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. However, to be forced to explicitly specify every type signature can be tiring or determining the most general types can even be difficult. By making type systems able to infer types by themselves these problems become obsolete. We describe and implement a global type inference algorithm for Featherweight Generic Java: a minimal core calculus for Java (Featherweight Java) extended with generics. The type inference algorithm is able to infer all method types for a program with specified class headers and field types which does not use polymorphic recursion. This type inference algorithm was developed by Andreas Stadelmeier, Martin Plümicke and Peter Thiemann in the paper: "Global Type Inference for Featherweight Generic Java"

Abstract. (german) Typsysteme sind mächtig und ein wichtiger Bestandteil einer jeden mordernen Programmiersprache. 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 `_+_` without any type annotations. This function works for all types that support the addition operator for instance `Int`, `Double` or `Float`. Annotating the function `_+_` 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 that almost all features are dropped, even assignments. Due to that Featherweight Java a functional subset of Java. Featherweight Java only contains classes with fields, methods and inheritance as well as 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 as expressive as 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:

```java
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. Firstly, one or more class definitions and secondly, 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 type-check 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 has type `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 time is not possible. Even in case the expression of the field lookup is an object creation of `Pair`, the type of the expression is `Object`. Consequently, it results in a compile time error.

This problem can be solved by giving fields a type parameter `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 longer. 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
```

$$T ::= X \mid N$$

$$N ::= C < \overline{T} >$$

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

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

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

Fig. 1. Syntax

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 formally as it is described in the paper "Global Type Inference for Featherweight Generic Java"[1] by Stadelmeier et al. Afterwards, it is shown how the Global Type Inference Algorithm is implemented. There are a few changes to bring the formalisation closer to the implementation.

2   Featherweight Generic Java

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

Types (`T`, `U`, `V`) in Featherweight Generic Java are either type parameters (`X`, `Y`, `Z`) or 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, which are used as a field type in the class definition, are declared within `<>`. The symbol $\lhd$ is used as a shortcut for the keyword `extends`. This is followed by the name of the superclass indicated by another `extends` in front of it. Inside the body of a class definition the fields $\overline{f}$ of the class are defined with their types in front of them, followed by the optional constructor `K` and all method definitions $\overline{M}$.

2.3   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 its name M, its arguments $\bar{x}$ 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 along its arguments or a cast. A cast binds less than any other expression. Method calls and object creation do not need any instantiation of their generic types.

2.5   Type Annotations

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

2.6   Constructors

In Featherweight Java as well as in Featherweight Generic Java a constructor is always required. The constructor has always the same form.

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

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 $\bar{g}$ for fields that are not defined in the current class are passed on. Second, every other argument $\bar{f}$ 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 is dropped in this implementation.

3   Auxiliary Functions

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

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

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

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

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

Fig. 2. Subtyping

3.1 Substitution

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

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

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

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 $[\text{T}_1/\text{X}_1, \ ... \ , \ \text{T}_n/\text{X}_n]\text{N}$.
The third rule (3) is short for $[\text{T}_1/\text{X}_1, \ ... \ , \ \text{T}_n/\text{X}_n]\text{N}_1, \ ... \ , \ [\text{T}_1/\text{X}_1, \ ... \ , \ \text{T}_n/\text{X}_n]\text{N}_n$. These rules are implemented in different functions by case splitting on the different types in N.

3.2 Substitution on Constraints

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

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

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

3.3 Subtyping

Figure 2 shows the subtyping rules. By definition, a type is always a subtype of itself and subtyping is transitive. The environment Δ maps every type parameter

to its upper bound. Moreover, 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

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

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

A field lookup on a class `C` returns every field of `C` with its type, including all fields of all superclasses of `C` along their types. `Object` itself has no fields and 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

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

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

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 inferred from 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

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

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

genericSupertype takes two subtype related classes C and D and a list of types $\overline{T}$ as arguments. If the two classes are the same, the list of types $\overline{T}$ is returned. Otherwise, the function is recursively called with the superclass C' of C. However, the instantiation of the type parameters of the superclass C' in the class definition of C can be different than the generic type parameters $\overline{T}$. Thus, $\overline{M}$ is taken as the second argument instead of $\overline{T}$. Now, the type parameters $\overline{Y}$ may occur in $\overline{M}$. However, $\overline{T}$ are exactly the instantiated types for $\overline{Y}$. Thus, $\overline{T}$ is substituted for $\overline{Y}$ in $\overline{M}$.

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.

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 FJType and constraint solving Unify. When solving the constraints leads to multiple solutions for a single class, simply one solution is assumed. If the algorithm later fails, it backtracks and assumes the next solution.

---

[1] Chapter 4.1 Type inference for a program

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

Fig. 3. Constraint Syntax

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

5 FJType

Figure 4 defines the function `FJType`. This function 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 present. The function returns a tuple consisting of a set of method signatures λ with new type variables and a constraint set C which constrains those type variables.

In Featherweight Generic Java methods can only be overridden when they have the same argument types. However, the return type can be a subtype of the return type of the overridden method. If a method is overridden 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

$\text{FJType}(\Pi, \text{ class } C{<}\overline{X} \lhd \overline{N}{>} \lhd N\{\overline{T} \; \overline{f}; \; \overline{M}\}) =$

    let $\overline{a}_m$ be fresh type variables for each $m \in \overline{M}$

      $\overline{\lambda}_0 = \{C{<}\overline{X} \lhd \overline{N}{>}.m : \; <\overline{Y} \lhd \overline{P}>\overline{T} \to a_m$

              $\mid m \in \overline{M}, \; \text{mtype}(m, \; N, \; \Pi) = <\overline{Y} \lhd \overline{P}>\overline{T} \to T\}$

      $C_0 = \{a_m < T \mid m \in \overline{M}, \; \text{mtype}(m, \; N, \; \Pi) = <\overline{Y} \lhd \overline{P}>\overline{T} \to T\}$

      $\overline{\lambda}' = \{(C{<}\overline{X} \lhd \overline{N}{>}.m : \; \overline{a} \to a_m)$

              $\mid m \in \overline{M}, \; \text{mtype}(m, \; N, \; \Pi) \text{ not defined}, \; \overline{a} \text{ fresh}\}$

      $C_m = \{\{a_m < \text{Object}, \overline{a} < \overline{\text{Object}}\} \mid (C{<}\overline{X} \lhd \overline{N}{>}.m : \; \overline{a} \; \to a_m) \in \overline{\lambda}'\}$

      $\Pi = \Pi \cup \overline{\lambda}' \cup \overline{\lambda}_0$

    in $(\Pi, C_0 \cup C_m \cup \bigcup_{m \in \overline{m}} \text{TYPEMethod}(\Pi, C{<}\overline{X}{>}, m))$

Fig. 4. FJType

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.

5.1 Method Type

$\text{TYPEMethod}(\Pi, \; C{<}\overline{X}{>}, \; m(\overline{x})\{ \text{ return } e; \}) =$

    let $<\overline{Y} \lhd \overline{P}>\overline{T} \to T = \Pi(C{<}\overline{X} \lhd \overline{N}{>}.m)$

      $(R, C) = \text{TYPEExpr}((\Pi; \{this : C{<}\overline{X}{>}\} \cup \{\overline{x} : \overline{T}\}), e)$

    in $\; C \cup \{R < T\}$

`TYPEMethod` 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 `TYPEMethod` on the expression `e` inside the method's body. In this call a type environment for variables is created. Where `this` has the type $C{<}\overline{T}{>}$ and the types of the method's arguments are inferred from the type signature which was brought into the scope earlier. `TYPEMethod` 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 `TYPEMethod`. 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

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

In case of a field lookup `e.f` first the `TYPEMethod` function is called recursively on the expression `e`, returning a type R and a constraint set C_R. Now, the difficult part is to deduce 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 constrains 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 $\overline{a}$. Another one that constrains the resulting type of the field lookup to be equal to the type of the corresponding field in `C`. And finally, one that constrains every a_i of $\overline{a}$ to be a subtype of the bounding type of the corresponding type parameter it replaces.

5.5   Method Invocation

$\text{TYPEExpr}((\Pi; \overline{\eta}), \text{e.m}(\overline{\text{e}})) =$

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

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, TYPEMethod 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 constraints: 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 restricts the type of the arguments for M to be subtypes of the argument types imposed by the method signature. The second constraint restricts 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 for method signatures of method M are put together to one big or-constraint which then is returned.

5.6   Object Creation

$\text{TYPEExpr}((\Pi; \overline{\eta}), \text{new C}(\overline{\text{e}})) =$

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

The constraints generated for an object creation are rather simple. TYPEMethod 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

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

For a cast the `TYPEMethod` 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 `Unify` solves a constraint set. Solving the constraints may lead to no, one or multiple 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, it 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 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 circumvent 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 5 are applied exhaustively to C'.

For `match` and `adopt` the implementation also includes the dual rules. 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 a 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 vice 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. This upper bound is a type variable a 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:

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

Such an upper bound constraint can also be implied by two or multiple other constraints of the form $a < C\langle\overline{T}\rangle$ and $a <^* b$. b can either be a subtype of $C\langle\overline{T}\rangle$ or it can be an upper bound to $C\langle\overline{T}\rangle$. In the first case everything is fine. The latter case implies the constraint $C\langle\overline{T}\rangle < b$. This constraint can be handled the same way as a direct upper bound constraint. Only the possibility of b being a subtype of $C\langle\overline{T}\rangle$ needs to be taken into account. Thus, the constraint $b < C\langle\overline{T}\rangle$ is included as an additional possibility in 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''.

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

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

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

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

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

$$\frac{C \cup \{D{<}\overline{T}{>} = D{<}\overline{U}{>}\}}{C \cup \{\overline{T} = \overline{U}\}} \qquad \text{reduce}$$

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

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

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

Fig. 5. resolve rules

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

Step 4 If the constraint set C" has changed after applying 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 carry any information with them. Thus, these constraints can be eliminated. This is done by exhaustively applying the rules sub-elimination and erase.

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

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

Step 6 The constraint set C is divided into $C_<$ and $C_=$. $C_<$ 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 deduced as follows:

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

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

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 frozen lists, frozen sets and frozen dictionaries. As frozen sets come with the default library they can be used without difficulty. In order to have frozen lists and frozen dictionaries the python libraries `FrozenList` and `frozendict` are used.

One important environment that is passed to almost every function, but is 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 changes afterwards. The class table environment simplifies the access to any class at any time by their name.

7.1   TypeInference

The function `TypeInference` connects constraint generation with constraint solving. The implementation of differs from the idea described 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. Firstly, `FJType` is called. The current class and the method type environment containing all the method type assumptions of earlier classes are given as arguments to `FJType`. 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 parameters of method types in the method type environment, that are overridden in the current class definition. The upper bounds of the field types can be directly deduced from the class definition. Furthermore, the upper bounds of the type variables occurring in method types can be inferred from the method type assumption that is returned from the call to `FJType`. Together both 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 looping over them.

A solution consists of two parts. Firstly, a mapping from every type variable occurring in the constraints to a type parameter or a non variable type. Secondly, a mapping from every new type parameter introduced by `Unify` to its upper bound. For every method type assumption created by `FJType` 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. Collecting all type parameters that occur in the method type assumption of a method 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, the program could be type checked. If this check fails, the algorithm continues with 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. This pointer is 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 terminates 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 in the form of 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 `Unify` are calculated at once, but just when they are needed. To do this the whole function `Unify` 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`. Firstly, 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. The function `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 must also be defined for them and every possible combination. In order to avoid this, generic type parameters are treated as parameterless classes with their upper bounds being now their superclasses. One important aspect here is that those new classes are not added to the class table. Instead they are stored in an extra environment. Now, when looking up classes in the class table, an extra check must be done to guaranty that the class is in it. If not, the superclass can be deduced from 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 sides of each constraint. In case of a non type variable, the transformations are applied recursively.

Exhaustively applying rules There is no pleasant 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, the condition variable is set to `True`. Some rules take two constraints as precondition, thus every possible combination of constraints must be tested. This is done by two nested for-loops. Applying a rule changes the constraint set and thus imposing a problem. Changing a set while iterating over it is not possible. Thus, the algorithm iterates over a copy of that constraint set. As soon as one rule applies, a constraint may be removed from the constraint set. However, that does not effect the copy. It results in another problem: Constraints that were removed are not allowed to be used again. A simple but not efficient way to solve this problem is to restart the for-loop every time a rule applies.

Rules in Figure 5 Rules have different amounts of constraints as preconditions. For rules that have more than one precondition every possible combination of constraints must be considered. Rules that take arbitrary many subtype 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`) subtype constraints on type variables. Then, the two type variables that occur in the constraints (either in the subtype constraint on type variables or in the two other constraints that are needed for `adopt`) are checked, if they are in any case subtype related. This leads to a maximum of two preconditions, thus two nested for-loops over the constraint set are needed. The decision of which rule to apply is made by case splitting the subtype 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 (subtype constraint with upper bound `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 is per definition 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 case an upper bound constraint already exists it needs to be checked whether the lower bound is a subtype of the upper bound. If not, this constraint set has no solution. Checking if it is a subtype 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. Next, 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.

Because 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 anything. 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 can still be implied by other constraints of the form $a < C{<}T{>}$ and $a <^* b$. The type variable can be a subtype or a supertype of $C{<}\overline{T}{>}$. If it is a supertype, it can be handled by creating an or-constraint as before. However, the possibility of b being a subtype of $C{<}\overline{T}{>}$ needs to be considered, too. In order to do so the subtype constraint $b < C{<}T{>}$ is added 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:

```
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. Firstly, 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, $(\overline{a} \to a)$ is equivalent to $([a_1, a_2, ...] \to a_0)$.

FJType The function `FJType` takes two arguments. Firstly, it takes the method type environment. Because `Pair` is the first class to be processed, this environment is empty. Secondly, it takes 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:

$$\lambda = \{(\text{Pair}{<}X \lhd \text{Object}{>}.\text{setfst}) : [a_1] \to a_0\}$$
$$C = \{a_0 < \text{Object}, \ a_1 < \text{Object}\}$$

Nextly, for every method defined in `Pair` the function `TYPEMethod` with the new type environment is called. In this case the method `setfst` is only called once. `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 is always one expression. Here it is an object creation of `Pair`. At 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`. 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, the constraint $a_1 < a_5$ is generated. 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 to represent 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:

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

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:

$$\lambda = \{(\text{Pair}{<}\text{X} \lhd \text{Object}{>}.\text{setfst}) :\ [a_1] \rightarrow a_0\}$$
$$C = \{a_0 < \text{Object},\ a_1 < \text{Object},$$
$$a_1 < a_5,\ a_5 < \text{Object},\ a_2 < a_6,\ a_6 < \text{Object},$$
$$\{\{\text{Pair}{<}\text{X, Y}{>} < \text{Pair}{<}a_3, a_4{>},\ a_2 = a_4,\ a_3 < \text{Object},\ a_4 < \text{Object}\}\},$$
$$\text{Pair}{<}a_5, a_6{>} < a_0$$
$$\}$$

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 to one simple constraint set. Before we go on we need to transform all type parameters to parameterless classes ($X \rightarrow X{<}{>}$). Then, we continue with step 1. The only rule that applies to anything is `adapt`.

$$\frac{C \cup \{\text{Pair}{<}\text{X}{<}{>}, \text{Y}{<}{>}{>} < \text{Pair}{<}a_3, a_4{>}\}}{C \cup \{\text{Pair}{<}\text{X}{<}{>}, \text{Y}{<}{>}{>} = \text{Pair}{<}a_3, a_4{>}\}} \quad \text{adapt}$$

Now, we can resolve this further with `reduce`:

$$\frac{C \cup \{\text{Pair}{<}\text{X}{<}{>}, \text{Y}{<}{>}{>} = \text{Pair}{<}a_3, a_4{>}\}}{C \cup \{\text{X}{<}{>} = a_3, \ \text{Y}{<}{>} = a_4\}} \quad \text{reduce}$$

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

$$\frac{C \cup \{\text{X}{<}{>} = a_3\}}{C \cup \{a_3 = \text{X}{<}{>}\}} \quad \text{and} \quad \frac{C \cup \{\text{Y}{<}{>} = a_3\}}{C \cup \{a_3 = \text{Y}{<}{>}\}} \quad \text{swap}$$

No other rules apply now anymore and we continue with step 2.

We find the constraint $\text{Pair}{<}a_5, a_6{>} < a_0$ 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 $a_0 < \text{Object}{<}{>}$. Next, we check if $\text{Pair}{<}a_5, a_6{>}$ 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 $\text{Pair}{<}a_5, a_6{>}$ 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. 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: $a_2 = a_4$, $a_3 = \text{X}{<}{>}$ and $a_4 = \text{Y}{<}{>}$. These substitutions change our constraint set and we start again from step 1 with the following constraint set:

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

With `adapt` and `reduce` we can drop both $\text{X}{<}{>} < \text{Object}{<}{>}$ and $\text{Y}{<}{>} < \text{Object}{<}{>}$. Then, we continue with step 2. Once again we find a constraint that has a type variable as upper bound. This time $\text{Y}{<}{>} < a_6$ the corresponding upper bound is $a_6 < \text{Object}{<}{>}$. Resolving this with `expandLB` gives us again two constraint sets: one with $a_6 = \text{Y}{<}{>}$ and one with $a_6 = \text{Object}{<}{>}$. We consider the more specific one first and continue with step 3. We substitute $\text{Y}{<}{>}$ 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 ($\text{X}{<}{>} \rightarrow \text{X}$). Now, we can start with step 6.

We first divide the constraint set into subtype and equal constraints resulting in the following:

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

For the subtype constraint we introduce a new type parameter `Z`. Its upper bound is the upper bound of that subtype 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 $[a_1] \to a_0$. If we substitute now, we get the type signature $[Z] \to \text{Pair}{<}Z,\ Y{>}$. The last thing to do is to add the upper bounds to all new generic types so we obtain the full type signature:

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

A more advanced example is shown in Figure 6.

We process one class after another: The first class does not bring any new features that we have not seen in the first example. The inferred method type of `id` is $< Z_0 \lhd \text{Object}{<}{>} > [Z_0] \to Z_0$. 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: Firstly, `X` has to be of type `Int`, because `Int` is the only class that defines the method `id`. Secondly, 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` has to be a subtype of `Int`. `x.id(x)` has the same type as `X`, thus the method type of `idd` is $< Z_1 \lhd \text{Int}{<}{>} > [Z_1] \to Z_1$. 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 which we know the type already. Thus, we know the argument `newfst` of the method call `setfst` 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

```
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. 6. Example 2

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 the expression that is casted and not just stop there. 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 traversed is desirable. Such a form is called an Abstract Syntax Tree short AST. Every node in this AST is represented by a dataclass in Python [2].

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

```
[...]

ClassTable = dict[str, ClassDef]


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

The class `Program` has two fields: `CT` and `expression`. CT, short for class table, is a mapping from every class name to its definition. `expression` is 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`:

```
@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.

---

[2] FGJ_AST.py

The implementations for types and expressions differs slightly. Because there are different types and different expressions for both, first a plain class with no fields is defined: `Type` and `Expressions`. All the different kinds of types inherit from `Type` and all kinds of expressions from `expression`. Thereby simulating the structure of algebraic data types. As a result it is possible to match against types and expressions.

```python
@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 performed by using the python parser library Lark. Lark is a powerful library that can parse any context-free grammar.

There are two parts when using Lark: firstly, the definition of the grammar. Because every Featherweight Generic Java program can be represented as an AST, the grammar rules are trivial. First, a rule for identifiers is created. This representation allows to easily change what is allowed to be an identifier at any time later. Then, for every class defined for the AST a rule is defined recursively. Second, for every rule in the grammar a function is defined. These functions get the parsed rule as an argument and return an instantiation of the respective class for the AST. The most outer rule is `Program` which then returns the whole program represented by the AST, starting with the class `Program`.

```python
# 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 described 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 the outside `Unify` behaves as it is described in the paper. However, the implementation slightly differs. Especially the handling of 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 useable for more than one class I changed the function `TypeInference` to not only consider 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 type checks, it is the correct one. If not, the algorithm backtracks.

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