated type to *TypePlaceholder.replaceWith(Type newType)* which triggers *TypePlaceholder.fireReplaceTypeEvent()*. Subsequently all registered *ITypeReplacementListeners* are notified and replace their *TypePlaceholder* with the calculated type.

6. CONCLUSION AND OUTLOOK

We gave a type inference algorithm for Java 5.0. The algorithm calculates a method's parameter types and its return type. For type terms without bounded type variables and without wildcards a principle type is inferred. The type inference algorithm is based on the type unification algorithm. It is possible to consider the type inference algorithm as a generic algorithm parameterized by the type unification algorithm. At the moment we gave a type unification algorithm, which calculates unifiers for type terms without bounded type variables and without wildcards.

For the introduction of bounded type variables step 4 of the type unification algorithm (section 3) must be extended. The pairs of the form $a < ty1$ and $a < ty2$, where the types $ty1$ and $ty2$ are not unifiable, should be transformed to $a < ty1 \& ty2$. Furthermore we aim to discover whether such a strategy leads to a principle type or not.

The introduction of wildcards leads to a problem if the type unification is computable. This is marked as open in [18]. There is an idea to solve this problem by extending our type unification algorithm.

Furthermore, we are working at a translation function for the byte-code to integrate intersection types as discussed in section 4.4.

7. REFERENCES

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*,

pages 31–41, 1993.

- [2] E. Berk. *JLex: A lexical analyzer generator for Java(TM)*. <http://www.cs.princeton.edu/apel/modern/java/JLex>, 1.2 edition, October 1997.
- [3] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. 9th Symposium on Principles of Programming Languages*, 1982.
- [4] A. Donovan, A. Kiežun, M. S. Tschantz, and M. 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.
- [5] J. Eifrig, S. Smith, and V. Trifonov. Type Inference for Recursively Constrained Types and its Application to Object Oriented Programming. *Electronic Notes in Theoretical Computer Science*, 1, 1995.
- [6] R. Fuhrer, F. Tip, A. Kiežun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005.
- [7] Gang of Four. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [8] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
- [9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java™ Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.
- [10] S. P. Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.
- [11] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.
- [12] B. Kühn and A.-T. Schreiner. *jay – Compiler bauen mit Yacc und Java*. iX, 1999. (in german).
- [13] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.
- [14] R. Milner. *The definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.
- [15] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. *Proceedings of ECOOP'92, Sixth European Conference on Object-Oriented Programming*, LNCS 615:329–349, July 1992.
- [16] M. Plümicke. *OBJ-P The Polymorphic Extension of OBJ-3*. PhD thesis, University of Tuebingen, WSI-99-4, 1999.
- [17] M. Plümicke. Type unification in Generic-Java. In M. Kohlhasse, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, July 2004.
- [18] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.