
# An Implementation of Type Inference for Featherweight Generic Java

Timpe Hörig

Chair of Programming Languages, University of Freiburg
timpe.hoerig@students.uni-freiburg.de

**Bachelor Thesis**

Examiner: Prof. Dr. Peter Thiemann

**Abstract.** Typesystems are an essential and powerfull part of every modern programming language. But to be forced to explicitly write down every typesignature can be annyoing or finding the most general types even can be difficult. By making the type system able to infer types by itself those problems become obsolete. This is a global typeinference algorithm for Featherweight Generic Java a minimal core calculus for Java (Featherweight Java) extended with Gemerics. [-?-] This implements a typeinference algorithm defined in the paper "Global Type Inference for Featherweight Generic Java" by Andreas Stadelmeier, Martin Plümicke and Peter Thiemann.

**Abstract. (german)** Abstact in german here ...

# Table of Contents

# 1 Introduction

## 1.1 Type Systems

Consider the following definition of the function `increment` which takes an argument `n`, increments it by one and returns the result.

```
increment(n) {
  return n + 1;
}
```

Calling this function with an `Integer`, for example 1, returns 2. However, calling it with a `String` results in a type error, because addition between a `String` and an `Integer` is not defined. Errors such as that are easy to make but cause the program to crash at runtime. With type systems it is possible to detect such errors at compile time and therefore increases the quality of programs dramatically. In order to detect such errors at compile time additional information is given to the function declaration. Consider the previous example but now with type annotations.

```
Integer increment(Integer n) {
  return n + 1;
}
```

Writing the type `Integer` in front of the function name indicates the function's return type. The arguments are annotated by writing their types in front of each argument. Now the type system knows of which type the argument must be and therefore calling `increment` with a `String` results in a type error at compile time rather than at runtime.

## 1.2 Type Inference

In order to correctly typecheck a whole program, everything that has a type needs an explicit type annotated, even if the type annotation seems to be redundant. For example an instantiation of a variable. An other down sight might be, that programmers do not always annotate the most general type. The `increment` function without any type annotations also works for other types that support the addition operator like `Double` or `Float`. But because of the type annotation only arguments of the type `Integer` are allowed.

– See more –

## 1.3 Featherweight Generic Java

Java is one of the most popular programming languages world wide. Extending Java with new features and providing proofs of soundness becomes more and more difficult as Java becomes more and more complex. To be able to provide proofs about complex programming languages often a smaller version of the programming languages is defined that contains less features but behaves similar.

Extending those smaller versions with new features leads to languages still small enough so providing proofs not only becomes doable but handy. Featherweight Java is exactly that. A smaller version of Java. In fact it is so small almost all features are dropped even assignments. Making Featherweight Java a functional subset of Java. Featherweight Java only contains classes with fields, methods and inheritance and five different forms of expressions: Variables, field lookups, method invocations, object creation and castings. As Java itself is not a functional programming languages it is easy to see that Featherweight Java is not equivalent to Java but because of it's similar behavior it is still possible to draw back conclusions to the full Java.

A typical Featherweight Java program may look like this:

```
class Pair extends Object{
  Object fst;
  Object snd;

  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}

(new Pair(new Pair(new Object(), new Object()), new Object()).fst).fst
```

A Featherweight Java program always consists of two parts. First, one ore more class definitions and second, an expression to be evaluated.

As well as Java, Featherweight Java includes a type system. With the type annotations given in the class definitions Featherweight Java is able to typecheck the given expression. In the example above the most inner expression that has a fixed type is the object creation of `new Pair(new Pair(new Object(), new Object()), new Object())`. It's type is `Pair`. Then the field `fst` is accessed. Because the field `fst` of class `Pair` is annotated with `Object` the resulting type is `Object` even if the expression itself is `new Pair(new Object(), new Object())`. Now trying to access the field `fst` the second time would be possible since the expression it is called on is an object creation of `Pair` but because it's type is `Object` this leads to a compile time error.

Instead of giving fields a concrete type like `Object`, giving them a type variable solves this problem. Extending Featherweight Java with Generics leads to Featherweight Generic Java. Type variables are just like normal variables but they range over types. In Featherweight Generic Java every type variables is given an upper bound. In the following example the type variables `X` and `Y` are introduced both with the upper bound `Object`. That means both type variables can range over any types that are a subtype of `Object`.

Type variables can also be used in method declarations to set relations between different arguments and or the return type. When a class is instantiated the type variables are also instantiated with a concrete type.

Rewriting the example above results in:

```
class Pair<X extends Object<>, Y extends Object<>> extends Object<>{
  X fst;
  Y snd;

  <Z extends Object<>> Pair<Z, Y> setfst(Z newfst) {
    return new Pair(newfst, this.snd);
  }
}

(new Pair<Pair<Object<>, Object<>>, Object<>>(new Pair<Object<>, Object<>>(new Object<>()
```

Now trying to access the field `fst` results in the same expression but with type `Pair<Object<>, Object<>>`. Therefore the second field lookup is successful.

Generics are very powerful and nice to have but at the same time make type annotations much more complicated and bigger. Adding type inference to Featherweight Generic Java would make it possible to rewrite the example above as follows:

```
class Pair<X extends Object<>, Y extends Object<>> extends Object<> {
  X fst;
  Y snd;

  setfst(newfst) {
    return new Pair(newfst, this.snd);
  }

  (new Pair(new Pair(new Object(), new Object()), new Object()).fst).fst
}
```

Here almost every type annotation is dropped except for class headers and field types.

The following sections first describe Featherweight Generic Java and Global Type Inference formal as it is describe in the Paper [...] (and [...]) except for a few changes and then shows how the Global Type Inference Algorithm is implemented.

## 2 Featherweight Generic Java

### 2.1 Syntax

### 2.2 Types

There are two different kinds of types (`T`, `U`, `V`) in Featherweight Generic Java. Type variables (`X`, `Y`, `Z`) and types other than type variables (`N`, `P`, `Q`). Writing `\overline {T}` short for `T_1, T_2, ... T_n`.

$$\begin{aligned} T &:= X \mid N \\ N &:= C\langle \overline{T} \rangle \end{aligned}$$

**Fig. 1.** Types

$$L := \text{class } C\langle \overline{X} \lhd \overline{N} \rangle \ \lhd \ D\{\ \overline{T}\ \overline{f}\ [K]\ \overline{M}\ \}$$

**Fig. 2.** Class Definitions

### 2.3 Class Definitions

A class definition `L` always begins with the keyword `class` followed by it's name. Then every type variable and it's upper bound that are used as a field type in the class definition are declared within <> writing `\vartriangleleft` as shortcut for the keyword `extends`. Followed by the name of the superclass. In the body of a classdefinition the fields (`f`, `g`) of the class are defined, writing their types infront of them. Followed by the optional constructor `K` and all methoddefinitions `\overline {M}`.

### 2.4 Methoddefinition

$$M := m(\overline{x})\ \{\text{return } e\}$$

**Fig. 3.** Method Definitions

As this is a type inference algorithm every type annotations of methods can be dropped. Leading a method definition only to consist of it's name `m`, it's arguments `\overline {x}` and a single expression `e` wich is returned.

### 2.5 Expressions

An expression `e` can be in five different forms. First, a simple variable `x`. Second, a field lookup written `e.f`. Third, a method invocation that takes one expression for each argument. Fourth, an object creation indicated by the keyword `new` followed by the name of the new Object and it's arguments. Fifth, a cast. A cast binds less than any other expression. Notice how methodcalls and object creation do not need any instanciation of their generic type annotations.

$$e := x \mid e.f \mid e.m(\bar{e}) \mid \text{new } C(\bar{e}) \mid (C)e$$

**Fig. 4.** Expressions

### 2.6 Typeannotation

Every typeannotation except for field types and their upper bounds in classdefinitions can be dropped.

### 2.7 Constructor

$$K := C(\bar{f},\ \bar{g})\ \{\text{super}(\bar{g});\ \text{this}.\bar{f} = \bar{f};\}$$

**Fig. 5.** Constructor

In Featherweight Java as well as in Featherweight Generic Java a constructor is always required. The constructor has always the same form. The name of the constructor is always the name of the class itself. For every field it takes excatly one argument wich must have the same name as the field it belongs to. The body of the constructor always consists of two parts. First, a call to super, in wich all arguments with fieldnames defined in superclasses are passed on. Second, every other argument is initialised in the form `this.f=f`.

As a constructor does not contain any information that cannot be found somewhere else in the class definition, writing the constructor for every class definition is redundant and can be dropped in this implementation.

## 3 Abstract Syntax Tree and Parser

### 3.1 Abstract Syntax Tree

Having a program in plain text form is rather useless for running the type inference algorithm. Instead, having the program in a specific form where classdefinitions can be accessed and went through is desirable. Such a form is called an Abstract Syntax Tree short AST. Every node of this AST is represented by a dataclass in Python [1].

The entry point for every full program is the class `Program`.

[1] FGJ_AST.py

```
[...]

ClassTable = dict[str, ClassDef]


@dataclass
class Program:
    CT: ClassTable,
    expression: Expression
```

The class `Program` has two fields: `CT` for Class Table a mapping from every class name to it's definition. Second, `expression` an expression to be evaluated.

The implementation of types, classdefinitions, methoddefinitions and expressions are straight forward adaptations of their formal definitions in the paper "Global Type Inference for Featherweight Generic Java".

For example the implementation for classdefinition `ClassDef`:

```
@dataclass
class ClassDef:
    name: str,
    superclass: Type,
    typed_fields: FieldEnv,
    methods: list[MethodDef]
```

There is no field for a constructor because as mentioned earlier constructors are dropped in this implemenatation.

The implemenatations for types and expressions differs a bit. Because there are different types and different expressions, for both first a plain class with no fields is defined `Type` and `Expressions` from wich then all the different kinds of types or expressions inherit. Therefore it is possible to match against types and expressions.

```
@dataclass
class Type:
    pass

@dataclass
class TypeVar(Type):
    ...
```

Every Featherweight Generic Java program can be represented by an AST.

### 3.2 Parsing

Parsing the programcode to an AST is done by using the python parser library Lark. Lark is powerful library that can parse any context-free grammar.

There are two parst when using Lark: First, the definition of the grammar. Because every Featherweight Generic Java program can now be represented as

an AST the grammar rules are trivial. First a rule for identifier is created. This representation allows to easily changed what is allowed to be an identifier at any time later. Then for every class defined for the AST a rule is define recursively. Scond, for every rule in the grammar a function is defined wich gets the parsed rule as an argument and returns a instanciation of the respective class of the AST. The most outer rule is `program` wich then returns the whole program represented by the AST starting with the class `Program`.

```
# grammar rules

identifier: ...

variable: identifier

[...]

# function for shaping

def variable(tuple_of_elemts_of_variable_rule):
    (name, ) = tuple_of_elemts_of_variable_rule
    return Variable(name)
```

## 4 Auxiliary Functions

In the following sections all auxiliary functions that are used in the typeinference algorithm are specified.

### 4.1 Auxiliary Typing

### 4.2 Subsitution

$$1 : [\mathrm{T}/\mathrm{X}]\mathrm{N}$$

$$2 : [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\mathrm{N}$$

$$3 : [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{N}}$$

**Fig. 6.** Substitution

The first rule (1) shows the standart substitution rule. In a non variable type N every variable type X is replaced by the given type T.

The second rule (2) is short for `[T_1/X_1, ... , T_n/X_n]N`.
The third rule (3) is short for `[T_1/X_1, ... , T_n/X_n]N_1, ... , [T_1/X_1, ... , T_n/X_n]N_n`.
These rules are implemented in different functions using pattern matching on the different type types.

### 4.3 Subtyping

$$1: \quad \Delta \vdash \mathrm{T} <: \mathrm{T}$$

$$2: \quad \frac{\Delta \vdash \mathrm{S} <: \mathrm{T} \quad \Delta \vdash \mathrm{T} <: \mathrm{U}}{\Delta \vdash \mathrm{S} <: \mathrm{U}}$$

$$3: \quad \Delta \vdash \mathrm{X} <: \Delta(\mathrm{X})$$

$$4: \quad \frac{\text{class C}<\overline{\mathrm{X}} \lhd \overline{\mathrm{N}}> \lhd \mathrm{N}\ \{...\} \quad \Delta \vdash \overline{\mathrm{T}} ok \quad \Delta \vdash \overline{\mathrm{T}} <: [\overline{\mathrm{T}}/\overline{\mathrm{X}}]\overline{\mathrm{N}}}{\Delta C < \overline{\mathrm{T}} > \ ok}$$

**Fig. 7.** Subtyping

### 4.4 Auxiliary Functions

## 5 Type Inference

The type inference algorithm looks at one class after another. Therefore all classes must be ordered in a way that method calls only call methods from classes defined before the current class. However, this does not constrain the algorithm as a program where this is not the case can be transformed to a program where it is. This transformation is shown in the paper [...].
In the following it is assumed that a program always fulfills this condition.

The algorithm mainly consist of two parts: First, constraint generation and second, constraint solving. As solving the constraints can lead to multiple solutions for a single class, simply one solution is assumed. If the algorithm later fails, it backtracks and assumes the next solution.

$$\begin{aligned}
&\text{FJTypeInference}(\prod,\ \text{class C}{<}\overline{\text{X}} \lhd \overline{\text{N}}{>} \lhd \text{N}\{...\}) = \\
&\qquad \text{let}\,(\overline{\lambda},\ \text{C}) = \text{FJType}(\prod, \text{class C}{<}\overline{\text{X}} \lhd \overline{\text{N}}{>} \lhd \text{N}\{...\}) \\
&\quad (\sigma, {<}\overline{\text{Y}} \lhd \overline{\text{P}}{>}) = \text{Unify}(\text{C}, \overline{\text{X}} <: \overline{\text{N}}) \\
&\qquad \text{in}\prod \cup\ \{(\text{C}{<}\overline{\text{X}} \lhd \overline{\text{N}}{>}.\text{m}\ :\ {<}\overline{\text{Y}} \lhd \overline{\text{P}}{>}\ \overline{\sigma(a)} \to \sigma(a)) \mid (\text{C}{<}\overline{\text{X}} \lhd \overline{\text{N}}{>}.\text{m} : \overline{a} \to a) \in \overline{\lambda}\}
\end{aligned}$$

**Fig. 8.** Type Inference

## 6 FJType

`FJType` is the function that generates constraints for a class. It takes the definition of that class and the method type environment where all the method signatures that have already been inferred from earlier classes are in. It returns a tuple consisting of a set of method signatures 'lambda' with new type variables and a constraint set C which constraints those type variables.

In Featherweight Generic Java methods can be overridden with covariant types. If a method is overridden or not can be checked by calling `mtype` on the class's supertype. If it does not find a method signature the type signature can be initialized with some new type variables which have only the constraint to be subtypes of `Object`. However, if the `mtype` lookup is successful the constraints look different. The return type of the current method needs to be a subtype of the return type of the overridden method. Whereas the types of the arguments stay the same. Then the method signature is added to the global method type environment. This is done for every method definition of the class. Then the `MethodType` function is called on every method defined in the class. Therefore, every method has access to the full method type environment. The method type environment and the generated constraints together with the constraints generated by `MethodType` are returned.

$$\begin{aligned}
&\text{FJType}(\prod, \text{class C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle \lhd \text{N}\{\overline{\text{T}}\ \bar{\text{f}};\ \overline{\text{M}}\}) = \\
&\text{let } \overline{a}_m \text{ be fresh type variables for each m} \in \overline{\text{M}} \\
\overline{\lambda}_0 = \{\text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m}:\ \langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{\text{T}} \to a_m \mid \text{m} \in \overline{\text{M}},\ &\text{mtype(m, N, } \prod) = \langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{\text{T}} \to \text{T}\} \\
C_0 = \{a_m < \text{T} \mid \text{m} \in \overline{\text{M}},\ &\text{mtype(m, N, } \prod) = \langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{T} \to T\} \\
\overline{\lambda}' = \{(\text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m}:\ \overline{a} \to a_m) \mid \text{m} \in \overline{\text{M}},\ &\text{mtype(m, N, } \prod) \text{ not defined}, \overline{a} \text{ fresh}\} \\
C_m = \{\{a_m < \text{Object}, \overline{a} < \overline{\text{Object}}\} \mid (\text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m}:\ \overline{a}\ to a_m) \in \overline{\lambda}'\} \\
\prod = \prod \cup \overline{\lambda}' \cup \overline{\lambda}_0 \\
\text{in } (\prod, C_0 \cup C_m \cup \cup textm \in \overline{\text{m}}\text{TYPEMethod}(\prod, C\langle\overline{\text{X}}\rangle, \text{m}))
\end{aligned}$$

**Fig. 9.** FJType

### 6.1 Method Type

$$\begin{aligned}
&\text{Type Method}(\prod, \text{C}\langle\overline{\text{X}}\rangle, \text{m}(\overline{\text{x}})\{\text{ return e;}\}) = \\
&\text{let}\langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{T} \to T = \prod(\text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m}) \\
&(\text{R}, \text{C}) = \text{TYPEExpr}((\prod; \{\text{this} : \text{C}\langle\overline{\text{X}}\rangle\} \cup \{\overline{\text{x}} : \overline{\text{T}}\}), \text{e}) \\
&\text{in} \text{C} \cup \{\text{R} < \text{T}\}
\end{aligned}$$

**Fig. 10.** Type Method

`MethodType` takes the method type environment, the class header `C<\overline {X}>` of the class that is currently checked and the method definitions as arguments. It first brings the type signature of the current method in scope and then calls `TypeExpression` on the expression `e` inside the body of the method. In this call also a type environment for variables is created, where `this` has the type C<X'> and the arguments of the method are read from the type signature brought into the scope earlier. `TypeExpression` returns a tuple of the type `R` of `e` and the generated constraints. The generated constraints together with the constraint that `R` must be a subtype of the return type of the method signature is returned.

### 6.2 Type Expressions

The last function for generating constraints is `Type Expression`. It takes the method type environment, the variable type environment created by `Type Method` and an expression `e` as arguments. The expression `e` is matched against the five possible forms for an expression and is handled differently for each.

### 6.3 Variable

$$\text{TYPEExpr}((\prod; \overline{\eta}), \text{x}) = (\overline{\eta}(\text{x}), \emptyset)$$

**Fig. 11.** Type Variable

If the expression is a simple variable it's type is looked up in the variable type environment and together with an empty set is returned. The empty set is returned because a simple variable does not generate any constraints.

### 6.4 Field Lookup

$$\text{TYPEExpr}((\prod; \overline{\eta}), e.f) =$$
$$\text{let}(\text{R}, C_R) = \text{TYPEExpr}((\prod; \overline{\eta}), e) a \text{ freshc} = \text{oc}\{\{\text{R} < \text{C}\overline{<\text{a}>}, \text{a} = [\overline{\text{a}}/\overline{\text{X}}]\text{T}, \overline{\text{a}} < [\overline{\text{a}}/\overline{\text{X}}]\overline{\text{N}} \mid a \text{ fresh}\}\}$$

**Fig. 12.** Type Field Lookup

In case of a field lookup `e.f` first the `Type Expression` function is called recursively on the expression `e`, returning a type R and a constraint set $C_R$. Now the difficult part is to find out which class the field `f` belongs to. The solution is simple, just generate a constraint set for every class `C` that has the field `f` and put them together in one big or-constraint. The constraint set for one class consists of three kinds of constraints. One that constraints the returned type R to be a subtype of the class `C` where the type variables of `C` are replaced by some fresh type variables `\overline {a}`. One that constraints the resulting type of the field lookup to be equal to the type of the corresponding field in `C`. And another one that constraints every $a_i$ of `\overline {a}` to be a subtype of the bounding type of the corresponding type variable it replaces.

### 6.5 Method Invocation

$$
\begin{aligned}
&\text{TYPEExpr}((\prod; \overline{\eta}), \text{e.m}(\overline{\text{e}})) = \\
&\text{let}(\text{R}, C_R) = \text{TYPEExpr}((\prod; \overline{\eta}), \text{e}) \\
&\forall e_i \in \overline{\text{e}} : (R_i, C_i) = \text{TYPEExpr}((\prod; \overline{\eta}), e_i) \\
&a \text{ fresh} \\
\text{c} = \text{oc}\{\{&\text{R} < \text{C}\overline{<\text{a}>}, \text{a} = [\overline{\text{b}}/\overline{\text{Y}}][\overline{\text{a}}/\overline{\text{X}}]\text{T}, \overline{\text{R}} < [\overline{\text{b}}/\overline{\text{Y}}][\overline{\text{a}}/\overline{\text{X}}]\overline{\text{T}}, \\
&\overline{\text{b}} < [\overline{\text{b}}/\overline{\text{Y}}][\overline{\text{a}}/\overline{\text{X}}]\overline{\text{P}}, \overline{\text{a}} < [\overline{\text{a}}/\overline{\text{X}}]\overline{\text{N}} \mid \overline{\text{a}}, b \text{ fresh}\} \\
&\mid (\text{C}\overline{<\text{X} \lhd \text{N}>}.\text{m} : \overline{<\text{Y} \lhd \text{P}>}\overline{T} \to T) \in \prod\}
\end{aligned}
$$

**Fig. 13.** Type Method Lookup

A method invocation imposes the same problem that a field lookup did and therefore is solved in a similar way. But first for the expression that the method `m` is invoked on and every expression that is passed as an argument to `m` `Type Expression` is called recursively on. Then for every method signature of a method with name `m` in the method type environment an constraint set is generated. This set of constraints contains five kinds of constraint, three of them are very similar to the ones generated for a field lookup except that two different kinds of fresh type variables are created. One for every generic type variable the class has and one for every generic type variable the method signature has. One that constraints the type of the arguments for m to be subtypes of the argument types imposed by the method signature. And the last one that constraints all fresh type variables generated for the type variables of the method signature to be a subtype of the corresponding bounding type.

### 6.6 Object Creation

The constraints generated for an object creation are rather simple. `TYPEExpr` is called recursively for every argument. Then the types of all fields of the class that the generated object is an instance of are brought in scope via the inlfields function therefor for every field a fresh type variable is introduced. Those fresh type variables must be subtypes of the corresponding upper bound and all returned types of the recursive call have to be subtypes of the field they are bound to.

### 6.7 Cast

For a cast the `TypeExpr` is called recursive on the expression of that cast but instead of returning the type that call returns, the type the expression is casted to together with the constraints generated by the recursive call are returned.

## 7 Unify

`FJUnify` is the function that tries to solve the constraints. This may lead to no, one or more possible solutions. If there is more than one solution simply the first solution is chosen. If there is no solution the overall algorithm backtracks to the last class where there was more than one solution and assumes the next one, then starts again from there. A solution is found if the constraint set can be transformed into solved form. A constraint set is in solved from if it only contains constraints of the following four forms:

1. $a < b$
2. $a = b$
3. $a < C\overline{<T>}$
4. $a = C\overline{<T>}$ with $a \notin \overline{T}$

No variable is allowed to occur twice on the left side of rules of the form three and four.

**Fig. 14.** Solved Form

`Unify` takes two arguments: First the constraint set to solve. Second, a mapping from type variables `X` to their upper bound `N` this mapping represents the type variables and their upper bound of the current class that the overall algorithm is currently checking. In order to not need a special treatment for these type variables all type variables $X_i$ are treated as parameterless classes $X_i<>$ with superclass $N_i$. The constraint set `C` given to `Unify` may contain multiple or-constraints. Flattening them to constraint sets `C'` that only contain simple-constraints leads to multiple `C'` that together cover all possible combinations of constraints. The following steps are done for the first `C'` if this leads to a solution, `TypeInference` continues with the next class. Otherwise, the next `C'` is tried.

**Step 1** The following rules are applied exhaustively to `C'`.

**Step 2** Now every constraint is either in solved form or in one of the following:
1. $\{C\langle\overline{T}\rangle < D\langle\overline{U}\rangle\}$ where `C` cannot be a subtype of `D`, therefore `C'` has no solution.
2. $\{a < C\langle\overline{T}\rangle,\ a < D\langle\overline{U}\rangle\}$ where `C` cannot be a subtype of `D` and vise versa. So `C'` has no solution.
3. $\{C\langle\overline{T}\rangle < a\}$

In the last case the non variable type $C\langle\overline{T}\rangle$ has an upper bound which is not a non variable type but a type variable which is not allowed in Featherweight Generic Java. This is solved by first searching the upper bound constraint of `a` if no such constraint exists `Object` is chosen as the upper bound. Then for every possible class from $C\langle\overline{T}\rangle$ up to the upper bound an or-constraint is generated which then replaces the one or two constraints it was generated from. The or-constraint is generated as follows:
HIER WAS ANDERS ALS IM PAPER

$$\text{expandLB}(C\langle\overline{T}\rangle < a,\ a < D\langle\overline{U}\rangle) = \{\{a = [\overline{T}/\overline{X}]N\} \mid \Delta \vdash C\langle\overline{X}\rangle < N,\ \Delta \vdash N < D\langle\overline{P}\rangle\}$$
$$\text{where } \overline{P} \text{ is determined by } \Delta \vdash C\langle\overline{X}\rangle < D\langle\overline{P}\rangle \text{ and } [\overline{T}/\overline{X}]\overline{P} = \overline{U}$$

**Fig. 15.** expandLB

In the paper ... is a second case described where such an upper bound can be implied. This case cannot happen in this implementation because of the added rules to step 1.
Now `C'` again may contain or-constraints. Therefore the constraint set `C'` is flattend once again and the following steps are done for each set `C''`.

**Step 3** k

## 8 Implementation

Python is not a functional language, therefore data structures like lists, dictionaries and sets are mutable. That imposes a problem since mutable data structures are not hashable but entries of sets must be. Therefore having a set of sets is not possible. In order to solve this problem all sets, lists and dictionaries are made immutable by using frozenlists, frozensets and frozendictionaries. As frozensets come with the default library they can be used easily. In order to have frozenlists and frozendicts the python libraries `FrozenList` and `frozendict` are used.

### 8.1 FJType

The implementation of the constraint generation is straight forward. Because every functions is described in pseudo code most of the functions can easily be translated into python code. Creating fresh type variables is simply done by creating a generator at the beginning that creates type variables of the form `x_0`, `x_1`, ... where `x` is a string given as argument to the generator instantiation.

## 9 Discussion and Conclusion

## References


1. Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S. & Stark, K. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal Of Functional Programming*. **29** pp. e19 (2019), http://dx.doi.org/10.1017/S0956796819000170
2. Barendregt, H. Introduction to generalized type systems. *Journal Of Functional Programming*. **1**, 125-154 (1991), https://doi.org/10.1017%2Fs0956796800020025
3. Bove, A., Dybjer, P., Norell, U. (2009). A Brief Overview of Agda – A Functional Language with Dependent Types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds) Theorem Proving in Higher Order Logics. TPHOLs 2009. Lecture Notes in Computer Science, vol 5674. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03359-9_6
4. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type classes in Haskell. ACM Trans. Program. Lang. Syst. 18, 2 (March 1996), 109–138. https://doi.org/10.1145/227699.227700
5. Chapman, J., Kireev, R., Nester, C. & Wadler, P. System F in Agda, for Fun and Profit. (2019,10)
6. Jones, M.P. (1992). A theory of qualified types. In: Krieg-Brückner, B. (eds) ESOP '92. ESOP 1992. Lecture Notes in Computer Science, vol 582. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-55253-7_17
7. Martin-Löf, P. & Sambin, G. Intuitionistic Type Theory. (Bibliopolis, 1984), https://www.cse.chalmers.se/~peterd/papers/MartinL%C3%B6f1984.pdf
8. Milner, R. A theory of type polymorphism in programming. *Journal Of Computer And System Sciences*. **17**, 348-375 (1978), https://www.sciencedirect.com/science/article/pii/0022000078900144
9. Odersky, M., Wadler, P. & Wehr, M. A Second Look at Overloading. *Proceedings Of The Seventh International Conference On Functional Programming Languages And Computer Architecture*. pp. 135-146 (1995), https://doi.org/10.1145/224164.224195
10. P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89). Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

**Declaration**

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg i. Br, 27.03.2023 M. Weidner

Place, Date Signature