An Implementation of Type Inference for Featherweight Generic Java

Timpe Hörig

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

Bachelor Thesis

Examiner: Prof. Dr. Peter Thiemann

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

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

Table of Contents

1 Introduction

1.1 Type Systems

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

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

Calling this function with an `Integer`, for example 1, returns 2. However, calling it with a `String` results in a type error, because addition between a `String` and an `Integer` is not defined. Errors such as that are easy to make but cause the program to crash at runtime. With type systems it is possible to detect such errors at compile time and 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 an instantiation of a variable. An other down sight 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`. Annotation this `plus` function with one of those types would mean its 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 the programming languages is defined that contains less features but behaves similar. Extending those smaller versions with new features leads to languages still small enough so providing proofs not only becomes doable but handy. Featherweight Java is exactly that. A smaller version of Java. In fact it is so small almost all features are dropped even assignments. Making Featherweight Java a functional subset of Java. Featherweight Java only contains classes with fields, methods and inheritance and five different forms of expressions: Variables, field lookups, method invocations, object creation and castings. As Java itself is not a functional programming languages it is easy to see that Featherweight Java is not equivalent to Java but because of its similar behavior it is still possible to draw back conclusions to the full Java.

A typical Featherweight Java program may look like this:

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

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

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

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

As well as Java, Featherweight Java includes a type system. With the type annotations given in the class definitions Featherweight Java is able to typecheck the given expression. In the example above the most inner expression that has a fixed type is the object creation of `new Pair(new Pair(new Object(), new Object()), new Object())`. 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` the second time would be possible because the expression it is called on is an object creation of `Pair` but because its type is `Object` this leads to a compile time error.

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

introduced both with the upper bound `Object`. That means both type variables can range over any types that are a subtype of `Object`.

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

Rewriting the example above results in:

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

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

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

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

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

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

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

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

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

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

$$T := X \mid N$$

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

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

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

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

Fig. 1. Syntax

2 Featherweight Generic Java

2.1 Types

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

2.2 Class Definitions

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

2.3 Methoddefinition

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` wich is returned.

2.4 Expressions

An expression `e` can be in five different forms. First, a simple variable `x`. Second, a field lookup written `e.f`. Third, a method invocation that takes one expression for each argument. Fourth, an object creation indicated by the keyword `new`

followed by the name of the new Object and its arguments. Fifth, a cast. A cast binds less than any other expression. Notice how methodcalls and object creation do not need any instanciation of their generic type annotations.

2.5 Typeannotation

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

2.6 Constructor

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

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

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

3 Abstract Syntax Tree and Parser

3.1 Abstract Syntax Tree

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

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

```
[...]

ClassTable = dict[str, ClassDef]


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

[1] FGJ_AST.py

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

For example the implementation for classdefinition `ClassDef`:

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

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

The implemenatations for types and expressions differs a bit. Because there are different types and different expressions, for both first a plain class with no fields is defined `Type` and `Expressions` from 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.

3.2 Parsing

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

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

```
# grammar rules

identifier: ...

variable: identifier

[...]

# function for shaping

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

4 Auxiliary Functions

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

4.1 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 standart substitution rule. In a non variable type N every variable type X is replaced by the given type T.
The second rule (2) is short for `[T_1/X_1, ... , T_n/X_n]N`.
The third rule (3) is short for `[T_1/X_1, ... , T_n/X_n]N_1, ... , [T_1/X_1, ... , T_n/X_n]N_n`.
These rules are implemented in different functions using pattern matching on the different type types.

4.2 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).

4.3 Subtyping

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

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

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

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

Fig. 2. Subtyping

By definition a type is always a subtype of itself. Subtyping is also transitive and the environment delta Δ maps every type variable to its upper bound.

4.4 Field Lookup

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

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

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

4.5 Method Type Lookup

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 of the method type environment. Once again generic type variables 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 generic type variables are substituted by the types the class `C` is instantiated with.

4.6 Generic Supertype

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

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

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, then 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 variables than C. That is why $\overline{\text{T}}$ is replaced by $\overline{\text{M}}$ where $\overline{\text{M}}$ are the generic type variables of C', but the generic type variables $\overline{\text{Y}}$ may occur in $\overline{\text{M}}$, thus $\overline{\text{T}}$ is substituted for $\overline{\text{Y}}$ in $\overline{\text{M}}$.

5 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 algorithm as a program where this is not the case can be transformed to a program where it is. This transformation is shown in the paper [...].
In the following it is assumed that a program always fulfills this condition.

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

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

6 FJType

`FJType` is the function that generates constraints for a class. It takes the definition of that class and the method type environment where all the method

signatures that have already been inferred from earlier classes are in. It returns a tuple consisting of a set of method signatures 'lambda' with new type variables and a constraint set C which constraints those type variables.

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

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

6.1 Method Type

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

`MethodType` takes the method type environment, the class header `C<\overline{X}>` of the class that is currently checked and the method definitions as arguments. It first brings the type signature of the current method in scope and then

calls `TypeExpression` on the expression `e` inside the body of the method. In this call also a type environment for variables is created, where `this` has the type C<X'> and the arguments of the method are read from the type signature brought into the scope earlier. `TypeExpression` returns a tuple of the type `R` of `e` and the generated constraints. The generated constraints together with the constraint that `R` must be a subtype of the return type of the method signature is returned.

6.2 Type Expressions

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

6.3 Variable

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

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

6.4 Field Lookup

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

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

6.5 Method Invocation

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

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

6.6 Object Creation

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

6.7 Cast

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

7 Unify

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

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

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

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

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

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

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

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

$$\frac{anders}{herum}$$

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

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

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

$$\frac{C \cup \{a = a\}}{C}$$

$$\frac{C \cup \{N = a\}}{C \cup \{a = N\}}$$

Fig. 3. swap

For `match` and `adopt` here also the dual rule is defined. This makes it later easier to detect illegal upper bounds because then only the highest type can have an illegal upper bound which can easily be detected.

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

In the last case the non variable type C<$\overline{\text{T}}$> has an upper bound which is not a non variable type but a type variable which is not allowed in Featherweight Generic Java. This is solved by first searching the upper bound constraint of `a` if no such constraint exists `Object` is chosen as the upper bound. Then for every possible class from C<$\overline{\text{T}}$> 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:

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

In the paper ... a second case is described where such an upper bound can be implied. This case cannot happen in this implementation because of the added reverse rules to step 1.
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".

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

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 constraint can be eliminated. This is done by exhaustively applying the rules sub-elimination and erase.

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

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

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 variables $\overline{\text{Y}}$ are generated, one for each constraint in $C_<$. Now the substitution σ and the mapping from $\overline{\text{Y}}$ to their upper bounds can be read of as follows:

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

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

The pair of σ and γ is returned.

8 Implementation

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.

8.1 FJType

The implementation of the constraint generation is straight forward. Because every functions is described in pseudo code most of the functions can easily be translated into python code.
Creating fresh type variables is simply done by 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.

8.2 FJUnify

Resolving or-constraints Converting a constraint set `C` which contains simple-constraint as well as or-constraints equals a Cartesian product with n sets, where n is the number of or-constraints in `C`. But 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 that yields one solution after another.

Treating type variables as parameterless classes All rules of step 1 and following, would also apply for generic type variables. So every rule also must be defined for them, not just tripling the code but also imposing some nasty edge cases. In order to solve this generic type variables 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 variables to parameterless classes and backwards is done in two extra functions that take and return a constraint set.

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. That alone may not seem too bad but the combination of two nested for-loops to go over every possible combination of constraints, while changing the constraint set the loop is going through results in some nasty code that is not the most efficient.

Rules in step 1 There are two kind of rule. Some that take one and some that take two constraints as pre-condition. This forces two for-loops to go over every possible combination. Deciding which rule to apply is done by pattern matching against sub- or equal-constraints and on the types of the constraint elements. Because some patterns are ambiguous it is important to take the order into account.

ExpandLB The implementation of expandLB differs from the idea the original paper imposes. Having constraints such that (a $<$ C$<\overline{\text{T}}>$) and (a $<$* b) so it is possible that (C$<\overline{\text{T}}>$ $<$ b) is implied is not possible. because the dual rules of DOES IT?

8.3 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);
```

```
        }
    }

    new Object()
```

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 introduced. 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 also be different. This is simply done to showcase the order of which the type variables are introduced.

FJType `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 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:

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

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:

$$
\begin{aligned}
\mathrm{C} = \{&\mathrm{a}_0 < \mathrm{Object},\ \mathrm{a}_1 < \mathrm{Object},\\
&\mathrm{a}_1 < \mathrm{a}_5,\ \mathrm{a}_5 < \mathrm{Object},\ \mathrm{a}_2 < \mathrm{a}_6,\ \mathrm{a}_6 < \mathrm{Object},\\
&((\mathrm{Pair}\langle \mathrm{X},\ \mathrm{Y}\rangle < \mathrm{Pair}\langle \mathrm{a}_3, \mathrm{a}_4\rangle,\ \mathrm{a}_2 = \mathrm{a}_4,\ \mathrm{a}_3 < \mathrm{Object},\ \mathrm{a}_4 < \mathrm{Object}))\\
&\}
\end{aligned}
$$

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`>` $< a_0$. Now we have the full method type environment and the full constraint set:

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

9 Discussion and Conclusion

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