Global Type Inference for Featherweight Generic Java

Andreas Stadelmeier

Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

Martin Plümicke

Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb, Germany

Peter Thiemann

Institut für Informatik, Universität Freiburg, Germany

Abstract

Java's type system mostly relies on type checking augmented with local type inference to improve programmer convenience.

We study global type inference for Featherweight Generic Java (FGJ), a functional Java core language. Given generic class headers and field specifications, our inference algorithm infers all method types if classes do not make use of polymorphic recursion. The algorithm is constraint-based and improves on prior work in several respects. Despite the restricted setting, global type inference for FGJ is NP-complete.

2012 ACM Subject Classification Software and its engineering → Language features

Keywords and phrases type inference, Java, subtyping, generics

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.28

Related Version *Full Version*: `https://arxiv.org/abs/2205.08768` [28]

Supplementary Material Source code for the accompaning prototype implementation
Software (Source Code): `https://github.com/JanUlrich/FeatherweightTypeInference`
archived at `swh:1:dir:dbbfa3ad5db1b423c098c392af1d4b720db93e0e`
Software (ECOOP 2022 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.8.2.18`

1 Introduction

Java is one of the most important programming languages. In 2019, Java was the second most popular language according to a study based on GitHub data.[1] Estimates for the number of Java programmers range between 7.6 and 9 million.[2] Java has been around since 1995 and progressed through 16 versions.

Swarms of programmers have taken their first steps in Java. Many more have been introduced to object-oriented programming through Java, as it is among the first mainstream languages supporting object-orientation. Java is a class-based language with static single inheritance among classes, hence it has nominal types with a specified subtyping hierarchy. Besides classes there are interfaces to characterize common traits independent of the inheritance hierarchy. Since version J2SE 5.0, the Java language supports F-bounded polymorphism in the form of generics.

[1] `https://www.businessinsider.de/international/the-10-most-popular-programming-languages-according-to-github-2018-10/`

[2] `https://www.zdnet.com/article/programming-languages-python-developers-now-outnumber-java-ones/`, `http://infomory.com/numbers/number-of-java-developers/`

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 28; pp. 28:1–28:27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```java
class Pair<X,Y> {
  X fst;
  Y snd;
  Pair<X,Y>(X fst, Y snd) {
    this.fst=fst;
    this.snd=snd;
  }
  <Z> Pair<Z,Y> setfst(Z fst) {
    return new Pair(fst, this.snd);
  }
  Pair<Y,X> swap() {
    return new Pair(this.snd,
        this.fst);
  }
}
```

(a) Featherweight Generic Java (FGJ).

```java
class Pair<X,Y> {
  X fst;
  Y snd;
  Pair(fst, snd) {
    this.fst=fst;
    this.snd=snd;
  }
  setfst(fst) {
    return new Pair(fst, this.snd);
  }
  swap() {
    return new Pair(this.snd,
        this.fst);
  }
}
```

(b) FGJ with global type inference (FGJ-GT).

Figure 1 Example code.

Java is generally explicitly typed with some amendments introduced in recent versions. That is, variables, fields, method parameters, and method returns must be adorned with their type. Figure 1a contains a simple example with generics.

While the overhead of explicit types look reasonable in the example, realistic programs often contain variable initializations like the following:[3]

```java
  HashMap<String, HashMap<String, Object>> outerMap =
    new HashMap<String, HashMap<String, Object>>();
```

Java's *local variable type inference* (since version 10[4]) deals satisfactorily with examples like the initialization of `outerMap`. In many initialization scenarios for local variables, Java infers their type if it is obvious from the context. In the example, we can write

```java
var outerMap = new HashMap<String, HashMap<String, Object>>();
```

because the constructor of the map spells out the type in full. More specifically, "obvious" means that the right side of the initialization is

- a constant of known type (e.g., a string),
- a constructor call, or
- a method call (the return type is known from the method signature).

The `var` declaration can also be used for an iteration variable where the type can be obtained from the elements of the container or from the initializer. Alternatively, if the variable is used as the method's return value, its type can be obtained from the current method's signature.

However, there are still many places where the programmer must provide types. In particular, an explicit type must be given for

- a field of a class,
- a local variable without initializer or initialized to `NULL`,
- a method parameter, or
- a method return type.

[3] Taken from `https://stackoverflow.com/questions/4120216/map-of-maps-how-to-keep-the-inner-maps-as-maps/4120268`.
[4] `https://openjdk.java.net/jeps/286`

```
<T> boolean eqPair (Pair<T,T> p){
  return p.fst.equals<T>(p.snd);
}
```

```
eqPair (p) {
  return p.fst.equals(p.snd);
}
```

Figure 2 `eqPair` in FGJ-GT and FGJ.

In this paper, we study *global type inference* for Java. Our aim is to write code that omits most type annotations, except for class headers and field types. Returning to the `Pair` example, it is sufficient to write the code in Figure 1b and global type inference fills in the rest so that the result is equivalent to Figure 1a. Our motivation to study global type inference is threefold.

- Programmers are relieved from writing down obvious types.
- Programmers may write types that leak implementation details. The `outerMap` example provides a good example of this problem. From a software engineering perspective, it would be better to use a more general abstract type like

  ```
  Map<String, Map<String, Object>> outerMap = ...
  ```

  Global type inference finds most general types.
- Programmers may write types that are more specific than necessary instead of using generic types. Here, type inference helps programmers to find the most general type. Suppose we wanted to add a static method `eqPair` for pairs of integers to the `Pair` class.

  ```
  boolean eqPair (Pair<Integer,Integer> p) {
    return p.fst.equals(p.snd);
  }
  ```

  With global type inference it is sufficient to write the code on the right of Figure 2 and obtain the FGJ code with the most general type on the left.

To make our investigation palatable, we focus on global type inference for Featherweight Generic Java [11] (FGJ), a functional Java core language with full support for generics. Our type inference algorithm applies to FGJ programs that specify the full class header and all field types, but omit all method signatures. Given this input, our algorithm infers a set of most general method signatures (parameter types and return types). Inferred types are generic as much as possible and may contain recursive upper bounds.

The inferred signatures have the following round-trip property (relative completeness). If we start with an FGJ program that does not make use of polymorphic recursion (see Section 2.5), strip all types from method signatures, and run the algorithm on the resulting stripped program, then at least one of the inferred typings is equivalent or more general than the types in the original FGJ program.

Contributions

We specify syntax and type system of the language FGJ-GT, which drops all method type annotations from FGJ and the typing of which rules out polymorphic recursion. This language is amenable to polymorphic type inference and each FGJ-GT program can be completed to an FGJ program (see Section 3).

We characterize uses of polymorphic recursion in FGJ and their impact on signatures of generic methods (Section 3.4).

We define a constraint-based algorithm that performs global type inference for FGJ-GT. This algorithm is sound and relatively complete for FGJ programs without polymorphic recursion (Sections 4 and 5).

In Section 7 we show that global type inference is NP-completene.

Our algorithm improves on previous attempts at type inference for Java in the literature as detailed in Section 8.

A prototype of global type inference is available as an evaluated artifact.

A full version of the paper with all proofs is available [28].

2 Motivation

This section presents a sequence of more and more challenging examples for global type inference (GTI). To spice up our examples somewhat, we assume some predefined utility classes with the following interfaces.

```
class Bool {
  Bool not();
}
class Int {
  Int negate ();
  Int add (Int that);
  Int mult (Int that);
}
class Double {
  Double negate ();
  Double add (Double that);
  Double mult (Double that);
}
```

We generally use upper case single-letter identifiers like $\mathtt{X}, \mathtt{Y}, \ldots$ for type variables. Given a FGJ-GT class Cl_0, we call any FGJ class Cl_i that can be transformed to Cl_0 by erasing type annotations a *completion of* Cl_0.

2.1 Multiplication

Here is the FGJ-GT code for multiplying the components of a pair.[5]

```
class MultPair {
  mult (p) { return p.fst.mult(p.snd); }
}
```

Assuming the parameter typing $\mathtt{p} : \mathtt{P}$, result type $\mathtt{R}$, and that `mult` in the body refers to `Int.mult`, we obtain the following constraints.

- From `p.fst`: $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{X}, \mathtt{Y}\rangle$ and $\mathtt{p.fst} : \mathtt{X}$.
- From `p.snd`: $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{Z}, \mathtt{W}\rangle$ and $\mathtt{p.snd} : \mathtt{W}$.
- The two constraints on `P` imply that $\mathtt{X} \doteq \mathtt{Z}$ and $\mathtt{Y} \doteq \mathtt{W}$.
- From `.mult (p.snd)`: $\mathtt{X} \lessdot \mathtt{Int}$, $\mathtt{Y} \lessdot \mathtt{Int}$, and $\mathtt{Int} \lessdot \mathtt{R}$.

The return type `R` only occurs positively in the constraints, so we can set $\mathtt{R} = \mathtt{Int}$. The argument type `P` only occurs negatively in the constraints, so $\mathtt{P} = \mathtt{Pair}\langle\mathtt{X}, \mathtt{Y}\rangle$. This reasoning gives rise to the following completion.

[5] We indicate FGJ-GT code fragments by using a gray background.

```
class A1 {
  m(x) { return x.add(x); }
}
class B1 extends A1 {
  m(x) { return x; }
}
```

```
class A2 {
  m(x) { return x; }
}
class B2 extends A2 {
  m(x) { return x.add(x); }
}
```

Figure 3 Method overriding.

```
class MultPair {
  <X extends Int, Y extends Int>
  Int mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

We obtain a second completion if we assume that `mult` refers to `Double.mult`.

```
class MultPair {
  <X extends Double, Y extends Double>
  Double mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

Finally, the definition of `mult` might be recursive, which generates different constraints for the method invocation of `mult`.

- From `.mult (p.snd)`: $\mathtt{X} \lessdot \mathtt{MultPair}$, $\mathtt{Y} \lessdot \mathtt{P}$, and $\mathtt{R} \lessdot \mathtt{R}$.

Transitivity of subtyping applied to $\mathtt{Y} \lessdot \mathtt{P}$ and $\mathtt{P} \lessdot \mathtt{Pair}\langle\mathtt{X},\mathtt{Y}\rangle$ yields the constraint $\mathtt{Y} \lessdot \mathtt{Pair}\langle\mathtt{X},\mathtt{Y}\rangle$, which triggers the occurs-check in unification and is hence rejected.

The two solutions can be combined to

```
class MultPair {
  <X extends T1, Y extends T2>
  T0 mult (Pair<X,Y> p) { return p.fst.mult(p.snd); }
}
```

where $(T_0, T_1, T_2) \in \{(\mathtt{Int}, \mathtt{Int}, \mathtt{Int}), (\mathtt{Double}, \mathtt{Double}, \mathtt{Double})\}$.

2.2 Inheritance

Let's start with the artificial example in the left listing of Figure 3 and ignore the `Double` class. Type inference proceeds according to the inheritance hierarchy starting from the superclasses. In class `A1`, the inferred method type is `Int A1.m (Int)`. Class `B1` is a subclass of `A1` which must override `m` as there is no overloading in FGJ. However, the inferred method type is `<T> T B1.m(T)`, which is not a correct method override for `A1.m()`. Hence, GTI must instantiate the method type in the subclass `B1` to `Int B1.m(Int)`.

Conversely, for the right listing of Figure 3, GTI infers the types `<T> T A2.m (T)` and `Int B2.m (Int)`. Again, these types do not give rise to a correct method override and GTI is now forced to instantiate the type in the superclass to `Int A2.m (Int)`.

In full Java, type inference would have to offer two alternative results: either two different overloaded methods (one inherited and one local) in `B1`/`B2` or impose the typing `Int B1.m(Int)` or `Int A2.m(Int)` to enforce correct overriding.

```
class Function<S,T> {
  T apply(S arg) { return this.apply (arg); }
}
```

Listing 1 Function class.

2.3 Inheritance and Generics

Suppose we are given a generic class for modeling functions in FGJ (Listing 1). This code is constructed to serve as an “abstract” super class to derive more interesting subclasses. The class `Function<S,T>` must be presented in this explicit way. Its type annotations **cannot** be inferred by GTI because the use of the generic class parameters in the method type cannot be inferred from the implementation.

If we applied GTI to the type-erased version of Listing 1, the `apply` method would be considered a generic method:

```
apply (arg) { ... }
```

-GTI->

```
<A,B> B apply (A arg) { ... }
```

The typing of `apply` in Listing 1 is an instance of this result, so that completeness of GTI is preserved!

Now that we have the abstract class `Function<S,T>` at our disposal, let us apply GTI to a class of boxed values with a `map` function:

```
class Box<S> {
  S val;
  map(f) {
    return new Box<>(f.apply(this.val));
} }
```

GTI finds the following constraints
- the return value must be of type `Box<T>`, for some type `T`,
- `T` is a supertype of the type returned by `f.apply`,
- `apply` is defined in class `Function<S1,T1>` with type `T1 apply(S1 arg)`,
- hence `T1` <: `T` and `S` <: `S1` (because `this.val` : `S`),

and resolves them to the desired outcome where `T1=T` and `S=S1` using the methods of Simonet [26].

```
class Box<S> {
  S val;
  <T> Box<T> map(Function<S,T> f) {
    return new Box<T>(f.apply<S,T>(this.val));
} }
```

But what happens if we add subclasses of `Function`? For example:

```
class Not extends Function<Bool,Bool> {
  apply(b) { return b.not(); }
}
class Negate extends Function<Int,Int> {
  apply(x) { return x.negate(); }
}
```

If we rerun GTI with these classes, we now have additional possibilities to invoke the `apply` method. With `Not`, we need to use the generic type of `Function.apply()`, but instantiate it according to `Function<Bool,Bool>`. Thus, we obtain the constraints `Bool` $\lessdot$ `T` and `S` $\lessdot$ `Bool` for `T = Bool` and `S = Bool`, which are both satisfiable. With `Negate` we run into the same situation with the constraints `Int` $\lessdot$ `Int` and `Int` $\lessdot$ `Int`.

```
class List<A> {
  List<A> add(A item) {...}
  A get() { ... }
}
```

```
class Global{
  m(a){
    return a.add(this).get();
} }
```

Figure 4 Example for multiple inferred types.

Here is another subclass of `Function<S,T>` that we want to consider.

```
class Identity<S> extends Function<S,S> {
  S apply(S arg) { return arg; }
}
```

Here, we obtain the following type constraints

- `apply` is defined in class `Identity<S1>` with type `S1 apply (S1 arg)`,
- hence `S1` $\lessdot$ `T` and `S` $\lessdot$ `S1`.

Resolving the constraints yields `S = T` thus the typing

```
Box<S> map(Identity<S> f);
```

which is an instance of the previous typing.

2.4 Multiple typings

Global type inference processes classes in order of dependency. To see why, consider the classes `List<A>` and `Global` in Figure 4. Class `Global` may depend on class `List` because `Global` uses methods `add` and `get` and `List` defines methods with the same names. The dependency is only approximate because, in general, there may be additional classes providing methods `add` and `get`.

In the example, it is safe to assume that the types for the methods of class `List` are already available, either because they are given (as in the code fragment) or because they were inferred before considering class `Global`.

The method `m` in class `Global` first invokes `add` on `a`, so the type of `a` as well as the return type of `a.add(this)` must be `List<T>`, for some `T`. As `this` has type `Global`, it must be that `Global` is a subtype of `T`, which gives rise to the constraint `Global` $\lessdot$ `T`. By the typing of `get()` we find that the return type of method `m` is also `T`.

But now we are in a dilemma because FGJ only supports *upper bounds* for type variables,[6] so that `Global` $\lessdot$ `T` is not a valid constraint in FGJ. To stay compatible with this restriction, global type inference expands the constraint by instantiating `T` with the (two) superclasses fulfilling the constraint, `Global` and `Object`. They give rise to two incomparable types for `m`, `List<Global> -> Global` and `List<Object> -> Object`. So there are two different FGJ programs that are completions of the `Global` class.

GTI models these instances by inferring an *intersection type* `List<Global> -> Global & List<Object> -> Object` for method `m` and the different FGJ-completions of class `Global` are instances of the intersection type:[7]

[6] Java has the same restriction. Lower bounds are only allowed for wildcards.

[7] The cognoscenti will be reminded of overloading. As FGJ does not support overloading, we rely on resolution by subsequent uses of the method. Moreover, this intersection type cannot be realized by overloading in a Java source program because it is resolved according to the raw classes of the arguments, in this case `List`. It can be realized in bytecode which supports overloading on the return type, too.

```
class UsePair {
  <X,Y> Object prc(Pair<X,Y> p) {
    return this.prc<Y,X>
        (p.swap<X,Y>());
} }
```

```
class UsePair {
  prc(p) {
    return this.prc (p.swap());

} }
```

Figure 5 Example for polymorphic recursion.

```
class Global {
  Global m(List<Global> a) {
    return a.add(this).get();
}
```

```
class Global {
  Object m(List<Object> a) {
    return a.add(this).get();
}
```

In this sense, the inferred intersection type represents a principal typing for the class. Additional classes in the program may further restrict the number of viable types. Suppose we define a class `UseGlobal` as follows:

```
class UseGlobal {
  main() {
    return new Global().m((List<Object>) new List());
} }
```

Due to the dependency on `Global.m()`, type inference considers this class after class `Global`. As it uses `m` at type `List<Object> -> Object`, global type inference narrows the type of `m` to just this alternative.

2.5 Polymorphic recursion

A program uses *polymorphic recursion* if there is a generic method that is invoked recursively at a more specific type than its definition. As a toy example for polymorphic recursion consider the FGJ class `UsePair` with a generic method `prc` that invokes itself recursively on a swapped version of its argument pair (Figure 5, left). This method makes use of polymorphic recursion because the type of the recursive call is different from the declared type of the method. More precisely, the declared argument type is `Pair<X,Y>` whereas the argument of the recursive call has type `Pair<Y,X>` – an instance of the declared type.

For this particular example, global type inference succeeds on the corresponding stripped program shown in Figure 5, right, but it yields a more restrictive typing of `<X> Object prc (Pair<X,X> p)` for the method. A minor variation of the FGJ program with a non-variable instantiation makes type inference fail entirely:

```
class UsePair2 {
  <X,Y> Object prc(Pair<X,Y> p) {
    return this.prc<Y,Pair<X,Y>> (new Pair (p.snd, p));
  }
}
```

Polymorphic recursion is known to make type inference intractable [9, 12] because it can be reduced to an undecidable semi-unification problem [13]. However, *type checking* with polymorphic recursion is tractable and routinely used in languages like Haskell and Java.

GTI does not infer method types with polymorphic recursion. Inference either fails or returns a more restrictive type. Classes making use of polymorphic recursion need to supply explicit typings for methods in question.

$$
\begin{aligned}
&\texttt{T} ::= \texttt{X} \mid \texttt{N}\\
&\texttt{N} ::= \texttt{C<}\overline{\texttt{T}}\texttt{>}\\
&\texttt{L} ::= \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N} \; \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\\
&\texttt{K} ::= \texttt{C}(\overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\}\\
&\texttt{M} ::= \texttt{m}(\overline{\texttt{x}})\ \{\texttt{return e};\}\\
&\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \texttt{new C}(\overline{\texttt{e}}) \mid (\texttt{N})\ \texttt{e}
\end{aligned}
$$

Figure 6 Syntax of FGJ-GT.

3 Featherweight Generic Java with Global Type Inference

This section defines the syntax and type system of a modified version of the language Featherweight Generic Java (FGJ) [11], which we call FGJ-GT (with Global Type Inference). The main omissions with respect to FGJ are method types specifications and polymorphic recursion. We finish the section by formally connecting FGJ and FGJ-GT and by establishing some properties about polymorphic recursion in FGJ.

3.1 Syntax

Figure 6 defines the syntax of FGJ-GT. Compared to FGJ, type annotations for method parameters and method return types are omitted. Object creation via `new` as well as method calls come do not require instantiation of their generic parameters. We keep the class constraints $\overline{\texttt{X}} \lhd \overline{\texttt{N}}$ as well as the types for fields $\overline{\texttt{T}}\ \overline{\texttt{f}}$ as we consider them as part of the specification of a class.

We make the following assumptions for the input program:

- All types `N` and `T` are well formed according to the rules of FGJ, which carry over to FGJ-GT (see Fig. 8).
- The methods of a class call each other mutually recursively.
- The classes in the input are topologically sorted so that later classes only call methods in classes that come earlier in the sorting order.

Our requirements on the method calls do not impose serious restrictions as any class, say `C`, can be transformed to meet them as follows. A preliminary dependency analysis determines an approximate call graph. We cluster the methods of `C` according to the n strongly connected components of the call graph. Then we split the class into a class hierarchy $\texttt{C}_1 \lhd \ldots \lhd \texttt{C}_n$ such that each class $\texttt{C}_i$ contains exactly the methods of one strongly connected component and assign a method cluster to $\texttt{C}_i$ if all calls to methods of `C` now target methods assigned to $\texttt{C}_j$, for some $j \geq i$. The class $\texttt{C}_1$ replaces `C` everywhere in the program: in subtype bounds, in `new` expressions, and in casts. More precisely, if `C` is defined by $\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ldots$, then the class headers for the $\texttt{C}_i$ are defined as follows:

- $\texttt{class C}_\texttt{i}\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{C}_{\texttt{i+1}}\texttt{<}\overline{\texttt{X}}\texttt{>}\ldots$, for $1 \leq i < n$ and
- $\texttt{class C}_\texttt{n}\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ldots$.

It follows from this discussion that the resulting classes have to be processed backwards starting with $\texttt{C}_n, \texttt{C}_{n-1}, \ldots, \texttt{C}_1$. Figure 7 showcases this process with a short example.

```
class C extends Object {
  m1(a){
    return a;
  }
  m2(b){
    return this.id(a);
  }
}
```

(a) The methods `m1` and `m2` can be separated.

```
class C1 extends C2 {
  m2(b){
    return this.id(a);
  }
}
class C2 extends Object {
  m1(a){
    return a;
  }
}
```

(b) After the transformation.

Figure 7 Example for splitting a class into its strongly connected components.

3.2 Typing

We start with some notation. An environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}} : \overline{\mathtt{T}}$; a type environment Δ is a finite mapping from type variables to nonvariable types, written $\overline{\mathtt{X}} \mathrel{<:} \overline{\mathtt{N}}$, which takes each type variable to its bound. As in FGJ, we do not impose an ordering on environment entries to enable F-bounded polymorphism.

There is a new method environment Π which maps pairs of a class header $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ and a method name `m` to a set of method types of the form $\texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}$. It supports the *mtype* function that relates a nonvariable type `N` and a method name `m` to a method type.

The judgments for subtyping $\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}$ and well-formedness of types $\Delta \vdash \mathtt{T}\ \mathtt{ok}$ (Figure 8) stay the same as in FGJ.

The overall approach to typing changes with respect to FGJ. In FGJ, classes can be checked in any order as the method typings of all other classes are available in the syntax. FGJ-GT processes classes in order such that early classes do not invoke methods in late classes.

The new program typing rule (GT-PROGRAM) for the judgment $\vdash \overline{\mathtt{L}} : \Pi$ reflects this approach. It starts with an empty method environment and applies class typing to each class in the sequence provided. Each processed class adds its method typings to the method environment which is threaded through to constitute the program type as the final method environment Π.

Expression typing $\Pi; \Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ changes subtly (see Figure 9). As FGJ-GT omits some type annotations, we are forced to adapt some of FGJ's typing rules. The new rules infer omitted types and disable polymorphic recursion.

The new method environment Π is only used in the revised rule for method invocation (GT-INVK), where it is passed as an additional parameter to *mtype*. The revised definition of *mtype* (Figure 10) locates the class that contains the method definition by traversing the subtype hierarchy and looks up the method type in environment Π, which contains the method types that were already inferred. Our definition of *mtype* does not support overloading as Π relate at most one type to each method definition (cf. rule (GR-CLASS)). The instantiation of the method's type parameters is inferred in FGJ-GT.

The rule (GT-NEW) changes to infer the instantiation of the class's type parameters: the rule simply assumes a suitable instantiation by some $\overline{\mathtt{U}}$.

Finally, (GT-CAST) replaces the three rules (GT-UCAST'), (GT-DCAST'), and (GT-SCAST') of FGJ. This is a slight simplification with respect to FGJ. While the three original rules cover disjoint use cases (upcast, downcast, and stupid cast that is sure to fail) of the cast operation, they are not exhaustive! The rule (GT-DCAST') only admits downcasts that

Subtyping:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \text{(S-REFL)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \qquad \text{(S-TRANS)}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \qquad \text{(S-VAR)}$$

$$\frac{\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\{\ldots\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \qquad \text{(S-CLASS)}$$

Well-formed types:

$$\Delta \vdash \mathtt{Object}\ \mathtt{ok} \qquad \text{(WF-OBJECT)}$$

$$\frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \mathtt{ok}} \qquad \text{(WF-VAR)}$$

$$\frac{\begin{array}{c}\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\{\ldots\} \\ \Delta \vdash \overline{\mathtt{T}}\ \mathtt{ok} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \mathtt{ok}} \qquad \text{(WF-CLASS)}$$

Figure 8 Well-formedness and subtyping.

work the same in a type-passing semantics as in a type erasure semantics. We elide this distinction for simplicity, though it could be handled by introducing constraints analogous to the *dcast* function from FGJ.

The typing rule for a method $\mathtt{m}$, (GT-METHOD), changes significantly. By our assumption on the order, in which classes are processed, the typing of $\mathtt{m}$ is already provided by the method environment Π. The type environment Δ is also provided as an input. Moreover, to rule out polymorphic recursion, the assumptions about the local methods of class $\mathtt{C}$ are monomorphic at this stage. The rule type checks the body for the inferred type of method $\mathtt{m}$.

All this information is provided and generated by the rule for class typing, (GT-CLASS). A class typing for $\mathtt{C}$ receives an incoming method type environment Π and generates an extended one Π'' which additionally contains the method types inferred for $\mathtt{C}$.

In Π', we generate some monomorphic types for all methods of class $\mathtt{C}$. We use these types to check the methods. Afterwards, we return generalized versions of these same types in Π''. All method types use the same generic type variables $\overline{\mathtt{Y}}$ with the same constraints $\overline{\mathtt{P}}$. It is safe to make this assumption in the absence of polymorphic recursion as we will show in Proposition 5.

3.3 Soundness of Typing

We show that every typing derived by the FGJ-GT rules gives rise to a completion, that is, a well-typed FGJ program with the same structure.

▶ **Definition 1** (Erasure). *Let* $\mathtt{e}'$, $\mathtt{M}'$, $\mathtt{K}'$, $\mathtt{L}'$ *be expression, method definition, constructor definition, class definition for FGJ. Define erasure functions* $|\mathtt{e}'|$, $|\mathtt{M}'|$, $|\mathtt{K}'|$, $|\mathtt{L}'|$ *that map to the corresponding syntactic categories of FGJ-GT as shown in Figure 11.*

▶ **Definition 2** (Completion). *An FGJ expression* $\mathtt{e}'$ *is a* completion *of a FGJ-GT expression* $\mathtt{e}$ *if* $\mathtt{e} = |\mathtt{e}'|$. *Completions for method definitions, constructor definitions, and class definitions are defined analogously.*

Expression typing:

$$\Pi;\Delta;\Gamma \vdash \texttt{x} : \Gamma(\texttt{x}) \qquad \text{(GT-VAR)}$$

$$\frac{\Pi;\Delta;\Gamma \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{fields}(\mathit{bound}_\Delta(\texttt{T}_0)) = \overline{\texttt{T}}\ \overline{\texttt{f}}}{\Pi;\Delta;\Gamma \vdash \texttt{e}_0.\texttt{f}_\texttt{i} : \texttt{T}_\texttt{i}} \qquad \text{(GT-FIELD)}$$

$$\frac{\begin{array}{c}\Pi;\Delta;\Gamma \vdash \texttt{e}_0 : \texttt{T}_0 \qquad \mathit{mtype}(\texttt{m}, \mathit{bound}_\Delta(\texttt{T}_0), \Pi) = \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U} \\ \Delta \vdash \overline{\texttt{V}}\ \texttt{ok} \qquad \Delta \vdash \overline{\texttt{V}} \texttt{<:}\ [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \qquad \Pi;\Delta;\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} \texttt{<:}\ [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}}\end{array}}{\Pi;\Delta;\Gamma \vdash \texttt{e}_0.\texttt{m}(\overline{\texttt{e}}) : [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}} \qquad \text{(GT-INVK)}$$

$$\frac{\Delta \vdash \texttt{N ok} \quad \texttt{N} = \texttt{C<}\overline{\texttt{U}}\texttt{>} \quad \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \quad \Pi;\Delta;\Gamma \vdash \overline{\texttt{e}} : \overline{\texttt{S}} \quad \Delta \vdash \overline{\texttt{S}} \texttt{<:}\ \overline{\texttt{T}}}{\Pi;\Delta;\Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \texttt{N}} \qquad \text{(GT-NEW)}$$

$$\frac{\Pi;\Delta;\Gamma \vdash \texttt{e}_0 : \texttt{T}_0}{\Pi;\Delta;\Gamma \vdash (\texttt{N})\texttt{e}_0 : \texttt{N}} \qquad \text{(GT-CAST)}$$

Method typing:

$$\frac{\begin{array}{c}\forall \overline{\texttt{T}}, \texttt{T} : \texttt{<>}\overline{\texttt{T}} \to \texttt{T} \in \Pi(\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m}) \qquad \Delta \vdash \texttt{S} \texttt{<:}\ \texttt{T} \\ \Pi;\Delta;\overline{\texttt{x}} : \overline{\texttt{T}},\ \texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>} \vdash \texttt{e}_0 : \texttt{S} \\ \mathit{override}(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}, \Pi)\end{array}}{\Pi, \Delta \vdash \texttt{m}(\overline{\texttt{x}})\{\texttt{return e}_0;\}\ \texttt{OK in C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N with <}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}} \qquad \text{(GT-METHOD)}$$

Class typing:

$$\frac{\begin{array}{c}\Pi' = \Pi \cup \{\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} \mapsto \texttt{<>}\overline{\texttt{T}_\texttt{m}} \to \texttt{T}_\texttt{m} \mid \texttt{m} \in \overline{\texttt{M}}\} \\ \Pi'' = \Pi \cup \{\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} \mapsto \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}_\texttt{m}} \to \texttt{T}_\texttt{m} \mid \texttt{m} \in \overline{\texttt{M}}\} \\ \Delta = \overline{\texttt{X}} \texttt{<:}\ \overline{\texttt{N}}, \overline{\texttt{Y}} \texttt{<:}\ \overline{\texttt{P}} \qquad \Delta \vdash \overline{\texttt{P}}\ \texttt{ok} \qquad \forall \texttt{m} : \Delta \vdash \overline{\texttt{T}_\texttt{m}}, \texttt{T}_\texttt{m}\ \texttt{ok} \\ \overline{\texttt{X}} \texttt{<:}\ \overline{\texttt{N}} \vdash \overline{\texttt{N}}, \texttt{N}, \overline{\texttt{T}}\ \texttt{ok} \qquad \mathit{fields}(\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}} \\ \Pi', \Delta \vdash \overline{\texttt{M}}\ \texttt{OK IN C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N with <}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>} \\ \texttt{K} = \texttt{C}(\overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}})\{\texttt{super}(\overline{\texttt{g}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\}\end{array}}{\Pi \vdash \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK} : \Pi''} \qquad \text{(GT-CLASS)}$$

Program typing:

$$\frac{\emptyset \vdash \texttt{L}_1 : \Pi_1 \quad \Pi_1 \vdash \texttt{L}_2 : \Pi_2 \quad \ldots \quad \Pi_{n-1} \vdash \texttt{L}_n : \Pi_n}{\vdash \overline{\texttt{L}} : \Pi_n} \qquad \text{(GT-PROGRAM)}$$

Figure 9 Typing rules.

Field lookup:

$$fields(\texttt{Object}) = \bullet \qquad \text{(F-OBJECT)}$$

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

Method type lookup:

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

$$\frac{\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mtype(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}, \Pi) = mtype(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}, \Pi)} \qquad \text{(MT-SUPER)}$$

Valid method overriding:

$$\frac{mtype(\texttt{m}, \texttt{N}, \Pi) = \texttt{<}\overline{\texttt{Z}} \lhd \overline{\texttt{Q}}\texttt{>}\overline{\texttt{U}} \to \texttt{U} \text{ implies } \overline{\texttt{P}}, \overline{\texttt{T}} = [\overline{\texttt{Y}}/\overline{\texttt{Z}}](\overline{\texttt{Q}}, \overline{\texttt{U}}) \text{ and } \overline{\texttt{Y}} \texttt{<:} \overline{\texttt{P}} \vdash \texttt{T}_0 \texttt{<:} [\overline{\texttt{Y}}/\overline{\texttt{Z}}]\texttt{U}_0}{override(\texttt{m}, \texttt{N}, \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}_0, \Pi)}$$

Figure 10 Auxiliary functions.

$$\begin{aligned}
|\texttt{x}| &= \texttt{x} \\
|\texttt{e.f}| &= |\texttt{e}|\texttt{.f} \\
|\texttt{e<}\overline{\texttt{T}}\texttt{>.m(}\overline{\texttt{e}}\texttt{)}| &= |\texttt{e}|\texttt{.m(}|\overline{\texttt{e}}|\texttt{)} \\
|\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}| &= \texttt{new C(}|\overline{\texttt{e}}|\texttt{)} \\
|\texttt{(N) e}| &= \texttt{(N)}\ |\texttt{e}| \\
|\texttt{<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{> T m(}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{return e;\}}| &= \texttt{m(}\overline{\texttt{x}}\texttt{) \{return } |\texttt{e}|\texttt{;\}} \\
|\texttt{C(}\overline{\texttt{U}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{;\}}| &= \texttt{C(}\overline{\texttt{g}}, \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}} = \overline{\texttt{f}}\texttt{;\}} \\
|\texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\,\overline{\texttt{M}}\}| &= \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ |\texttt{K}|\ |\overline{\texttt{M}}|\}
\end{aligned}$$

Figure 11 Erasure functions.

▶ **Theorem 3.** *Suppose that* $\vdash \overline{\mathtt{L}} : \Pi$ *such that* $|\Pi(\mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>.m})| = 1$*, for all* $\mathtt{C.m}$ *defined in* $\overline{\mathtt{L}}$*. Then there is a completion* $\overline{\mathtt{L}}'$ *of* $\overline{\mathtt{L}}$ *such that* $\overline{\mathtt{L}}'$ $\mathtt{OK}$ *is derivable in FGJ.*

Proof. The proof is by induction on the length of $\overline{\mathtt{L}}$.

Consider the class typing $\Pi \vdash \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK} : \Pi''$ for an element of $\overline{\mathtt{L}}$.

We assume that all classes before $\mathtt{L}$ are completed according to the incoming Π: If $\Pi(\mathtt{D<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>.n}) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T}$, then $\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ n(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)}\ldots$ is in the completion of $\mathtt{D}$.

Clearly, we can construct a completion for the class, if we can do so for each method. So we have to construct $\overline{\mathtt{M}}'$ such that $\overline{\mathtt{M}}'$ $\mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}$.

Inversion of (GT-CLASS) yields

$$\Pi' = \Pi \cup \{\mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>.m} \mapsto \mathtt{<>}\overline{\mathtt{T_m}} \to \mathtt{T_m} \mid \mathtt{m} \in \overline{\mathtt{M}}\} \tag{1}$$

$$\Pi'' = \Pi \cup \{\mathtt{C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>.m} \mapsto \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T_m}} \to \mathtt{T_m} \mid \mathtt{m} \in \overline{\mathtt{M}}\} \tag{2}$$

$$\Pi', \Delta \vdash \overline{\mathtt{M}}\ \mathtt{OK\ \ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \ with\ \ <}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>} \tag{3}$$

$$\Delta = \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}}, \overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{P}} \tag{4}$$

Given some $\mathtt{M} = \mathtt{m}(\overline{\mathtt{x}})\{\mathtt{return\ e_0};\} \in \overline{\mathtt{M}}$, we show that

$$\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T_m\ m(}\overline{\mathtt{T_m}}\ \overline{\mathtt{x}}\mathtt{)}\{\mathtt{return\ e_0'};\}\ \mathtt{OK\ \ IN\ \ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \tag{5}$$

is derivable for such completion $\mathtt{e}_0'$ of $\mathtt{e_0}$.

By inversion of (3) for $\mathtt{M}$, we obtain

$$\mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T_m}} \to \mathtt{T_m}, \Pi) \tag{6}$$

$$\Pi; \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T_m}},\ \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e_0} : \mathtt{S} \tag{7}$$

$$\Delta \vdash \mathtt{S} \mathtt{<:} \mathtt{T_m} \tag{8}$$

As Δ in (4) is defined as in (GT-METHOD'), the well-formedness judgments are all given, the subtyping judgment (8) is given as well as the override (7), the rule (GT-METHOD') applies if we can establish

$$\Delta; \overline{\mathtt{x}} : \overline{\mathtt{T_m}},\ \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e_0'} : \mathtt{S} \tag{9}$$

for a completion of $\mathtt{e_0}$.

To see that, we need to consider the rules (GT-NEW), (GT-CAST), and (GT-INVK). The (GT-NEW) rule poses the existence of some $\overline{\mathtt{U}}$ such that $\mathtt{N} = \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}$ for checking $\mathtt{e} = \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{N}$. In the completion, we define $\mathtt{e}' = \mathtt{new\ N(}\overline{\mathtt{e}'}\mathtt{)} : \mathtt{N}$ to apply rule (GT-NEW') to the completions of the arguments.

The rule (GT-CAST) splits into three rules (GT-UCAST'), (GT-DCAST'), and (GT-SCAST'). These rules are disjoint, so that at most one of them applies to each occurrence of a cast. Here we assume a more liberal version of (GT-DCAST') that admits downcasts that are not stable under type erasure semantics.

For the rule (GT-INVK), we first consider calls to methods not defined in the current class. By our assumption on previously checked classes $\mathtt{D}$ and their methods $\mathtt{n}$, $\mathit{mtype}(\mathtt{n}, \mathtt{D}, \Pi)$ $= \{\mathit{mtype}'(\mathtt{n}, \mathtt{D}')\}$ where the right side lookup happens in the completion following the definitions for FGJ (i.e., $\mathtt{D}'$ is the completion for $\mathtt{D}$). The (GT-INVK) rule poses the existence of some $\overline{\mathtt{V}}$ that satisfies the same conditions as in (GT-INVK'). Hence, we define the completion of $\mathtt{e_0.n(}\overline{\mathtt{e}}\mathtt{)} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$ as $\mathtt{e_0'.n<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}'}\mathtt{)} : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}$.

Next we consider calls to methods $\mathtt{n}$ defined in the current class, say, $\mathtt{C}$. For those methods, $\mathit{mtype}(\mathtt{n}, \mathtt{C}, \Pi) = \mathtt{<>}\overline{\mathtt{U}} \to \mathtt{U}$, a non-generic type. By the definition of Π'', we know that the type of this method will be published in the completion as $\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$. Hence, $\mathit{mtype}'(\mathtt{n}, \mathtt{C}') = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$. As methods in $\mathtt{C}$ are mutually recursive, the rule must pose that $\overline{\mathtt{V}} = \overline{\mathtt{Y}}$ (cf. Proposition 5). This setting fulfills all assumptions:

$$\Delta \vdash \overline{\mathtt{Y}}\ \mathtt{ok} \tag{10}$$

$$\Delta \vdash \overline{\mathtt{Y}} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \tag{11}$$

We set the completion of $\mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}}) : [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\mathtt{U}$ to $\mathtt{e}_0'.\mathtt{n<}\overline{\mathtt{Y}}\mathtt{>}(\overline{\mathtt{e}}') : [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}]\mathtt{U}$, which is derivable in FGJ.

The remaining expression typing rules are shared between FGJ and FGJ-GT, so they do not affect completions. ◀

3.4 Polymorphic Recursion, Formally

Consider an FGJ class $\mathtt{C}$ with n mutually recursive methods $\mathtt{m}_i : \forall \overline{\mathtt{X}}_i.\overline{\mathtt{A}}_i \to \overline{\mathtt{A}}_i$, for $1 \le i \le n$. Define the *instantiation multigraph* $\mathit{IG}(\mathtt{C})$ as a directed multigraph with vertices $\{1, \dots, n\}$. Edges between i and j in this graph are labeled with a substitution from $\overline{\mathtt{X}}_j$ to types in $\mathtt{m}_i$, which may contain type variables from $\overline{\mathtt{X}}_i$. In particular, if $\mathtt{m}_i$ invokes $\mathtt{m}_j$ where the generic type variables in the type of $\mathtt{m}_j$ are instantiated with substitution $\overline{\mathtt{U}}/\overline{\mathtt{X}}_j$ (see rule GT-INVK), then $i \overset{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j}{\longrightarrow} j$ is an edge of $\mathit{IG}(\mathtt{C})$.

Define the *closure of the instantiation multigraph* $\mathit{IG}^*(\mathtt{C})$ as the multigraph obtained from $\mathit{IG}(\mathtt{C})$ by applying the following rule, which composes the instantiating substitutions, exhaustively:

$$i \overset{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j}{\longrightarrow} j \quad \wedge \quad j \overset{\overline{\mathtt{V}}/\overline{\mathtt{X}}_k}{\longrightarrow} k \qquad \Rightarrow \qquad i \overset{[\overline{\mathtt{U}}/\overline{\mathtt{X}}_j]\overline{\mathtt{V}}/\overline{\mathtt{X}}_k}{\longrightarrow} k \tag{12}$$

▶ **Definition 4.** *Method* $\mathtt{m}_i$ *is* involved in polymorphic recursion *if there is an edge*

$$i \overset{\overline{\mathtt{W}}/\overline{\mathtt{X}}_i}{\longrightarrow} i \quad \in \mathit{IG}^*(\mathtt{C}) \qquad \textit{such that} \qquad \overline{\mathtt{W}} \neq \overline{\mathtt{X}}_i \tag{13}$$

For the toy example in Figure 5, we obtain the multigraph $\mathit{IG}^*(\mathtt{UsePair})$ which indicates that `prc` is involved in polymorphic recursion:

$\mathit{IG}(\mathtt{UsePair})$	$\mathit{IG}^*(\mathtt{UsePair})$	
$\mathtt{prc} \overset{\mathtt{Y,X/X,Y}}{\longrightarrow} \mathtt{prc}$	$\mathtt{prc} \overset{\mathtt{Y,X/X,Y}}{\longrightarrow} \mathtt{prc}$	$\mathtt{prc} \overset{\mathtt{X,Y/X,Y}}{\longrightarrow} \mathtt{prc}$

The call to `swap` does not appear in the graph because `swap` is defined in a different class.

For `UsePair2`, we obtain a multigraph $\mathit{IG}^*(\mathtt{UsePair2})$ with infinitely many edges which is also clear indication for polymorphic recursion:

$\mathit{IG}(\mathtt{UsePair2})$	$\mathit{IG}^*(\mathtt{UsePair2})$
$\mathtt{prc} \overset{\mathtt{Y,Pair<X,Y>/XY}}{\longrightarrow} \mathtt{prc}$	$\mathtt{prc} \overset{\mathtt{Y,Pair<X,Y>/XY}}{\longrightarrow} \mathtt{prc}$
	$\mathtt{prc} \overset{\mathtt{Pair<X,Y>,Pair<Y,Pair<X,Y>>/XY}}{\longrightarrow} \mathtt{prc}$
	$\dots$

Clearly, $\mathit{IG}(\mathtt{C})$ is finite and can be constructed effectively by collecting the instantiating substitutions from all method call sites. Repeated application of the propagation rule (12) either results in saturation where no edge of the resulting multigraph satisfies (13) or it detects an instantiating edge as in condition (13).

The following condition is necessary for the absence of polymorphic recursion.

▶ **Proposition 5.** *Suppose an FGJ class* `C` *has* n *methods, which are mutually recursive. If* `C` *does not exhibit polymorphic recursion, then*

- *all methods quantify over the same number of generic variables;*
- *if a method has generic variables* $\overline{\mathtt{X}}$*, then each call to a method of* `C` *instantiates with a permutation of the* $\overline{\mathtt{X}}$*;*
- $IG^*(\mathtt{C})$ *is finite.*

Proof. Suppose for a contradiction that there are two distinct methods $\mathtt{m}_i$ and $\mathtt{m}_j$ with generic variables $\overline{\mathtt{X}}_i$ and $\overline{\mathtt{X}}_j$, respectively, where $|\overline{\mathtt{X}}_i| < |\overline{\mathtt{X}}_j|$. By mutual recursion, $\mathtt{m}_i$ invokes $\mathtt{m}_j$ directly or indirectly and vice versa. Hence, $IG^*(\mathtt{C})$ contains edges from i to j and back:

$$i \xrightarrow{\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j \qquad j \xrightarrow{\overline{\mathtt{V}}/\overline{\mathtt{X}}_i} i$$

As $IG^*(\mathtt{C})$ is closed under composition, it must also contain the edge

$$j \xrightarrow{[\overline{\mathtt{V}}/\overline{\mathtt{X}}_i]\overline{\mathtt{U}}/\overline{\mathtt{X}}_j} j.$$

By assumption `C` does not use polymorphic recursion, so it must be that $[\overline{\mathtt{V}}/\overline{\mathtt{X}}_i]\overline{\mathtt{U}}/\overline{\mathtt{X}}_j = \overline{\mathtt{X}}_j/\overline{\mathtt{X}}_j$. To fulfill this condition, all components of $\overline{\mathtt{U}}$ must be variables $\in \overline{\mathtt{X}}_i$. As $|\overline{\mathtt{X}}_i| < |\overline{\mathtt{X}}_j| = |\overline{\mathtt{U}}|$, there must be some variable $\mathtt{X} \in \overline{\mathtt{X}}_i$ that occurs more than once in $\overline{\mathtt{U}}$, say, at positions j_1 and j_2. But that means the variables at positions j_1 and j_2 in $\overline{\mathtt{X}}_j$ are mapped to the same component of $\overline{\mathtt{V}}$. This is a contradiction because this substitution cannot be the identity substitution $\overline{\mathtt{X}}_j/\overline{\mathtt{X}}_j$.

Hence, all methods have the same number of generic variables and all instantiations must use variables.

Suppose now that there is a direct call from $\mathtt{m}_i$ to $\mathtt{m}_j$ where the instantiation $\overline{\mathtt{U}}/\overline{\mathtt{X}}_j$ is not a permutation. Hence, there is a variable that appears more than once in $\overline{\mathtt{U}}$, which leads to a contradiction using similar reasoning as before.

Hence, all instantiations must be permutations over a finite set of variables, so that $IG^*(\mathtt{C})$ is finite. ◀

Moreover, if a class has only mutually recursive methods without polymorphic recursion, we can assume that each method uses the same generic variables, say $\overline{\mathtt{X}}$, and each instantiation for class-internal method calls is the identity $\overline{\mathtt{X}}/\overline{\mathtt{X}}$.

Using the same generic variables is achieved by α conversion. By Proposition 5, we already know that each instantiation is a permutation. Each self-recursive call must use an identity instantiation already, otherwise it would constitute an instance of polymorphic recursion. Suppose that method `m` calls method `n` instantiated with a non-identity permutation, say π so that parameter $\mathtt{X_i}$ of `n` gets instantiated with $\mathtt{X}_{\pi(\mathtt{i})}$ of `m`. In this case, we reorder the generic parameters of `n` according to the inverse permutation π^{-1} and propagate this permutation to all call sites of `n`. For the call in `m`, we obtain the identity permutation $\pi \cdot \pi^{-1}$, for self-recursive calls inside `n`, the instantiation remains the identity (for the same reason), for a call site in another method which instantiates `n` with permutation σ, we change that permutation to $\sigma \cdot \pi^{-1}$, which is again a permutation. This way, we can eliminate all non-identity instantiations from calls inside `m`.

We move our attention to `n`. Each self-recursive call and each call to `m` uses the identity instantiation, the latter by construction. So we only need to consider calls to $\mathtt{n}' \notin \{\mathtt{n}, \mathtt{m}\}$ with an instantiation which is not the identity permutation. We can also assume that $\mathtt{n}'$ is not called from `m`: otherwise, `n'` would have the generic variables in the same order as `m` and hence as `n`. But that means we can fix all calls to $\mathtt{n}'$ by applying the inverse permutations as for `n` *without disturbing the already established identity instantiations.*

```
class C1 extends Object {
  m1(){ return new C2().m2(); }
}
class C2 extends Object{
  m2(){ return new C1().m1(); }
}
```

Figure 12 Invalid FGJ-GT program.

```
class D1 extends Object {
  m(){ return ...; }
}
class D2 extends Object{
  m1(x){ return new D2().m2(); }
  m2( ){ return new D2().m1(
                new D1().m()); }
}
```

Figure 13 Valid FGJ-GT program.

Each such step eliminates all non-identity instantiations for at least one method without disturbing previous identity instantiations. Hence, the procedure terminates after finitely many steps with a class with all instantiations being identity permutations.

4 Type inference algorithm

This section presents our type inference algorithm. The algorithm is given method assumptions Π and applied to a single class `L` at a time:

$$
\begin{array}{l}
\textbf{FJTypeInference}(\Pi, \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\ldots\}) = \\
\qquad \begin{array}{lll}
\textbf{let}\ (\overline{\lambda}, C) & = \textbf{FJType}(\Pi, \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>} \lhd \texttt{N}\ \{\ldots\}) & \text{// constraint generation} \\
\quad (\sigma, \overline{\texttt{Y}} \lhd \overline{\texttt{P}}) & = \textbf{Unify}(C, \overline{\texttt{X}} <: \overline{\texttt{N}}) & \text{// constraint solving}
\end{array} \\
\textbf{in}\ \Pi \cup \{(\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} : \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{\sigma(a)} \to \sigma(a)) \mid (\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} : \overline{a} \to a) \in \overline{\lambda}\}
\end{array}
$$

The overall algorithm is nondeterministic. The function **Unify** may return finitely many times as there may be multiple solutions for a constraint set. A local solution for class `C` may not be compatible with the constraints generated for a subsequent class. In this case, we have to backtrack to `C` and proceed to the next local solution; if thats fail we have to backtrack further to an earlier class.

4.1 Type inference for a program

Type inference processes a program one class at a time. To do so, it must be possible to order the classes such that early classes never call methods in later classes. As an example, Figure 12 shows a program that is acceptable in FGJ, but rejected by FGJ-GT because the methods `m1` and `m2` are mutually recursive across class boundaries. There is no order in which classes `C1` and `C2` can be processed.

Figure 13 contains a program acceptable to both FGJ-GT and FGJ because the mutual recursion of methods `m1` and `m2` is taking place inside class `D2`. As `D2` invokes method `m` of `D1`, type inference must process `D1` before `D2`, which corresponds to the constraints imposed by the typing of FGJ-GT in Section 3.2.

We obtain a viable order for processing the class declarations by computing an approximate call graph based solely on method names. That is, if method `m` is used in `C3` and defined both in `C1` and `C2`, then `C1` and `C2` must both be processed before `C3`. In such a case, the use of `m` might be ambiguous so that type inference for class `C3` proposes more than one solution. Global type inference attempts to extend each partial solution to a solution for the whole program and backtracks if that fails.

$T, U ::= a \mid \mathtt{X} \mid N$	type variable, bounded type parameter, or type
$N ::= \mathtt{C}\texttt{<}\overline{T}\texttt{>}$	class type (with type variables)
$sc ::= T \lessdot U \mid T \doteq U$	simple constraint: subtype or equality
$oc ::= \{\{\overline{sc_1}\}, \ldots, \{\overline{sc_n}\}\}$	or-constraint
$c ::= sc \mid oc$	constraint
$C ::= \{\overline{c}\}$	constraint set
$\lambda ::= \mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>.m} : \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\ \overline{T} \to T$	method type assumption
$\eta ::= \mathtt{x} : T$	parameter assumption
$\Pi ::= \mathtt{\Pi} \cup \overline{\lambda}$	method type environment
$\Theta ::= (\Pi; \overline{\eta})$	

Figure 14 Syntax of constraints and type assumptions.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{FJType}(\mathtt{\Pi}, \texttt{class C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{> extends N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\,\overline{\mathtt{M}}\}) =} \\
\textbf{let} & \overline{a}_{\mathtt{m}} \text{ be fresh type variables for each } \mathtt{m} \in \overline{\mathtt{M}} \\
& \overline{\lambda}_o = \{\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>.m} : \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\ \overline{\mathtt{T}} \to a_{\mathtt{m}} \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \mathtt{\Pi}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}\} \\
& C_o = \{a_{\mathtt{m}} \lessdot \mathtt{T} \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \mathtt{\Pi}) = \texttt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\texttt{>}\overline{\mathtt{T}} \to \mathtt{T}\} \\
& \overline{\lambda}' = \{(\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>.m} : \overline{a} \to a_{\mathtt{m}}) \mid \mathtt{m} \in \overline{\mathtt{M}}, \mathit{mtype}(\mathtt{m}, \mathtt{N}, \mathtt{\Pi}) \text{ not defined}, \overline{a} \text{ fresh}\} \\
& C_m = \{\{a_{\mathtt{m}} \lessdot \texttt{Object}, \overline{a} \lessdot \overline{\texttt{Object}}\} \mid (\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>.m} : \overline{a} \to a_{\mathtt{m}}) \in \overline{\lambda}'\} \\
& \Pi = \mathtt{\Pi} \cup \overline{\lambda}' \cup \overline{\lambda}_o \\
\textbf{in} & (\Pi, C_o \cup C_m \cup \bigcup_{\mathtt{m} \in \overline{\mathtt{M}}} \textbf{TYPEMethod}(\Pi, \mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}, \mathtt{m}))
\end{array}
$$

Figure 15 Constraint generation for classes.

4.2 Constraint generation

Figure 14 defines the syntax of constraints. We extend types with *type variables* ranged over by a. A constraint is either a simple constraint sc or an or-constraint oc, which is a set of sets of simple constraints. An or-constraint represents different alternatives, similar to an intersection type, and cannot be nested. The output of constraint generation is a set of constraints C, which can hold simple constraints as well as or-constraints.

Figure 15 contains the algorithm **FJType** to generate constraints for classes. Its input consists of the method type environment $\mathtt{\Pi}$ of the previously checked classes. It distinguishes between overriding and non-overriding method definitions. The former are recognized by successful lookup of their type using *mtype*. We set up the method type assumptions accordingly and generate a constraint between the inferred return type $a_{\mathtt{m}}$ and the one of the overridden method to allow for covariant overriding. Constraints for the latter methods are generated with all fresh type variables for the argument and result types.

Constraint generation alternates with constraint solving: After generating constraints with **FJType**, we solve them to obtain one or more candidate extensions for the method type environment $\mathtt{\Pi}$. Next, we pick a candidate and continue with the next class until all classes are checked and we have an overall method type environment. Otherwise, we backtrack to check the next candidate.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{TYPEMethod}(\Pi, \texttt{C<}\overline{\texttt{X}}\texttt{>},\ \texttt{m}(\overline{\texttt{x}})\{\texttt{return e};\}) =} \\
 & \textbf{let} & \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{T} \to T = \Pi(\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m}) \\
 & & (R, C) = \textbf{TYPEExpr}(\Pi; \{\texttt{this} : \texttt{C<}\overline{\texttt{X}}\texttt{>}\} \cup \{\overline{\texttt{x}} : \overline{T}\}), \texttt{e}) \\
 & \textbf{in} & C \cup \{R \lessdot T\}
\end{array}
$$

The **TYPEMethod** function for methods calls the **TYPEExpr** function with the return expression. It adds the assumptions for `this` and for the method parameters to the global assumptions before passing them to **TYPEExpr**.

In the following we define the **TYPEExpr** function for every possible expression: $\textbf{TYPEExpr} : \Theta \times \texttt{Expression} \to T \times C$

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

When we encounter a field `e.f`, we consider all classes `C` that define field `f` and impose an or-constraint that covers all alternatives: the type R of the expression `e` must be a subtype of a generic instance of `C` and the return type must be the corresponding field type.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e.f}) =} \\
 & \textbf{let} & (R, C_R) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}) \\
 & & a \text{ fresh} \\
 & & c = oc\{\{R \lessdot \texttt{C<}\overline{a}\texttt{>}, a \doteq [\overline{a}/\overline{\texttt{X}}]\texttt{T}, \overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}} \mid \overline{a} \text{ fresh}\} \\
 & & \qquad\qquad \mid \texttt{T f} \in \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>}\{\overline{\texttt{T}}\ \overline{\texttt{f}}; \ldots\}\} \\
 & \textbf{in} & (a, (C_R \cup \{c\}))
\end{array}
$$

We treat method calls in a similar way. We impose an or-constraint that considers a generic instance of a method type in a class providing that method (with the same number of parameters). Each choice imposes a subtyping constraint on the receiver type R as well as subtyping constraints on the argument types $\overline{R}$. Moreover, we need to check that the subtyping constraints of the method type are obeyed by instantiating them accordingly.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e.m}(\overline{\texttt{e}})) =} \\
 & \textbf{let} & (R, C_R) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}) \\
 & & \forall \texttt{e}_i \in \overline{\texttt{e}} : (R_i, C_i) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}_i) \\
 & & a \text{ fresh} \\
 & & c = oc\{\{\ R \lessdot \texttt{C<}\overline{a}\texttt{>}, a \doteq [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]T, \overline{R} \lessdot [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]\overline{T}, \\
 & & \qquad\quad \overline{b} \lessdot [\overline{b}/\overline{\texttt{Y}}][\overline{a}/\overline{\texttt{X}}]\overline{\texttt{P}}, \overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}} \mid \overline{a}, \overline{b} \text{ fresh}\} \\
 & & \qquad \mid (\texttt{C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>.m} : \texttt{<}\overline{\texttt{Y}} \lhd \overline{\texttt{P}}\texttt{>}\ \overline{T} \to T) \in \Pi\} \\
 & \textbf{in} & (a, (C_R \cup \bigcup_i C_i \cup \{c\}))
\end{array}
$$

The `new`-expression is comparatively simple. Starting from a generic instance of the class type, we calculate the types $\overline{T}$ of the fields, impose subtyping constraints on the constructor argument $\overline{R}$, and check the subtyping constraints of the class.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{new C}(\overline{\texttt{e}})) =} \\
 & \textbf{let} & \forall \texttt{e}_i \in \overline{\texttt{e}} : (R_i, C_i) = \textbf{TYPEExpr}((\Pi; \overline{\eta}), \texttt{e}_i) \\
 & & \overline{a} \text{ fresh} \\
 & & \mathit{fields}(\texttt{C<}\overline{a}\texttt{>}) = \overline{T}\ \overline{\texttt{f}} \\
 & & C = \{\overline{R} \lessdot \overline{T}\} \cup \{\overline{a} \lessdot [\overline{a}/\overline{\texttt{X}}]\overline{\texttt{N}}\} \qquad \text{where } \texttt{class C<}\overline{\texttt{X}} \lhd \overline{\texttt{N}}\texttt{>}\{\ldots\} \\
 & \textbf{in} & (\texttt{C<}\overline{a}\texttt{>}, C \cup \bigcup_i C_i)
\end{array}
$$

For cast expressions, we ignore the return type and pass on the constraints for the subexpression. We return the target type of the cast.

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

▶ **Example 6.** To illustrate the constraint generation step we will apply it to the program depicted in figure 1b. First the **FJType** function assigns the fresh type variable f to the parameter `fst`. Afterwards the **TYPEExpr** function is called on the return expression of the `setfst` method. The local variable `fst` does not emit any constraints. For the `this.snd` part of the expression the **TYPEExpr** function returns an or-constraint:

$$\begin{aligned}\overline{c}_1 &= \textbf{TYPEExpr}((\Pi;\overline{\eta}),\mathtt{this.snd}) \\ &= (b, oc(\{(\mathtt{Pair<X,Y>} \lessdot \mathtt{Pair<}w,y\mathtt{>}),(b \doteq y),(w \lessdot \mathtt{Object}),(y \lessdot \mathtt{Object})\}))\end{aligned}$$

This constraint is merged with the constraints generated by the `new Pair` constructor call:

$$\begin{aligned}&\textbf{TYPEExpr}((\Pi;\overline{\eta}),\mathtt{new\ Pair(fst,\ this.snd)}) \\ &\quad = (\mathtt{Pair<}d,e\mathtt{>},\{(f \lessdot d),(b \lessdot e),(d \lessdot \mathtt{Object}),(e \lessdot \mathtt{Object})\} \cup \overline{c}_1)\end{aligned}$$

5 Constraint Solving

This section describes the **Unify** algorithm which is used to find solutions for the constraints generated by **FJType**.

It first attempts to transform a constraint set into solved form and reads off a solution in the form of a substitution.

▶ **Definition 7** (Solved form). *A set C of constraints is in solved form if it only contains constraints of the following form:*

1. $a \lessdot b$
2. $a \doteq b$,
3. $a \lessdot \mathtt{C<}\overline{T}\mathtt{>}$,
4. $a \doteq \mathtt{C<}\overline{T}\mathtt{>}$, *with* $a \notin \overline{T}$.

In case 3 and 4 the type variable a does not appear on the left of another constraint of the form 3 or 4.

For brevity, we write $a_0 \lessdot^* a_n$ for a non-empty chain of subtyping constraints between type variables $a_0 \lessdot a_1, a_1 \lessdot a_2, \ldots, a_{n-1} \lessdot a_n$ where $n > 0$.

5.1 Algorithm Unify(C, Δ)

The input of the algorithm is a set of constraints C and a type environment Δ. The type environment binds the generic type variables $\overline{\mathtt{X}}$ to their upper bounds. It is used in invocations of the subtyping judgment.

The treatment of the generic class variables $\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}$ deserves some explanation. The algorithm must not substitute for these variables. Instead it treats them like parameterless abstract classes $\mathtt{X_i<>}$ which are subtypes of their respective $\mathtt{N}_i$ (where the variable name $\mathtt{X}_i$ is now treated like a class name). Example 8 illustrates this approach.

The first step of the algorithm eliminates or-constraints from constraint set C. To do so, we consider all combinations of selecting simple constraints from or-constraints in C. In general, we have that $C = \{\overline{sc}, oc_1, \ldots, oc_n\}$ and we execute the remaining steps for all $C' = \{\overline{sc}\} \cup \{\overline{sc}_1\} \cup \cdots \cup \{\overline{sc}_n\}$ where $\overline{sc}_i \in oc_i$.

Step 1. We apply the rules in Figures 16 and 17 exhaustively to C'.

Step 2. At this point, all constraints $sc \in C'$ are either in solved form or one of the following cases applies:

1. $\{\texttt{C<}\overline{T}\texttt{>} \lessdot \texttt{D<}\overline{U}\texttt{>}\} \subseteq C'$ where $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$ (roughly, `C` cannot be a subtype of `D`) – in this case C' has no solution;
2. $\{a \lessdot \texttt{C<}\overline{T}\texttt{>}, a \lessdot \texttt{D<}\overline{V}\texttt{>}\} \subseteq C'$ where $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$ and $\forall \overline{\texttt{X}}, \overline{\texttt{N}} \colon \Delta \not\vdash \texttt{D<}\overline{\texttt{X}}\texttt{>} <: \texttt{C<}\overline{\texttt{N}}\texttt{>}$ (roughly, `C` and `D` are not subtype-related) – in this case C' has no solution; or
3. $\{\texttt{C<}\overline{T}\texttt{>} \lessdot b\} \subseteq C'$.

The last case is a lower bound constraint which is embraced by Scala, but which is not legal in FGJ (nor in Java). As we insist on inferring a type, we have to find a concrete instance for $\texttt{C<}\overline{T}\texttt{>}$. To do so, we generate an or-constraint from each lower bound constraint and its corresponding upper bound constraint (using upper bound `Object` if no such constraint exists) as follows:

$$
\begin{aligned}
\mathit{expandLB}(\texttt{C<}\overline{T}\texttt{>} \lessdot b, b \lessdot \texttt{D<}\overline{U}\texttt{>}) = \{\{b \doteq [\overline{T}/\overline{\texttt{X}}]\texttt{N}\} \mid \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{N}, \Delta \vdash \texttt{N} <: \texttt{D<}\overline{\texttt{P}}\texttt{>}\}\\
\text{where } \overline{\texttt{P}} \text{ is determined by } \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{P}}\texttt{>} \text{ and } [\overline{T}/\overline{\texttt{X}}]\overline{\texttt{P}} = \overline{U}
\end{aligned}
$$

This constraint replaces the lower and upper bound constraint from which it was generated.

A lower bound may also be implied by a constraint set with constraints of the form $C_{ab} = a \lessdot \texttt{C<}\overline{T}\texttt{>}, a \lessdot^* b$. In this case $\texttt{C<}\overline{T}\texttt{>}$ must either be a upper or lower bound for b. We implement it by *expandLB*, which adds a lower bound constraint for b and also adding a upper bound to b. While C_{ab} remains in the constraint set: $\mathit{expandLB}(\texttt{C<}\overline{T}\texttt{>} \lessdot b, b \lessdot \texttt{D<}\overline{U}\texttt{>}) \ \cup \ \{b \lessdot \texttt{C<}\overline{T}\texttt{>}\}$

Now we are in a similar situation as before. Our current constraint set C' is a mix of simple constraints and or-constraints and, again, we consider all (simple) constraint sets C'' that arise as combinations of selecting simple constraints from C'.

Step 3. We apply the rule (subst) exhaustively to C'':

$$
\text{(subst)} \quad \frac{C \cup \{a \doteq T\}}{[T/a]C \cup \{a \doteq T\}} \quad a \text{ occurs in } C \text{ but not in } T
$$

We fail if we find any $a \doteq T$ such that a occurs in T.

Step 4. If C'' has changed from applying (subst), we continue with C'' from step 1.

Step 5. Otherwise, C'' is in solved form and it remains to eliminate subtyping constraints between variables by exhaustive application of rule (sub-elim) and (erase) (see Figure 17). Applying this rule does not affect the solve form property.

$$
\text{(sub-elim)} \quad \frac{C \cup \{a \lessdot b\}}{[a/b]C \cup \{b \doteq a\}}
$$

Step 6. We finish by generating a solving substitution from the remaining $\doteq$-constraints and generic variable declarations from the remaining $\lessdot$-constraints. Let $C'' = C_{\doteq} \cup C_{\lessdot}$ such that $C_{\doteq}$ contains only $\doteq$-constraints and $C_{\lessdot}$ contains only $\lessdot$-constraints. Now $C_{\lessdot} = \{\overline{a} \lessdot \overline{N}\}$ and choose some fresh generic variables $\overline{\texttt{Y}}$ of the same length as $\overline{a}$. We can read off the substitution σ from $C_{\doteq}$ where we need to substitute the generic variables for the type variables. We obtain the generic variable declarations directly from $C_{\lessdot}$ using the same generic variable substitution. We need not apply σ here because we applied (subst) exhaustively in Step 3.

$$
\begin{aligned}
&\sigma = \{b \mapsto [\overline{\texttt{Y}}/\overline{a}]T \mid (b \doteq T) \in C_{\doteq}\} \cup \{\overline{a} \mapsto \overline{\texttt{Y}}\} \cup \{b \mapsto \texttt{X} \mid (b \lessdot \texttt{X}) \in C_{\lessdot}\},\\
&\gamma = \{\texttt{Y} \lhd [\overline{\texttt{Y}}/\overline{a}]N \mid (a \lessdot N) \in C_{\lessdot}\}
\end{aligned}
$$

We return the pair (σ, γ).

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

$$\text{(adopt)}\quad \frac{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, b \lessdot^* a,\, b \lessdot \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{a \lessdot \texttt{C<}\overline{T}\texttt{>},\, b \lessdot^* a,\, b \lessdot \texttt{D<}\overline{U}\texttt{>},\, b \lessdot \texttt{C<}\overline{T}\texttt{>}\}}$$

$$\text{(adapt)}\quad \frac{C \cup \{\texttt{C<}\overline{T}\texttt{>} \lessdot \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{\texttt{D<}[\overline{T}/\overline{\texttt{X}}]\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}} \quad \Delta \vdash \texttt{C<}\overline{\texttt{X}}\texttt{>} <: \texttt{D<}\overline{\texttt{N}}\texttt{>}$$

$$\text{(reduce)}\quad \frac{C \cup \{\texttt{D<}\overline{T}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}}{C \cup \{\overline{T} \doteq \overline{U}\}}$$

$$\text{(equals)}\quad \frac{C \cup \{a_1 \lessdot a_2,\, a_2 \lessdot a_3, \ldots, a_n \lessdot a_1\}}{C \cup \{a_1 \doteq a_2,\, a_2 \doteq a_3, \ldots\}} \quad n > 0$$

Figure 16 Reduce and adapt rules.

$$\text{(erase)}\quad \frac{C \cup \{a \doteq a\}}{C}$$

$$\text{(swap)}\quad \frac{C \cup \{N \doteq a\}}{C \cup \{a \doteq N\}}$$

Figure 17 Erase and swap rules.

▶ **Example 8.** To illustrate our treatment of generic variables, we consider a typical case involving the (adapt) rule from Figure 16.

Consider $C = \{\texttt{X} \lessdot \texttt{D<}\overline{U}\texttt{>}\}$ and let $\texttt{X}<: \texttt{C<}\overline{\texttt{T}}\texttt{>} \in \Delta$ be the bound for `X`.

The side condition of the rule (adapt) asks for some $\overline{\texttt{N}}$ such that $\Delta \vdash \texttt{X}<: \texttt{D<}\overline{\texttt{N}}\texttt{>}$, i.e., "is there a way that `X` can be a subtype of `D`?"

By inversion of subtyping and transitivity, this judgment holds if $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}<: \texttt{D<}\overline{\texttt{N}}\texttt{>}$ holds.

Hence, applying (adapt) to C yields $\{\texttt{D<}\overline{\texttt{N}}\texttt{>} \doteq \texttt{D<}\overline{U}\texttt{>}\}$. The substitution in the rule is empty because `X` is considered a parameterless type.

The remaining rules work similarly. In particular, different variables $\texttt{X} \neq \texttt{Y}$ give rise to different (abstract) classes. For example, the (reduce) rule removes the constraint $\texttt{X} \doteq \texttt{X}$, but it does *not* apply to $\texttt{X} \doteq \texttt{Y}$. Rather, an equation like this renders the constraint set unsolvable.

▶ **Example 9.** To see that the algorithm is able to infer bounded generic types, we consider the example from the introduction (Figure 1a). The `setfst` method obtains a new generic type variable `Z`. We clarify this via a counterexample. Let's assume in the example in Figure 1b that there is an additional method call `setfst(new Integer())` inside the class `Pair`. This call would cause the method `setfst` to obtain type `Pair<Integer, Y> setfst(Integer fst)`, which corresponds to the typing rules (c.f. `GT-CLASS` in Figure 9): Inside the same class methods cannot be used in a polymorphic way. We have to make this restriction to avoid polymorphic recursion.

6 Properties of Unify

▶ **Theorem 10** (Soundness). *If* $\textbf{\textit{Unify}}(C, \Delta) = (\sigma, \overline{\mathtt{Y}} \lessdot \overline{\mathtt{P}})$*, then* σ *is unifier of* $(C, \Delta \cup \{\overline{\mathtt{Y}} <: \overline{\mathtt{P}}\})$.

▶ **Theorem 11** (Completeness). $\textbf{\textit{Unify}}(C, \Delta)$ *calculates the set of general unifiers for* (C, Δ).

A set of general unifiers can provide any unifier as a substitution instance of one of its members.

▶ **Definition 12** (Set of general unifiers). *Let* C *be a set of constraints and* Δ *a type environment.*

A set of unifiers M *for* (C, Δ) *is called* set of general unifiers *if for any unifier* ω *for* (C, Δ) *there is some unifier* $\sigma \in M$ *and a substitution* λ *such that* $\omega = \lambda \circ \sigma$.

7 Soundness, completeness and complexity of type inference

After showing that type unification is sound and complete, we can now show that type inference **FJTypeInference** also is sound and complete.

▶ **Theorem 13** (Soundness). *For all* Π, $\mathtt{L}$, Π', $\textbf{\textit{FJTypeInference}}(\Pi, L) = \Pi'$ *implies* $\Pi \vdash L : \Pi'$.

▶ **Theorem 14** (Completeness). *For all* Π, $\mathtt{L}$, Π', $\Pi \vdash L : \Pi'$ *implies there is a* Π'' *with* $\textbf{\textit{FJTypeInference}}(\Pi, L) = \Pi''$, $\Pi \vdash L : \Pi''$*, and the types of* Π' *are instances of* Π''.

▶ **Theorem 15** (NP-Hardness). *The type inference algorithm for typeless Featherweight Java is NP-hard.*

▶ **Theorem 16** (NP-Completeness). *The type inference algorithm for typeless Featherweight Java is NP-Complete.*

8 Related Work

8.1 Formal models for Java

There is a range of formal models for Java. Flatt et al [7] define an elaborate model with interfaces and classes and prove a type soundness result. They do not address generics. Igarashi et al [11] define Featherweight Java and its generic sibling, Featherweight Generic Java. Their language is a functional calculus reduced to the bare essentials, they develop the full metatheory, they support generics, and study the type erasing transformation used by the Java compiler. MJ [4] is a core calculus that embraces imperative programming as it is targeted towards reasoning about effects. It does not consider generics. Welterweight Java [17] and OOlong [5] are different sketches for a core language that includes concurrency, which none of the other core languages considers.

We chose to base our development on FGJ because it embraces a relevant subset of Java without including too much complexity (e.g., no imperative features, no interfaces, no concurrency). It seems that results for FGJ are easily scalable to full Java. We leave the addition of these feature to future work, as we see our results on FGJ as a first step towards a formalized basis for global type inference for Java.

8.2 Type inference

Some object-oriented languages like Scala, C#, and Java perform *local* type inference [16,18]. Local type inference means that missing type annotations are recovered using only information from adjacent nodes in the syntax tree without long distance constraints. For instance, the type of a variable initialized with a non-functional expression or the return type of a method can be inferred. However, method argument types, in particular for recursive methods, cannot be inferred by local type inference.

Milner's algorithm $\mathcal{W}$ [15] is the gold standard for global type inference for languages with parametric polymorphism, which is used by ML-style languages. The fundamental idea of the algorithm is to enforce type equality by many-sorted type unification [14,25]. This approach is effective and results in so-called principal types because many-sorted unification is unitary, which means that there is at most one most general result.

Plümicke [20] presents a first attempt to adopt Milner's approach to Java. However, the presence of subtyping means that type unification is no longer unitary, but still finitary. Thus, there is no longer a single most general type, but any type is an instance of a finite set of maximal types (for more details see Section 8.3). Further work by the same author [22,24], refines this approach by moving to a constraint-based algorithm and by considering lambda expressions and Scale-like function types. In Plümicke's work there is no formal definition of the type system as a basis of the type inference algorithm. One contribution of this paper is a formal definition of the underlying type system.

We rule out polymorphic recursion because its presence makes type inference (but not type checking: see FGJ) undecidable. Henglein [9] as well as Kfoury et al [12] investigate type inference in the presence of polymorphic recursion. They show that type inference is reducible to semi-unification, which is undecidable [13]. However, the undecidability of this problem apparently does not matter much in practice [6].

Ancona, Damiani, Drossopoulou, and Zucca [1] consider polymorphic byte code. Their approach is modular in the sense that it infers polymorphic structural types. As Java does not support structural types, their approach would have to be simulated with generated interfaces. Plümicke [23] follows this approach. Furthermore Ancona and coworkers do not consider generic classes.

8.3 Unification

We reduce the type inference problem to constraint solving with equality and subtype constraints. The procedure presented in Section 5 is inspired by polymorphic order-sorted unification which is used in logic programming languages with polymorphic order-sorted types [2,8,10,27].

Smolka's thesis [27] mentions type unification as an open problem. He gives an incomplete type inference algorithm for the logical language TEL. The reason for incompleteness is the admission of subtype relationships between polymorphic types of different arities as in $\texttt{List(a)} <: \texttt{myLi(a,b)}$. In consequence, the subtyping relation does not fulfill the ascending chain condition. For example, given $\texttt{List(a)} <: \texttt{myLi(a,b)}$, we obtain:

$$\texttt{List(a)} <: \texttt{myLi(a,List(a))} <: \texttt{myLi(a,myLi(a,List(a)))} <: \ldots$$

However, this subtyping chain exploits covariant subtyping, which does not apply to FGJ.

Smolka's algorithm also fails sometimes in the absence of infinite chains, although there is a unifier. For example, given $\texttt{nat} <: \texttt{int}$ and the set of subtyping constraints $\{\texttt{nat} \lessdot \texttt{a}, \texttt{int} \lessdot \texttt{a}\}$, it returns the substitution $\{\texttt{a} \mapsto \texttt{nat}\}$ generated from the first constraint encountered. This substitution is not a solution because $\{\texttt{int} \lessdot \texttt{nat}\}$ fails. However, $\{\texttt{a} \mapsto \texttt{int}\}$ is a unifier, which can be obtained by processing the constraints in a different order: from $\{\texttt{int} \lessdot \texttt{a}, \texttt{nat} \lessdot \texttt{a}\}$ the algorithm calculates the unifier $\{\texttt{a} \mapsto \texttt{int}\}$.

Hill and Topor [10] propose a polymorphically typed logic programming language with subtyping. They restrict subtyping to type constructors of the same arity, which guarantees that all subtyping chains are finite. In this approach a *most general type unifier (mgtu)* is defined as an upper bound of different principal type unifiers. In general, two type terms need not have an upper bound in the subtype ordering, which means that there is no mgtu in the sense of Hill and Topor. For example, given $\texttt{nat} <: \texttt{int}$, $\texttt{neg} <: \texttt{int}$, and the set of inequations $\{\texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a}\}$, the mgtu $\{\texttt{a} \mapsto \texttt{int}\}$ is determined. If the subtype ordering is extended by $\texttt{int} <: \texttt{index}$ and $\texttt{int} <: \texttt{expr}$, then there are three unifiers $\{\texttt{a} \mapsto \texttt{int}\}$, $\{\texttt{a} \mapsto \texttt{index}\}$, and $\{\texttt{a} \mapsto \texttt{expr}\}$, but none of them is an mgtu [10].

The type system of PROTOS-L [2] was derived from TEL by disallowing any explicit subtype relationships between polymorphic type constructors. Beierle [2] gives a complete type unification algorithm, which can be extended to the type system of Hill and Topor. They also prove that the type unification problem is finitary.

Given the declarations $\texttt{nat} <: \texttt{int}$, $\texttt{neg} <: \texttt{int}$, $\texttt{int} <: \texttt{index}$, and $\texttt{int} <: \texttt{expr}$, applying the type unification algorithm of PROTOS-L to the set of inequations $\{\texttt{nat} \lessdot \texttt{a}, \texttt{neg} \lessdot \texttt{a}\}$ yield three general unifiers $\{\texttt{a} \mapsto \texttt{int}\}$, $\{\texttt{a} \mapsto \texttt{index}\}$, and $\{\texttt{a} \mapsto \texttt{expr}\}$.

Plümicke [21] realized that the type system of TEL is related to subtyping in Java. In contrast to TEL, where the ascending chain condition does not hold, Java with wildcards violates the descending chain condition. For example, given `myLi<b,a>` $<:$ `List<a>` we find:

$$\ldots <: \texttt{myLi<?}\lhd\texttt{myLi<?}\lhd\texttt{List<a>,a>,a>} <: \texttt{myLi<?}\lhd\texttt{List<a>,a>} <: \texttt{List<a>}$$

Plümicke [21] solved the open problem of infinite chains posed by Smolka [27]. He showed that in any infinite chain there is a finite number of elements such that all other elements of the chain are instances of them. The resulting type unification algorithm can be used for type inference of Java 5 with wildcards [20]. As FGJ has no wildcards, we based our algorithm on an earlier work [19]. In contrast to that work, which only infers generic methods with unbounded types, our algorithm infers bounded generics. To this end, we do not expand constraints of the form $a \lessdot N$, where a is type variable and N is is a non-variable type, but convert them to bounded type parameters of the form `X extends N`. This change results in a significant reduction of the number of solutions of the type unification algorithm without restricting the generality of typings of FGJ-programs. Unfortunately, constraints of the form $N \lessdot a$ have to be expanded as FGJ (like Java) does not permit lower bounds for generic parameters. If lower bounds were permitted (as in Scala), the number of solutions could be reduced even further.

9 Conclusions

This paper presents a global type inference algorithm applicable to Featherweight Generic Java (FGJ). To this end, we define a language FGJ-GT that characterizes FGJ programs amenable to type inference: its methods carry no type annotations and it does not permit polymorphic recursion. This language corresponds to a strict subset of FGJ.

The inference algorithm is constraint based and is able to infer generalized method types with bounded generic types, as demonstrated with the example in Figure 1a.

In future work, we plan to extend FGJ-GT to a calculus with wildcards inspired by Wild FJ [29]. We also plan to extend the formal calculus with lambda expressions (cf. [3]), but using true function types in place of interface types.

References

1 Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic bytecode: compositional compilation for java-like languages. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 26–37. ACM, 2005. doi:10.1145/1040305.1040308.

2 Christoph Beierle. Type inferencing for polymorphic order-sorted logic programs. In *International Conference on Logic Programming*, pages 765–779, 1995.

3 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Venneri Betti. Java & lambda: A featherweight story. *Logical Methods in Computer Science*, 14(3:17):1–24, 2018.

4 G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, April 2003. doi:10.48456/tr-563.

5 Elias Castegren and Tobias Wrigstad. OOlong: an extensible concurrent object calculus. In Hisham M. Haddad, Roger L. Wainwright, and Richard Chbeir, editors, *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, pages 1022–1029. ACM, 2018. doi:10.1145/3167132.3167243.

6 Martin Emms and Hans Leiß. Extending the type checker of standard ML by polymorphic recursion. *Theor. Comput. Sci.*, 212(1-2):157–181, 1999. doi:10.1016/S0304-3975(98)00139-X.

7 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269. Springer, 1999. doi:10.1007/3-540-48737-9_7.

8 Michael Hanus. Parametric order-sorted types in logic programming. *Proc. TAPSOFT 1991*, LNCS(394):181–200, 1991.

9 Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993. doi:10.1145/169701.169692.

10 Patricia M. Hill and Rodney W. Topor. A Semantics for Typed Logic Programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

11 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.

12 A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993. doi:10.1145/169701.169687.

13 A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. The undecidability of the semi-unification problem. *Inf. Comput.*, 102(1):83–101, 1993. doi:10.1006/inco.1993.1003.

14 A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

15 Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.

16 Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. *Proc. 28th ACM Symposium on Principles of Programming Languages*, 36(3):41–53, 2001.

17 Johan Östlund and Tobias Wrigstad. Welterweight java. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116. Springer, 2010. doi:10.1007/978-3-642-13953-6_6.

18 Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 252–265, 1998.

19 Martin Plümicke. Type Unification in Generic–Java. In Michael Kohlhase, editor, *Proceedings of 18th International Workshop on Unification (UNIF'04)*, Cork, July 2004.

20 Martin Plümicke. Typeless Programming in Java 5.0 with Wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

21 Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

22 Martin Plümicke. More type inference in Java 8. In Andrei Voronkov and Irina Virbitskaite, editors, *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*, volume 8974 of *Lecture Notes in Computer Science*, pages 248–256. Springer, 2015.

23 Martin Plümicke. Structural type inference in java-like languages. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016.*, pages 109–113, 2016. URL: `http://ceur-ws.org/Vol-1559/paper09.pdf`.

24 Martin Plümicke and Andreas Stadelmeier. Introducing Scala-like function types into Java-TX. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 23–34, New York, NY, USA, 2017. ACM. `doi:10.1145/3132190.3132203`.

25 J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, January 1965.

26 Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, November 2003. Springer. `doi:10.1007/978-3-540-40018-9_19`.

27 Gert Smolka. *Logic Programming over Polymorphically Order-Sorted Types*. PhD thesis, Department Informatik, University of Kaiserslautern, Kaiserslautern, Germany, May 1989.

28 Andreas Stadelmeier, Martin Plümicke, and Peter Thiemann. Global type inference for Featherweight Generic Java, 2022. URL: `https://arxiv.org/abs/2205.08768`.

29 Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In Philip Wadler, editor, *Proceedings of FOOL 12*, Long Beach, California, USA, January 2005. ACM, School of Informatics, University of Edinburgh. URL: `http://homepages.inf.ed.ac.uk/wadler/fool/`.