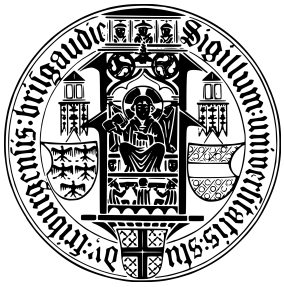

An Implementation of Type Inference for Featherweight Generic Java

Timpe Hörig

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

Bachelor Thesis

Examiner: Prof. Dr. Peter Thiemann
Advisor: Prof. Dr. Peter Thiemann

Abstract. Type systems are an essential and powerful part of every modern programming language. But to be forced to explicitly write down every type signature can be tiring or finding the most general types can even be difficult. By making the type system able to infer types by itself those problems become obsolete. This is a global type inference algorithm for Featherweight Generic Java: a minimal core calculus for Java (Featherweight Java) extended with generics. This implements a type inference algorithm defined in the paper "Global Type Inference for Featherweight Generic Java" by Andreas Stadelmeier, Martin Plümicke and Peter Thiemann.

Abstract. (german) Typsysteme sind mächtig und ein wichtiger Bestandteil einer jeden mordernen Programmiersprache. Jede Typsignatur explizit angeben zu müssen kann nicht nur ermüdend sein, auch den generellsten Typen zu finden kann schwierig sein. Nicht nur kann es ermüdend sein, jede Typsignatur explizit angeben zu müssen. Manchmal kann es sogar schwierig sein, den generellsten Typ zu finden. Wenn das Typsystem die Typsignatur selbst herleiten kann, entstehen diese Probleme erst gar nicht. Dies ist ein globaler Typinferenz-Algorithmus für Featherweight Generic Java: ein minimales Modell von Java (Featherweight Java), erweitert mit generischen Typen. Hier wird der Typinferenz-Algorithmus

implementiert, wie er in dem Paper "Global Type Inference for Featherweight Generic Java" von Andreas Stadelmeier, Marting Plümicke und Peter Thiemann beschrieben ist.

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 thus 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 thus 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, for an instantiation of a variable. Another downside might be, that programmers do not always annotate the most general type. Consider the function `plus` (+) without any type annotations. This function works for all types that support the addition operator like `Int`, `Double` or `Float`. Annotating the function (`plus`) with one of those types would mean it is not possible to call it with another one.
So having the advantages of a type system without being forced to write the additional information yourself would be optimal. Type inference does just this. However, type inference has its boundaries. Inferring every type without any information is often not just hard but impossible. Different languages have different restrictions to make global type inference possible.

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 them is defined that contains less features but behaves similarly. Extending those smaller versions with new features leads to languages still small enough so providing proofs not only becomes feasible 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 due to its similar behaviour 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 or 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())`. Its 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` for the second is not possible, even if the expression of the field lookup is an object creation of `Pair`, its type is `Object`. This results in a compile time error.

This problem can be solved by giving fields a type parameter like `X` instead of a concrete type like `Object`. Extending Featherweight Java with generics leads

to Featherweight Generic Java. Type parameters are just like normal parameters but they range over types. In Featherweight Generic Java every type parameter is given an upper bound. In the following example the type parameters `X` and `Y` are both introduced with the upper bound `Object`. That means both type parameters can range over any types that are a subtype of `Object`.

Type parameters 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 parameters are instantiated with a concrete type.

Rewriting the previous example 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<>(), new Object<>()),
  new Object<>()
).fst).fst
```

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

Generics are very powerful and useful 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 described in the paper "Global Type Inference for Featherweight Generic Java"[1] by Stadelmeier et al. except for a few changes. Afterwards, it shows how the Global Type Inference Algorithm is implemented.

2 Featherweight Generic Java

$$T := X \mid N$$

$$N := C\langle\overline{T}\rangle$$

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

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

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

Fig. 1. Syntax

Figure 1 defines the syntax of Featherweight Generic Java: types (`T` and `N`), class definitions (`L`), method definitions (`M`) and expressions (`e`).

2.1 Types

There are two different kinds of types (`T`, `U`, `V`) in Featherweight Generic Java: type parameters (`X`, `Y`, `Z`) and types other than type parameters (`N`, `P`, `Q`). In the following $\overline{T}$ is written short for $T_1, T_2, ...T_n$.

2.2 Class Definitions

A class definition `L` always begins with the keyword `class` followed by its name. Then, every type parameter and its upper bound, that are used as a field type in the class definition, are declared within <> writing $\lhd$ as shortcut for the keyword `extends`. This is followed by the name of the superclass indicated by another `extends` in front of it. In the body of a class definition the fields (`f`, `g`) of the class are defined, writing their types in front of them. Followed by the optional constructor `K` and all method definitions $\overline{M}$.

2.3 Method Definition

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

2.4 Expressions

There are five different forms of an expression `e`: a simple variable `x`, a field lookup written `e.f`, a method invocation that takes one expression for each argument, an object creation indicated by the keyword `new` followed by the name of the new Object and its arguments, a cast. A cast binds less than any other expression. Method calls and object creation do not need any instantiation of their generic type annotations.

2.5 Type Annotation

Every type annotation except for field types and their upper bounds in class definitions can be dropped.

2.6 Constructor

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

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 of the class the constructor takes exactly one argument which must have the same name as the field it belongs to. The body of the constructor always consists of two parts. First, there is a call to super, which calls the constructor of the superclass. Here, all arguments for fields that are not defined in the current class are passed on to. Second, every other argument is initialized 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 Auxiliary Functions

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

3.1 Substitution

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

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

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

The first rule (1) shows the standard substitution rule. In a non variable type N every type parameter X is replaced by the given type T.
The second rule (2) is short for $[\mathrm{T}_1/\mathrm{X}_1,\ \ldots\ ,\ \mathrm{T}_n/\mathrm{X}_n]\mathrm{N}$.
The third rule (3) is short for $[\mathrm{T}_1/\mathrm{X}_1,\ \ldots\ ,\ \mathrm{T}_n/\mathrm{X}_n]\mathrm{N}_1,\ \ldots\ ,\ [\mathrm{T}_1/\mathrm{X}_1,\ \ldots\ ,\ \mathrm{T}_n/\mathrm{X}_n]\mathrm{N}_n$.
These rules are implemented in different functions using pattern matching on the different types in `N`.

3.2 Substitution on constraints

$$1 : \quad [\mathrm{T}/\mathrm{a}]\mathrm{C}$$

$$2 : \quad [\mathrm{a}/\mathrm{b}]\mathrm{C}$$

Substitution is also possible on constraint sets. Type variables can be replaced by types (1) or by other type variables (2).

3.3 Subtyping

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

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

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

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

Fig. 2. Subtyping

Figure 2 shows the subtyping rules. By definition, a type is always a subtype of itself. Subtyping is also transitive and the environment delta Δ maps every type parameter to its upper bound. Thus, a type parameter is always a subtype of its upper bound. A class is always a subclass of its superclass. However, a generic class is only a subtype of its superclass, if the concrete types that replace the type parameters of the class also replace these type parameters in the superclass.

3.4 Field Lookup

$$\text{fields}(\text{Object}) = \{\}$$

$$\frac{\text{class C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle \lhd \text{N } \{\ \overline{\text{S}}\ \bar{\text{f}};\ [\text{K}]\ \overline{\text{M}}\ \} \quad \text{fields}([\overline{\text{T}}/\overline{\text{X}}]\text{N}) = \overline{\text{U}}\ \bar{\text{g}}}{\text{fields}(\text{C}\langle\overline{\text{T}}\rangle) = \overline{\text{U}}\ \bar{\text{g}},\ [\overline{\text{T}}/\overline{\text{X}}]\overline{\text{S}}\ \bar{\text{f}}}$$

A field lookup on a class `C` returns every field of `C` with its type, including all fields of all superclasses of `C` with their types. `Object` itself has no fields thus, `fields(Object)` returns the empty dictionary. The type parameters need a special treatment here. The field types of a class `C` may include type parameters. These must be substituted with the types the class is instantiated with. This also needs to be done for the superclass of `C` before the recursive call of `fields`.

3.5 Method Type Lookup

$$\frac{\begin{array}{c}\text{class C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle \lhd \text{N } \{\ \overline{\text{X}}\ \overline{\text{N}};\ [\text{K}]\ \overline{\text{M}}\ \} \quad \text{m} \in \overline{\text{M}} \\ \langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{\text{U}} \to \text{U} \in \Pi(\text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m})\end{array}}{\text{mtype}(\text{m},\ \text{C}\langle T\rangle,\ \Pi) = [\overline{\text{T}}/\overline{\text{X}}]\langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{\text{U}} \to \text{U}}$$

$$\frac{\text{class C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle \lhd \text{N } \{\ \overline{\text{X}}\ \overline{\text{N}};\ [\text{K}]\ \overline{\text{M}}\ \} \quad \text{m} \notin \overline{\text{M}}}{\text{mtype}(\text{m},\ [\overline{\text{T}}/\overline{\text{X}}]\text{N},\ \Pi)}$$

In Featherweight Generic Java methods of superclasses are inherited. Hence, method type lookups on classes with or without the method defined can be possible. If the method is defined in the class, the method type signature can be read off the method type environment. Once again, generic type parameters may occur in that type signature and must be substituted. If the method is not defined in the class `C`, then `mtype` is called on the superclass of `C` where the generic type parameters are substituted by the types the class `C` is instantiated with.

3.6 Generic Supertype

$$\text{genericSupertype}(\text{C}, \overline{\text{T}}, \text{C}) = \overline{\text{T}}$$

$$\frac{\text{class C}\langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle \lhd \text{C'}\langle\overline{\text{M}}\rangle\ \{...\}}{\text{genericSupertype}(\text{C}, \overline{\text{T}}, \text{D}) = \text{genericSupertype}(\text{C'}, [\overline{\text{T}}/\overline{\text{Y}}]\overline{\text{M}}, \text{D})}$$

`genericSupertype` takes two subtype related classes C and D and a list of types $\overline{\text{T}}$ as arguments. If the two classes are the same, the list of types $\overline{\text{T}}$ is returned. Otherwise, the function is recursively called with the superclass C' of C. However, the superclass C' may have different generic type parameters as C. That is why $\overline{\text{T}}$ is replaced by $\overline{\text{M}}$. $\overline{\text{M}}$ are the generic type parameters of C'. The generic type parameters $\overline{\text{Y}}$ may occur in $\overline{\text{M}}$. Thus, $\overline{\text{T}}$ is substituted for $\overline{\text{Y}}$ in $\overline{\text{M}}$.

4 Type Inference

The type inference algorithm looks at one class after another. Thus, 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 program. A program that does not respect this order can be transformed to a program that does. This transformation is described in the paper[1].
In the following it is assumed that a program always fulfills this condition.

$\text{T} := \text{a} \mid \text{X} \mid \text{N}$	type variable, type parameter, non type variable
$\text{N} := \text{C}\langle\overline{\text{T}}\rangle$	class type (with type variables)
$\text{sc} := \text{T} < \text{U} \mid \text{T} = \text{U}$	simple constraint: subtype or equality
$\text{oc} := \{\{\overline{\text{sc}}_1\},\ ..., \{\overline{\text{sc}}_n\}\}$	or-constraint
$\text{c} := \text{sc} \mid \text{oc}$	constraint
$\text{C} := \{\overline{\text{c}}\}$	constraint set
$\lambda := \text{C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle.\text{m} : \langle\overline{\text{Y}} \lhd \overline{\text{P}}\rangle\overline{\text{T}} \to \text{T}$	method type assumption
$\eta := \text{x} : \text{T}$	parameter assumption
$\Pi := \Pi \cup \overline{\lambda}$	method type environment
$\Theta := (\Pi; \overline{\eta})$	

Fig. 3. Constraint Syntax

[1] Chapter 4.1 Type inference for a program

In Figure 3 the syntax for constraints is defined. New type variables `a` are introduced. There are only two forms of simple constraints: equal and subtype constraints. An or-constraint represents different possibilities. Only one of the simple constraint sets in an or-constraint must be satisfied. A constraint can either be a simple constraint or an or-constraint. A method type assumption λ maps a method name and the class it is defined in to a method type signature. A parameter assumption η maps parameters to types.

The algorithm mainly consists of two parts: constraint generation and 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.

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

5 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 λ 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 superclass. If no method signature is found a new method signature for that method 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. Next, the `MethodType` function is called on every method defined in the class. Hence, 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.

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

5.1 Method Type

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

`TYPEMethod` takes the method type environment, the class header $\text{C}\langle\overline{\text{X}}\rangle$ 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 `TYPEExpr` on the expression `e` inside the method's body. In this call a type environment for variables is created. Where `this` has the type $\text{C}\langle\overline{\text{T}}\rangle$ and the types of the method's arguments are read off the type signature which was brought into the scope earlier. `TYPEExpr` 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.

5.2 Expression Type

The last function for generating constraints is `TYPEExpr`. It takes the method type environment, the variable type environment created by `TYPEMethod` 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.

5.3 Variable

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

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

5.4 Field Lookup

$$\begin{aligned}
&\text{TYPEExpr}((\Pi;\overline{\eta}),e.f) = \\
&\qquad \text{let } (\text{R},C_R) = \text{TYPEExpr}((\Pi;\overline{\eta}),e) \\
&\qquad\quad a \text{ fresh} \\
&\qquad\quad \text{c} = \text{oc}\{\{\text{R} < \text{C}\text{<}\overline{\text{a}}\text{>}, \text{a} = [\overline{\text{a}}/\overline{\text{X}}]\text{T}, \overline{\text{a}} < [\overline{\text{a}}/\overline{\text{X}}]\overline{\text{N}} \mid \overline{\text{a}} \text{ fresh}\} \\
&\qquad\qquad\qquad \mid \text{T f} \in \text{class C}\text{<}\overline{\text{X}} \lhd \overline{\text{N}}\text{>} \lhd \text{N } \{\ \overline{\text{X}}\ \overline{\text{N}};\ [\text{K}]\ \overline{\text{M}}\ \}\} \\
&\qquad \text{in } (\text{a},(\text{C}_R \cup \{\text{c}\}))
\end{aligned}$$

In case of a field lookup `e.f` first the `TYPEExpr` 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` defined 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 parameters of `C` are replaced by some fresh type variables $\overline{\text{a}}$. Another one that constraints the resulting type of the field lookup to be equal to the type of the corresponding field in `C`. And finally, one that constraints every a_i of $\overline{\text{a}}$ to be a subtype of the bounding type of the corresponding type parameter it replaces.

5.5 Method Invocation

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

A method invocation imposes the same problem that a field lookup did and thus 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`, `TYPEExpr` is called recursively on. Then, for every method signature of a method named `m` in the method type environment a 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 parameter the class has and one for every generic type parameter the method signature has. The other two constraints are new. The first constraint constraints the type of the arguments for `m` to be subtypes of the argument types imposed by the method signature. The second constraint constraints all fresh type variables generated for the type parameters of the method signature to be a subtype of the corresponding bounding types. All those constraint sets are put together to one big or-constraint which then is returned.

5.6 Object Creation

$$
\begin{aligned}
&\text{TYPEExpr}((\Pi;\overline{\eta}), \text{new C}(\overline{\text{e}})) = \\
&\qquad \text{let } \forall e_i \in \overline{\text{e}} : (\text{R}_i, \text{C}_i) = \text{TYPEExpr}((\Pi;\overline{\eta}), \text{e}_i) \\
&\qquad\quad \overline{\text{a}} \text{ fresh} \\
&\qquad\quad \text{fields}(\text{C}\langle\overline{\text{a}}\rangle) = \overline{\text{T}}\ \overline{\text{f}} \\
&\qquad\quad \text{C} = \{\overline{\text{R}} < \overline{\text{T}} \cup \{\overline{\text{a}} < [\overline{\text{a}}/\overline{\text{C}}]\overline{\text{N}}\} \quad \text{where} \\
&\qquad\qquad\qquad \text{class C}\langle\overline{\text{X}} \lhd \overline{\text{N}}\rangle \lhd \text{N}\{...\}\} \\
&\qquad \text{in } \ (\text{C}\langle\overline{\text{a}}\rangle, \text{C} \cup \bigcup_i \text{C}_i)
\end{aligned}
$$

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 new object is an instance of are brought in scope via the `fields` function. Thus, for every field a fresh type variable is introduced. Those fresh type variables must be subtypes of their corresponding upper bound and all returned types of the recursive calls have to be subtypes of the field they are bound to.

5.7 Cast

$$
\begin{aligned}
&\text{TYPEExpr}((\Pi;\overline{\eta}), (\text{N})\text{e}) = \\
&\qquad \text{let } \ = (\text{R}, \text{C}) = \text{TYPEExpr}((\Pi;\overline{\eta}), \text{e}) \\
&\qquad \text{in } \ (\text{N}, \text{C})
\end{aligned}
$$

For a cast the `TypeExpr` is called recursively on the expression that is casted. This recursive call returns a type and a constraint set. However, instead of

returning that type, the type the expression is casted to and the constraint set are returned.

6 Unify

The function `FJUnify` solves a constraint set. Solving the constraints 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. $\mathrm{a} < \mathrm{b}$
2. $\mathrm{a} = \mathrm{b}$
3. $\mathrm{a} < \mathrm{C}\!<\!\overline{\mathrm{T}}\!>$
4. $\mathrm{a} = \mathrm{C}\!<\!\overline{\mathrm{T}}\!>$ with $\mathrm{a} \notin \overline{\mathrm{T}}$

No variable is allowed to occur twice on the left side of rules 3 and 4.

`Unify` takes two arguments: First the constraint set to solve. Second, a mapping from type parameters `X` to their upper bound `N` this mapping represents the type parameters and their upper bounds of the current class that the overall algorithm is currently checking. In order to not need a special treatment for these type parameters all type parameters 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 rules defined in Figure 4 are applied exhaustively to `C'`.

$$\frac{C \cup \{a < C\langle\overline{T}\rangle,\ a < D\langle\overline{V}\rangle\}}{C \cup \{a < C\langle\overline{T}\rangle,\ C\langle\overline{T}\rangle < D\langle\overline{V}\rangle\}} \quad \Delta \vdash C\langle\overline{X}\rangle <: D\langle\overline{N}\rangle \qquad \text{match}$$

$$\frac{C \cup \{C\langle\overline{T}\rangle < a,\ D\langle\overline{V}\rangle < a\}}{C \cup C\langle\overline{T}\rangle < D\langle\overline{V}\rangle,\ D\langle\overline{V}\rangle < a} \quad \Delta \vdash C\langle\overline{X}\rangle <: D\langle\overline{N}\rangle \qquad \text{match reverse}$$

$$\frac{C \cup \{a < C\langle\overline{T}\rangle,\ b <^* a, b < D\langle\overline{U}\rangle\}}{C \cup \{a < C\langle\overline{T}\rangle,\ b <^* a,\ b < D\langle\overline{U}\rangle,\ b < C\langle\overline{T}\rangle\}} \qquad \text{adopt}$$

$$\frac{C \cup \{C\langle\overline{T}\rangle < a,\ a <^* b, D\langle\overline{U}\rangle < b\}}{C \cup \{C\langle\overline{T}\rangle < a,\ a <^* b,\ D\langle\overline{U}\rangle < b,\ C\langle\overline{T}\rangle < b\}} \qquad \text{reverse adopt}$$

$$\frac{C \cup \{C\langle\overline{T}\rangle < D\langle\overline{U}\rangle\}}{C \cup \{D\langle[\overline{T}/\overline{X}]\overline{N}\rangle = D\langle\overline{U}\rangle\}} \Delta \vdash C\langle\overline{X}\rangle <: D\langle\overline{N}\rangle \qquad \text{adapt}$$

$$\frac{C \cup \{D\langle\overline{T}\rangle = D\langle\overline{U}\rangle\}}{C \cup \{\overline{T} = \overline{U}\}} \qquad \text{reduce}$$

$$\frac{C \cup \{a_1 < a_2,\ a_2 < a_3,\ ...,\ a_n < a_1\}}{C \cup \{a_1 = a_2,\ a_2 = a_3,\ ...\}} \quad n > 0 \qquad \text{equals}$$

$$\frac{C \cup \{a = a\}}{C} \qquad \text{erase}$$

$$\frac{C \cup \{N = a\}}{C \cup \{a = N\}} \qquad \text{swap}$$

Fig. 4. resolve rules

For `match` and `adopt` in this implementation also the dual rules are defined. Thus only the highest non variable type can have an illegal upper bound by a type variable. This decreases the amount of or-constraints generated.

Step 2 Check if one of the following cases applies, if not the constraint set is in solved form:
1. $\{C\langle\overline{T}\rangle < D\langle\overline{U}\rangle\}$ where `C` cannot be a subtype of `D`, as result `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. $\{\mathrm{C}\langle\overline{\mathrm{T}}\rangle < a\}$

In the last case the non variable type $\mathrm{C}\langle\overline{\mathrm{T}}\rangle$ 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 $\mathrm{C}\langle\overline{\mathrm{T}}\rangle$ 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:

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

Such an upper bound constraint can also be implied by two or more other constraints of the form ($\mathtt{a} < \mathrm{C}\langle\overline{\mathrm{T}}\rangle$) and ($\mathtt{a} <^* \mathtt{b}$). `b` can either be a subtype of $\mathrm{C}\langle\overline{\mathrm{T}}\rangle$ in this case everything is fine, or it can be an upper bound to $\mathrm{C}\langle\overline{\mathrm{T}}\rangle$. The later case implies the constraint ($\mathrm{C}\langle\overline{\mathrm{T}}\rangle < \mathtt{b}$). This constraint can be handled the same way as a direct upper bound constraint but the possibility of `b` being a subtype of $\mathrm{C}\langle\overline{\mathrm{T}}\rangle$ needs to be taken into account. Thus the constraint ($\mathtt{b} < \mathrm{C}\langle\overline{\mathrm{T}}\rangle$) is added as an own possibility to the or-constraint.

Now `C'` again may contain or-constraints. The constraint set `C'` is flattened once again and the following steps are done for each simple-constraint set `C"`.

Step 3 The following rule is applied exhaustively to C".

$$\frac{C \cup \{\mathrm{a} = \mathrm{T}\}}{[\mathrm{T}/\mathrm{a}]\mathrm{C} \cup \{\mathrm{a} = \mathrm{T}\}} \quad \text{a occurs in C but not in T}$$

Step 4 If the constraint set C" was changed by the rule `subst`, the algorithm starts over with C" from step 1.

Step 5 Now the constraint set C" is in solved form. Constraints of the first form (a < b) do not bring any information with them, thus these constraints can be eliminated. This is done by exhaustively applying the rules sub-elimination and erase.

$$\frac{\mathrm{C} \cup \{\mathrm{C} \cup \mathrm{a} < \mathrm{b}\}}{[\mathrm{a}/\mathrm{b}]\mathrm{C} \cup \{\mathrm{b} = \mathrm{a}\}} \quad \text{sub elim}$$

$$\frac{\mathrm{C} \cup \{\mathrm{a} = \mathrm{a}\}}{\mathrm{C}} \quad \text{erase}$$

Step 6 The constraint set C is divided in $C_<$ and in $C_=$ where $C_<$ contains all subtype constraints and $C_=$ all equal constraints. Then new generic type parameters $\overline{Y}$ are generated, one for each constraint in $C_<$. Now the substitution σ and the mapping from $\overline{Y}$ to their upper bounds can be read off as follows:

$$\sigma = \{b \to [\overline{Y}/\overline{a}]T \mid (b = T) \in C_=\} \cup \{\overline{a} \to \overline{Y}\} \cup \{b \to X \mid (b < X) \in C_<\}$$

$$\gamma = \{Y \lhd [\overline{Y}/\overline{a}]N \mid (a < N) \in C_<\}$$

The pair of σ and γ is returned.

7 Implementation

This Global Type Inference algorithm is implemented in Python (3.10).

Python is not a functional language, hence data structures like lists, dictionaries and sets are mutable. That imposes a problem because mutable data structures are not hashable but entries of sets must be. 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.
One important environment that is passed to almost every functions but never mentioned in the abstract definitions is the `Class Table`. The Class Table or short CT is a mapping from class names to their class definitions. This environment is filled at the parsing step and never changed afterwards. This environment makes it easy to have full access to any class at any time only by having their name.

7.1 TypeInference

The function `TypeInference` connects constraint generation with constraint solving. The implementation of it differs from the idea describe in the paper. Classes are processed one after another, thus every class has access to the method types inferred for all methods that are defined in earlier classes. First, `Type` is called. The current class and the method type environment containing all the method type assumptions of earlier classes are given as arguments. It returns a method type assumption for every method defined in the current class together with a constraint set.

Then `Unify` with that constraint set is called. `Unify` needs a mapping from every type parameter that occurs in the constraint set to its upper bound (which is a non type variable). Type parameters that may occur are the field types of the current class and type parameter in method types from methods defined

in earlier classes that are overridden in the current class definition. The upper bounds of the field types can be directly read off the class definition. The upper bounds of the type variables occurring in method types can be read off the method type assumption that is returned from the call to `Type`. Both together are passed to `Unify`.

`Unify` returns a generator that yields every possible solution, one at a time. The following steps are done for every possible solution by simply looping over them.

A solution consists of two parts. First, a mapping from every type variable occurring in the constraints to a type parameter or a non variable type. Second, a mapping from every new type parameter introduced by `Unify` to its upper bound. For every method type assumption created by `Type` all type variables are replaced by the type they are mapped to and a generic type annotation is added. This generic type annotation maps every type parameter that occurs in the method type assumption to its upper bound. Finding all type parameters that do occur is done by the function `getTypeSigOf` which traverses every type of the method type assumption recursively and keeps a list of all type variables that occur.

At this point additional checks can be added, for example type checking the program. If one of those checks fails the algorithm would skip to the next possible solution.

Then `TypeInference` is recursively called with the new method type assumptions and the next class.

The next class is found by giving `TypeInference` a new argument `index` that points to the class that is currently checked and incremented by one before passing it to the recursive call. If the index is bigger than the amount of classes the program has, the algorithm finishes and a solution is found. However this solution may not be a correct one. This can happen for example when overriding a method. In order to obtain a correct global solution the program must be type checked. This is not done in this implementation.

If overall no solution is found the exception `NoSolutionFound` is raised.

7.2 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 instantiating a generator at the beginning that generates type variables of the form x_0, x_1, ... where `x` is a string given as argument to the generator instantiation.

7.3 FJUnify

Unify may produce more than one solution for a class. One of these solutions is assumed and the next class is processed. If a solution does not fit to the overall

solution the next one is assumed. If all solutions of a class fail, the algorithm backtracks to the class processed before and assumes the next solution of that class. This is depth first search. Thus it is more efficient if not all solutions of `FJUnify` are calculated at once, but just when they are needed. To do this the whole function `FJUnify` is implemented as a generator, yielding one solution after another.

Resolving or-constraints Converting a constraint set `C` which contains simple-constraints as well as or-constraints equals a Cartesian product with n sets, where n is the number of or-constraints in `C`. First the constraint set `C` is divided in two parts. One that contains all simple-constraints and one that contains all or-constraints. Then the Cartesian product of all or-constraint is calculated. To each result all simple constraints are added. This is exponential in the number of or-constraint sets, thus this is implemented as a generator (`gen_C_prime`) that yields one solution after another. `gen_C_prime` has an optional argument to give or-constraints an ordering. If such an ordering is provided the constraints inside the or-constraints are considered in that ordering.

Treating type parameters as parameterless classes All rules of step 1 and following, would also apply for generic type parameters. So every rule also must be defined for them and every possible combinations. In order to avoid this, generic type parameters are treated as parameterless classes with their upper bounds being now their superclasses. One important thing here is, that those new classes are not added to the Class Table, instead they are stored in an extra environment. When looking up classes in the Class Table an extra check must be done if the class is in it, if not the superclass can be read out of the new environment. The transformation from generic type parameters to parameterless classes and backwards is done in two extra functions that take and return a constraint set. These functions first apply the transformations to both side of each constraint. In case of a non type variable, the transformations are applied recursively.

Exhaustively applying rules There is not really a nice way in python to do such a thing. The most simple way to do so is to have a while-loop on a condition variable `changed` that is True at the beginning but changes to False as soon as the while-loop is entered. Then, if one rule applies and changes something this condition variable is set to True. Some rules exists that work on two constraint, thus every possible combination of constraints must be tried. This is done by two nested for-loops. Applying a rule changes the constraint set, that imposes a problem. Changing a set while iterating over it is not possible. Thus the set is copied at the beginning, as soon as one rule applies, a constraint may be removed of the constraint set. However, that does not effect the copy, resulting in another problem: Constraints that were removed are not allowed to be used again. A simple but not efficient way to solve this is to restart the for-loop every time a rule applies.

Rules in Figure 4 Rules have different many constraint as pre-condition, for rules that have more than one pre-condition every possible combination of constraints must be considered. Rules that take arbitrary many sub-constraints on type variables can be implemented very easily by taking just one (in case of the rule equals) or none (in case of adopt) sub-constraint on type variables. Then the two type variables that do occur in the constraints (either in the one subtype constraint on type variables or in the two other constraints needed for adopt) are checked if they are in any case subtype related. This leads to a maximum of two pre-condition, thus two nested for-loops over the constraint set are needed. Deciding which rule to apply is done by pattern matching against sub- or equal-constraints and on the types occurring in them. The rules `match` and `adapt` need a check if two types are subtype related, this is done with the function `genericSupertype`.

The rule `adopt` imposes a problem. If the upper bound of `b` is not `Object` but the upper bound of `a` is, then the constraint (`b` $<$ `Object`) is added. However this constraint together with the original subtype constraint on `b` can be resolved with `match` and `reduce` resulting in a constraint set equivalent to the constraint set at the beginning. This results in an endless loop where the constraint set always changes. Thus if the constraint that would be added by `adopt` is a trivial constraint (something is a subtype of `Object`) it is not added and thus not changing the constraint set.

ExpandLB A subtype constraint with a type variable as upper bound is not legal in Featherweight Generic Java. A type variable as upper bound indicates an unbounded type parameter which per definition is not possible. Thus an upper bound constraint for that type variable is needed. Such an upper bound constraint might already be in the constraint set. If not, a new subtype constraint for that type variable bounded by `Object` is created and added to the constraint set. In the case that an upper bound constraint already existed it needs to be checked if the lower bound is a subtype of the upper bound. If not, this constraint set has no solution. This is done by replacing the upper and lower bound constraints by one subtype constraint of the lower and upper bound. Then the algorithm continues resolving, if this does not end in an unsolvable constraint the subtype constraint is fulfilled. Then an or-constraint that contains an equal constraint for the type variable and all possible types between the lower and upper bound (inclusively) is added to the constraint set. All possible candidates are observed by calling `genericSupertype` on the lower and upper bound while maintaining a list of all intermediate steps.

As constraint sets are described and implemented as sets, they do not impose any ordering. In general this is not a problem because the overall algorithm produces all possible solutions. However, in this implementation or-constraints generated by `expandLB` have a hidden ordering such that the most specific constraint is taken first. This hidden ordering is passed to `gen_C_prime` which flattens the or-constraints. If all possible solutions are considered this does not change any-

thing. However, this may be important in the feature to find the most specific solution.

When no such upper bound constraint with a type variable exists it still can be implied by some other constraints of the form ($a <$ C$<T>$) and ($a <^* b$). The type variable can be a subtype of `C` or a supertype. If it is a supertype it is handled by creating an or-constraint as before, but if not, the case that `b` can be a subtype of `C` must also be considered. In order to handle this, the subtype constraint ($b <$ C$<T>$) is added as one option to the or-constraint. If such a subtype constraint is added to the or-constraint it is always added as first possible solution.

7.4 Example

Consider the following Featherweight Generic Java program without any type annotation for methods:

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

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

This example only contains one class definition, thus `FJType` and `Unify` run exactly once. Still, `Unify` may return more than one solution. First, we look at `FJType` and then at `Unify`.

In the following the fresh introduced type variables have the form a_0, a_1, ... the number respects the order in which they are generated. In the following description the variables do not occur in the right order. This is because the algorithm is recursive and explaining the algorithm is easier if some information is given that in the algorithm itself would be known later. The name of the type variables does not matter and could be different. This is simply done to showcase the order of which the type variables are generated.

In the following multiple arguments to a function are represented in a list. Thus ($\overline{\text{a}} \rightarrow$ a) is equivalent to ([a_1, a_2, ...] $\rightarrow \text{a}_0$).

FJType The function `FJType` takes two arguments. First, the method type environment, because `Pair` is the first class to be processed this environment is empty. Second, the class definition of `Pair`. Now for every method defined in `Pair` new type variables are introduced. Here the only method is `setfst` which takes one argument, thus two type variables are introduced, one for the return type and one for the argument type. This type annotation is added to the method

type environment. Because there are no other definitions of the method `setfst` in any superclasses of `Pair` the only constraints for both variables are that they need to be subtypes of `Object`. Thus the constraint set and the method type environment look like this:

$$\lambda = \{(\text{Pair<X} \lhd \text{Object>.setfst}) : \ [\text{a}_1] \to \text{a}_0\}$$
$$\text{C} = \{\text{a}_0 < \text{Object}, \ \text{a}_1 < \text{Object}\}$$

Next for every method defined in `Pair` the function `TYPEMethod` with the new type environment is called. In this case only once for the method `setfst`. `TYPEMethod` creates the local variable type environment where `this` has type `Pair<X, Y>` (`this` always refers to the class it is in, which in this case is `Pair`) and `newfst` has type a_1.

Then the body of `setfst` is processed. The body of a method always is one expression which here is an object creation of `Pair`. First we get the types of the fields of `Pair<X, Y>` while substituting `X` and `Y` by some fresh type variables a_5 and a_6. Both of them need to be subtypes of `Object`. Here, because all fields are defined in the current class this result in the types a_5 and a_6. Now we process the arguments of `new Pair(newfst, this.snd)`. The first argument is `newfst` which is a simple variable. We know it must be a subtype of a_5, if we lookup the type of the variable `newfst` in the variable type environment we obtain that `newfst` has type a_1. Thus generating the constraint $a_1 < a_5$. The second argument is `this.snd` which is a field lookup. We again know it must be a subtype of a_6. A fresh type variable a_2 is introduced representing the type of `this.snd`, thus we obtain the constraint $a_2 < a_6$. a_2 represents `this.snd` which can be further constrained. We know `this` is a simple variable of type `Pair<X, Y>`. Now for every class where the field `snd` is defined we generate an or-constraint. Here `Pair<X, Y>` is the only class with the field `snd`. Thus in this or-constraint `this` is a subtype of `Pair<X, Y>` where we substitute some fresh type variables for `X` and `Y` resulting in the constraint `Pair<X, Y>` $<$ `Pair<`a_3, a_4`>`. In the case that the field `snd` refers to the class `Pair<`a_3, a_4`>` we obtain that a_2 is equal to a_3. Both a_3 and a_4 need to be subtypes of `Object`. Now the constraint set looks like this:

$$\begin{aligned}\text{C} = \{&\text{a}_0 < \text{Object}, \ \text{a}_1 < \text{Object},\\ &\text{a}_1 < \text{a}_5, \ \text{a}_5 < \text{Object}, \ \text{a}_2 < \text{a}_6, \ \text{a}_6 < \text{Object},\\ &\{\{\text{Pair<X, Y>} < \text{Pair<a}_3, \text{a}_4\text{>}, \ \text{a}_2 = \text{a}_4, \ \text{a}_3 < \text{Object}, \ \text{a}_4 < \text{Object}\}\}\\ &\}\end{aligned}$$

We know that `setfst` returns a `Pair`, we introduced the type variables a_5 and a_6 to represent the types of the parameters of the object creation, hence the

return type of `setfst` is `Pair`$<a_5, a_6>$. At the beginning we said the return type of `newfst` is a_0 these two things lead to the last constraint `Pair`$<a_5, a_6> < a_0$. Resulting in the full method type environment and constraint set:

$$\begin{aligned}
\lambda &= \{(\text{Pair}<\text{X} \lhd \text{Object}>.\text{setfst}) : \ [\text{a}_1] \to \text{a}_0\} \\
\text{C} &= \{\text{a}_0 < \text{Object},\ \text{a}_1 < \text{Object}, \\
&\quad \text{a}_1 < \text{a}_5,\ \text{a}_5 < \text{Object},\ \text{a}_2 < \text{a}_6,\ \text{a}_6 < \text{Object}, \\
&\quad \{\{\text{Pair}<\text{X}, \text{Y}> < \text{Pair}<\text{a}_3, \text{a}_4>,\ \text{a}_2 = \text{a}_4,\ \text{a}_3 < \text{Object},\ \text{a}_4 < \text{Object}\}\}, \\
&\quad \text{Pair}<\text{a}_5, \text{a}_6> < \text{a}_0 \\
&\quad \}
\end{aligned}$$

Now we have all constraints and method types that we need and can start to solve them.

Unify The first thing that `Unify` does is to resolve the or-constraints. Here we have only one possibility in the or-constraint, thus we can just add all constraints together. Before we go on we need to transform all type parameters to parameterless classes ($\text{X} \to \text{X}<>$). Then we continue with step 1. The only rule that applies to anything is `adapt`.

$$\frac{\text{C} \cup \{\text{Pair}<\text{X}<>, \text{Y}<>> < \text{Pair}<\text{a}_3, \text{a}_4>\}}{\text{C} \cup \{\text{Pair}<\text{X}<>, \text{Y}<>> = \text{Pair}<\text{a}_3, \text{a}_4>\}} \quad \text{adapt}$$

Now we can resolve this further with `reduce`:

$$\frac{\text{C} \cup \{\text{Pair}<\text{X}<>, \text{Y}<>> = \text{Pair}<\text{a}_3, \text{a}_4>\}}{\text{C} \cup \{\text{X}<> = \text{a}_3,\ \text{Y}<> = \text{a}_4\}} \quad \text{reduce}$$

Then the two resolved rules are `swap`ped such that the type variables `a` occur on the left hand side.

$$\frac{\text{C} \cup \{\text{X}<> = \text{a}_3\}}{\text{C} \cup \{\text{a}_3 = \text{X}<>\}} \quad \text{and} \quad \frac{\text{C} \cup \{\text{Y}<> = \text{a}_3\}}{\text{C} \cup \{\text{a}_3 = \text{Y}<>\}} \quad \text{swap}$$

No other rules apply and we continue with step 2.

We find the constraint ($\text{Pair}<\text{a}_5,\ \text{a}_6> < \text{a}_0$) which has a type variable as upper bound and thus requires us to resolve it with `expandLB`. We search the upper bound constraint for a_0 which is ($\text{a}_0 < \text{Object}<>$). Next we check if $\text{Pair}<\text{a}_5,\ \text{a}_6>$ can be a subtype of `Object` which is trivially satisfied (because everything is a subtype of `Object`) and thus we know a_0 is either $\text{Pair}<\text{a}_5,\ \text{a}_6>$

or `Object`. We add those two constraints as an or-constraint to the constraint set, but now we have again or-constraints and thus have to flatten the constraint set once again. However, this time the or-constraint has two possible constraint sets resulting overall in two different simple constraint sets which we both have to consider one after the other. Here it is important to look at the constraint set that is more specific first. We continue with step 3 and the first simple-constraint set. We can substitute in three different cases: ($a_2 = a_4$), ($a_3 = X<>$) and ($a_4 = Y<>$). These substitutions change our constraint set and we start again from step 1 with the following constraint set:

$$\begin{aligned} C = \{&Y<> < a_6,\ a_1 < a_5,\ a_6 < \text{Object}<>,\ a_1 < \text{Object}<>,\\ &X<> < \text{Object}<>,\ a_4 = Y<>,\ Y<> < \text{Object}<>,\\ &a_0 = \text{Pair}<a_5,\ a_6>,\ a_3 = X<>, a_2 = Y<>,\ a_5 < \text{Object}<>\\ &\} \end{aligned}$$

With adapt and reduce we can drop both, ($X<> < \text{Object}<>$) and ($Y<> < \text{Object} < >$). Then we continue with step 2. Once again we find a constraint that has a type variable as upper bound, this time ($Y<> < a_6$) the corresponding upper bound is ($a_6 < \text{Object}<>$). Resolving this with `expandLB` gives us again two constraint sets, one with ($a_6 = Y<>$) and one with ($a_6 = \text{Object}<>$). We consider the more specific first and continue with step 3. We substitute $Y<>$ for a_6, this changes our constraint set and we start again from step 1. This time no rule of step 1 to 4 applies and we can jump right to step 5. Here we find the constraint ($a_1 < a_5$), we can substitute a_1 for a_5 and add the constraint that they need to be equal. No further rule applies and we can go to step 6. Before we start with step 6 we need to transform the type parameters from parameterless classes back to normal type parameters ($X<> \rightarrow X$). Now we can start with step 6. We first divide the constraint set into sub- and equal-constraints resulting in the following:

$$\begin{aligned} C_= &= \{a_0 = \text{Pair}<a_1,\ Y>,\ a_6 = Y,\ a_5 = a_1,\ a_4 = Y,\ a_3 = X,\ a_2 = Y\}\\ C_< &= \{a_1 < \text{Object}<>\} \end{aligned}$$

For the sub-constraint we introduce a new type parameters `Z` with its upper bound determined by the sub-constraint. We substitute every right hand side of all equal-constraints until there are no more occurrences of type variables.

The type signature for `newfst` determined by `FJType` was $[a_1] \rightarrow a_0$, if we substitute now we get the type signature $[Z] \rightarrow \text{Pair}<Z,\ Y>$. The last thing to do is to add the upper bounds to all new generic types and we have the full type signature:

$$<Z \lhd \text{Object}<>>[Z] \rightarrow \text{Pair}<Z,\ Y>$$

A more advanced example is shown in Figure 5.

```
class Int<> extends Object<> {
  id(x) {
    return x;
  }
}

class SomeMethods<> extends Object<> {
  idd(x) {
    return x.id(x);
  }
}

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

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

  setboth(newfst, newsnd) {
    return new Pair(
      this.setfst(newfst).fst,
      (int<>)this.idd(newsnd.id(newsnd))
    );
  }
}
```

Fig. 5. Example 2

We process one class after another, the first class does not bring any new features we have not seen in the first example. The inferred method type of `id` is $< \mathrm{Z}_0 \lhd \mathrm{Object}<> > [\mathrm{Z}_0] \to \mathrm{Z}_0$. With this information the next class `SomeMethods` is processed. While processing the body of `idd` we encounter a method call of `id` thus every method with that name in the method type environment is considered. The method type environment contains only one method signature. The one for `id` of class `Int` thus we know two things. First, `x` has to be of type `Int` because `Int` is the only class that defines the method `id`. Second, the type signature of `id` states that the argument and the return type must be the same. That gives us all the information we need. `x` must be a sub type of `Int`, `x.id(x)` has the same type as `x`, thus the method type of `idd` is $< \mathrm{Z}_1 \lhd \mathrm{Int}<> > [\mathrm{Z}_1] \to \mathrm{Z}_1$. Next we process the class `Pair`. The first method `setfst` is the same as in the previous example and thus has the same type, the

only difference is, that the generic type parameters of `Pair` are bound by `Int` rather than `Object`. This changes the upper bound in the method type. Hence the method type of `setfst` is $< Z_2 \lhd \text{Int}<> > [Z_2] \to \text{Pair}<Z_2,\ Y>$. The last method `setboth` contains every possible expression there is: Variables, a field lookup, method calls, object creation and a cast. The overall return type is `Pair`, the first argument of `Pair` is a field lookup, thus its type depends on the object it is looked up on. This is a method call of `setfst` for what we know the type already. Thus we know the argument to `setfst newfst` must be a subtype of `Int`. The return type of `setfst` is a `Pair` where the first field has the same type as the argument `newfst` and the second field has type `Y`. Thus, a field lookup `fst` on that `Pair` has the same type as `newfst`. Likewise the type of the second argument can be inferred. However, the most outer expression of the second field is a cast to `int`, thus the type must be `int`. The expression that is casted may introduce constraints that lead to conflicts, thus it is important to check it and not just stop at casts. In this case all constraints that are introduce by this expression can be solved. All this information leads to the following method type: $< Z_3 \lhd \text{Int}<>,\ Z_4 \lhd \text{Int}<> > [Z_3,\ Z_4] \to \text{Pair}<Z_3,\ \text{Int}<>>$.

8 Abstract Syntax Tree and Parser

8.1 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 class definitions 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 [2].

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

```
[...]

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 its definition. Second, `expression` an expression to be evaluated.

The implementation of types, class definitions, method definitions and expressions are straight forward adaptations of their formal definitions in the paper.

Here an example for the class definition `ClassDef`:

[2] FGJ_AST.py

```
@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 implementation.

The implementations 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 which then all the different kinds of types or expressions inherit. As result 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.

8.2 Parsing

Parsing the program code 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 parts 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. Second, for every rule in the grammar a function is defined which gets the parsed rule as an argument and returns a instantiation of the respective class of the AST. The most outer rule is `program` which 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_elements_of_variable_rule):
    (name, ) = tuple_of_elements_of_variable_rule
    return Variable(name)
```

9 Discussion and Conclusion

I have implemented the type inference algorithm for Featherweight Generic Java as describe in the paper[1] given by Stadelmeier et al. The implementation includes a parser, a constraint generator and a constraint solver as well as an algorithm to combine them. The function `Type` that generates the constraints, is implemented and behaves as it is described in the paper. The function `Unify` that solves the constraints may produce more than one solution and thus is implemented as a generator that yields one solution after another. From outside `Unify` behaves as it is describe in the paper. However, the implementation slightly differs. Especially handling type variables as upper bounds. In the original definition the different solutions are not ordered in any way and even are nondeterministic. In this implementation an ordering is introduced, ordering the solutions from most to least specific.

To combine both of those functions the paper defines the function `TypeInference`. However this function only combines those functions for one class.

In order to make the algorithm usable for more than one class I changed the function `TypeInference` to not only considers one class but a whole program. However, this is not complete. It implements a depth first algorithm through the possible solutions. But the first solution found may not be a correct one. This can be solved by adding a type checker. If the solution does type check it is a correct one.

Both papers "Global Type Inference for Featherweight Generic Java"[1] and "Featherweight Java: A Minimal Core Calculus for Java and GJ"[2] define their own typing rules for a Featherweight Generic Java Program which I both implemented[3]. Combining either one with the type inference algorithm imposes different problems which remain to be solved.

The code[4] of this implementation can be found online. It comes with a command line tool to run the algorithm on any file that contains a Featherweight Generic Program.

A good next step would be to combine the algorithm with one of the type checkers.

[3] The rules of "Global Type Inference for Featherweight Generic Java" are defined in FGJ_typing_1
The rules of "Featherweight Java: A Minimal Core Calculus for Java and GJ" are defined in FGJ_typing_2

[4] `https://github.com/Proglang-Uni-Freiburg/FGJ-inference`

References

1. tadelmeier, Andreas and Plümicke, Martin and Thiemann, Peter. 2022. Global Type Inference for Featherweight Generic Java - Prototype Implementation (Artifact) Schloss Dagstuhl – Leibniz-Zentrum für Informatik 18:1–18:4 https://drops.dagstuhl.de/opus/volltexte/2022/16216
2. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

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, 03.10.2023

Place, Date

Signature