Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Corrective Hints for Type Incorrect Generic Java Programs

Jurriaan Hage
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Joint work with Nabil El Boustani.

Department of Information and Computing Sciences, Universiteit Utrecht

January 18, 2010

The topic of our work

- How can we modify the type checking process of Java, so that implementations may give more informative error messages?

The topic of our work

- How can we modify the type checking process of Java, so that implementations may give more informative error messages?
  - Zooming in: only for generic method invocations
    A first step was published in (Boustani et al, PEPM '09)

The topic of our work

- How can we modify the type checking process of Java, so that implementations may give more informative error messages?
  - Zooming in: only for generic method invocations
    A first step was published in (Boustani et al, PEPM '09)
    - Zooming in: providing hints that may help resolve type errors.

In a larger context

- My agenda: Error Diagnosis Oriented Static Analysis
- Multiple languages, multiple paradigms, multiple analyses.
  - Java, Haskell, PHP, JavaScript
  - intrinsic type systems, security analysis, optimizing analyses with explicit annotations.
- Cross-fertilization: parametric polymorphism in Haskell prepares for genericity in Java.
- Sharing experience, heuristics and implementations between compilers
- Back to Java...

A selection of complications

- The (generics part of the) JLS is large and complicated
- The JLS is operational rather than declarative.
  - Hard on programmers, harder on compiler builders.
- Type error messages provided by major tools are not very informative
- Types not part of the program are constructed and appear in error messages
- Compilers follow the JLS slovenly
  - Similar issues, but dissimilar messages
  - Deviation from JLS gives programs that should not compile
- See (Boustani et al, PEPM '09) for illustrations.

The current paper

Make error messages more informative by providing hints as to how they may be resolved.

What is a hint?

The current paper

Make error messages more informative by providing hints as to how they may be resolved.

What is a hint?

A hint describes how the program may be changed so that the observed error does not anymore occur.

The current paper

Make error messages more informative by providing hints as to how they may be resolved.

What is a hint?

A hint describes how the program may be changed so that the observed error does not anymore occur.

Complication: we have to guess at the programmer's intentions.

The current paper

Make error messages more informative by providing hints as to how they may be resolved.

What is a hint?

A hint describes how the program may be changed so that the observed error does not anymore occur.

Complication: we have to guess at the programmer's intentions.

Danger: we may easily guess wrong.

The current paper

Make error messages more informative by providing hints as to how they may be resolved.

What is a hint?

A hint describes how the program may be changed so that the observed error does not anymore occur.

Complication: we have to guess at the programmer's intentions.

Danger: we may easily guess wrong.

A solution: weigh evidence in order to decide when to provide hints.

Implemented into Jastad EJC (Hedin et al.)

1. Examples

Example 1: exploit return type invariance


```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

javac:

```
cxt_heuristic/Test1.java:6:
<T>foo(Map<T,? super T>) in Test1 cannot be applied
to (Map<Number,Integer>)
```

ejc:

```
The method foo(Map<T, ? super T) in the type Test1
is not applicable for the arguments
(Map<Number,Integer>)
```

Example 1: exploit return type invariance


```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

ours:

```
cxt_heuristic/Test1.java:6
Method <T>foo(Map<T, ? super T>) of type
Test1 is not applicable for the argument of
type (Map<Number, Integer>), because:
  [*] The type Integer in Map<Number, Integer>
  on 5:9(5:21) is not a supertype of the inferred
  type for T: Number. However, replacing Number
  on 5:13 with Integer may solve the type conflict.
```

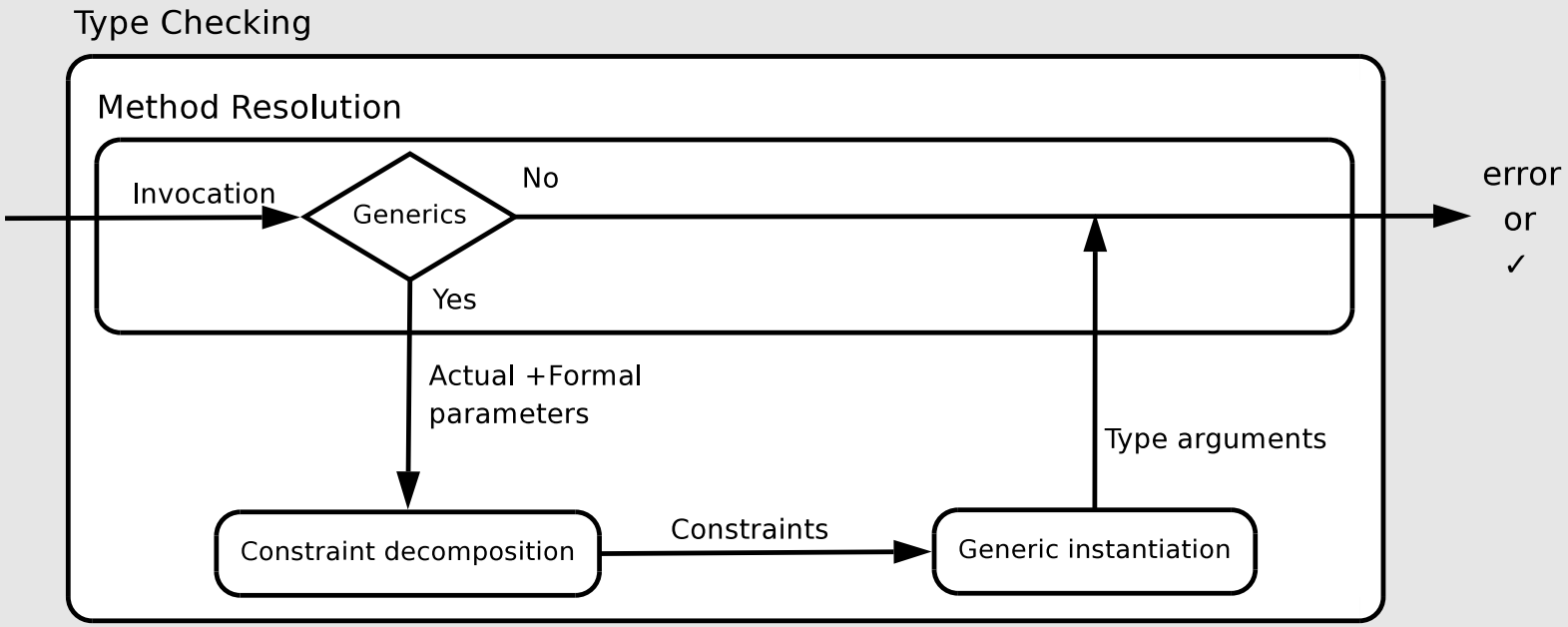
Example 2: satisfy subtype conflicts


```
<T> void foo(T a, T b,
             Map<? super T, ? super T> c) {}
...
Map<Number, Double> m = ...;
Number n = ...;
foo(1, n, m);
```

```
Test1.java:13
Method <T>foo(T, T, Map<? super T, ? super T>) of
type Test1 is not applicable to the arguments of
type(int, Number, Map<Number, Double>), because:
  [*] The type Double in Map<Number, Double> on
  11:9(11:21) is not a supertype of the inferred
  type for T: Number. However, replacing Double on
  11:21 with Number may solve the type conflict.
```

2. Adapting the type checking process

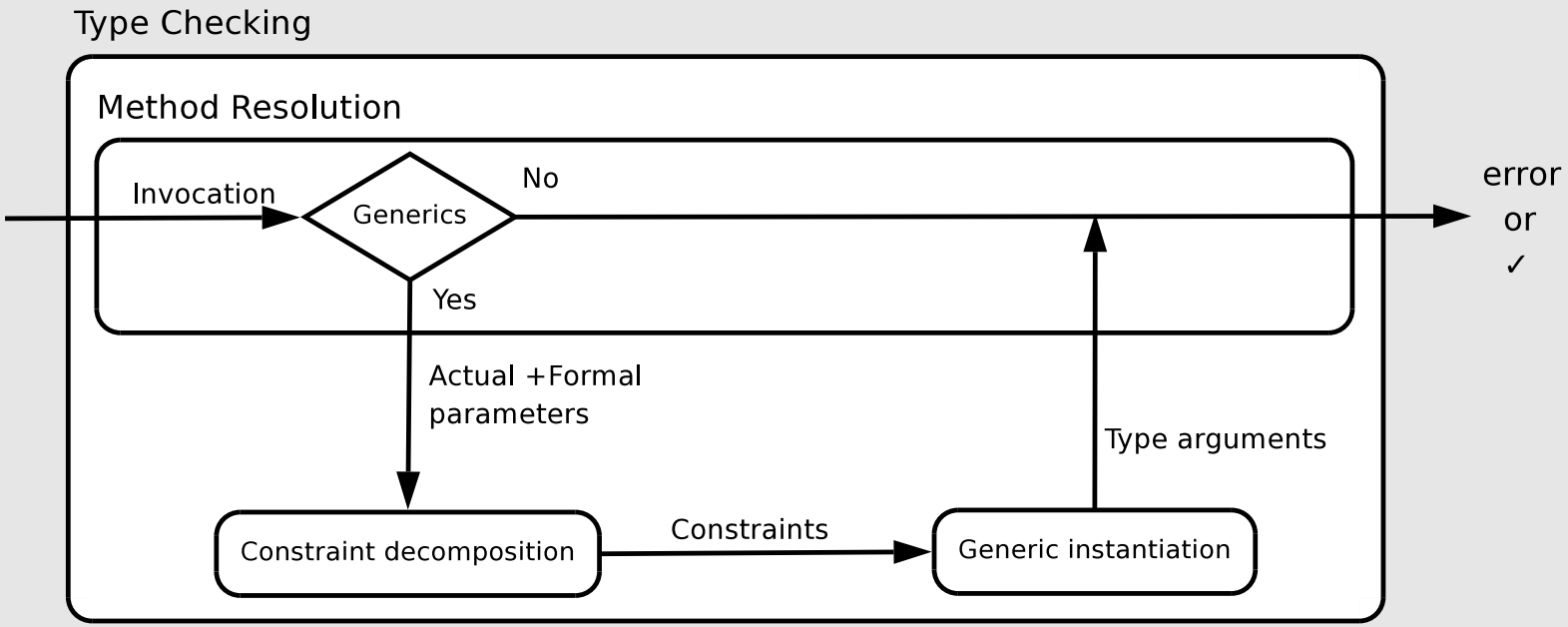
The Java type checking process



Method resolution determines a single, most specific method that the programmer may be calling.

- Multiset is reduced by applying various heuristics.
- Ambiguity, or lack of a fitting method: error message is returned.

The Java type checking process



Relate arguments to parameters via constraints:

`foo(new HashMap<Number, String>, new Integer(2))`

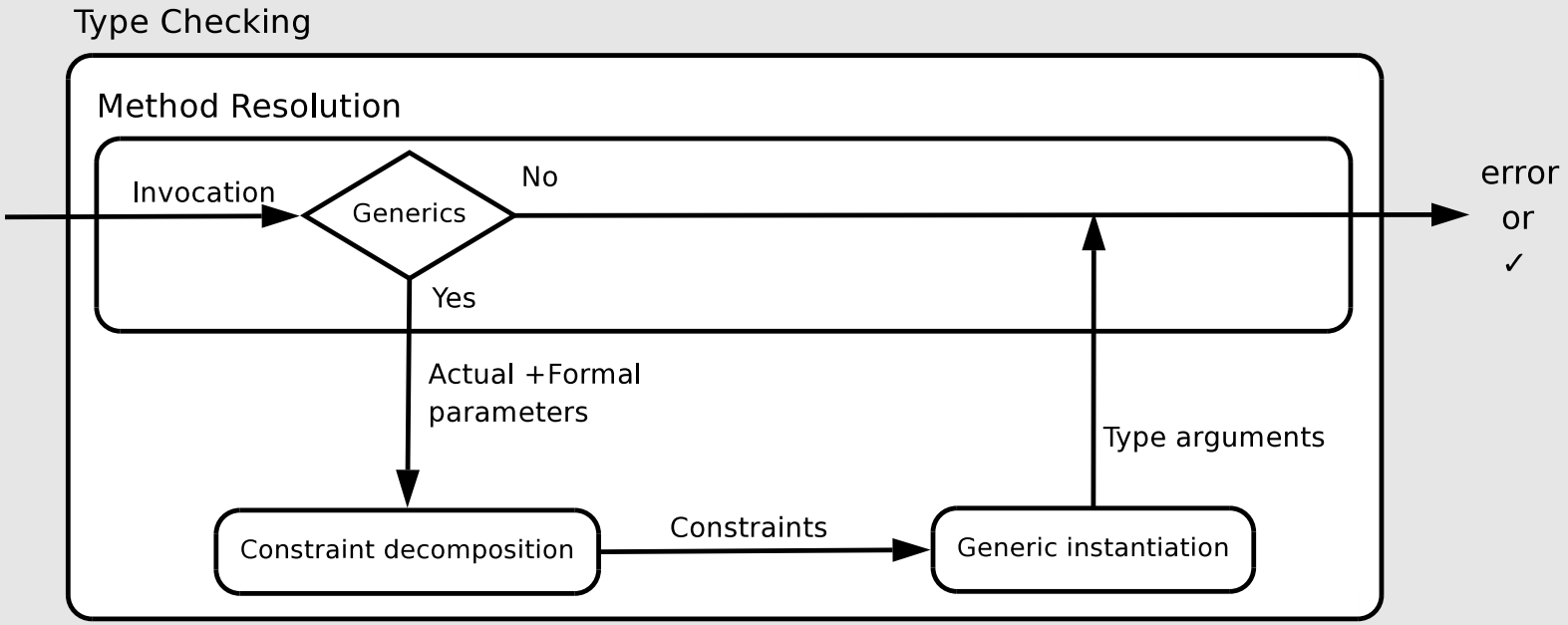
to call `void foo(HashMap<T,S extends T> map, S)`

gives

`HashMap<Number, String>` $<:$ `HashMap<T, S extends T>`

`Integer` $<:$ `S`

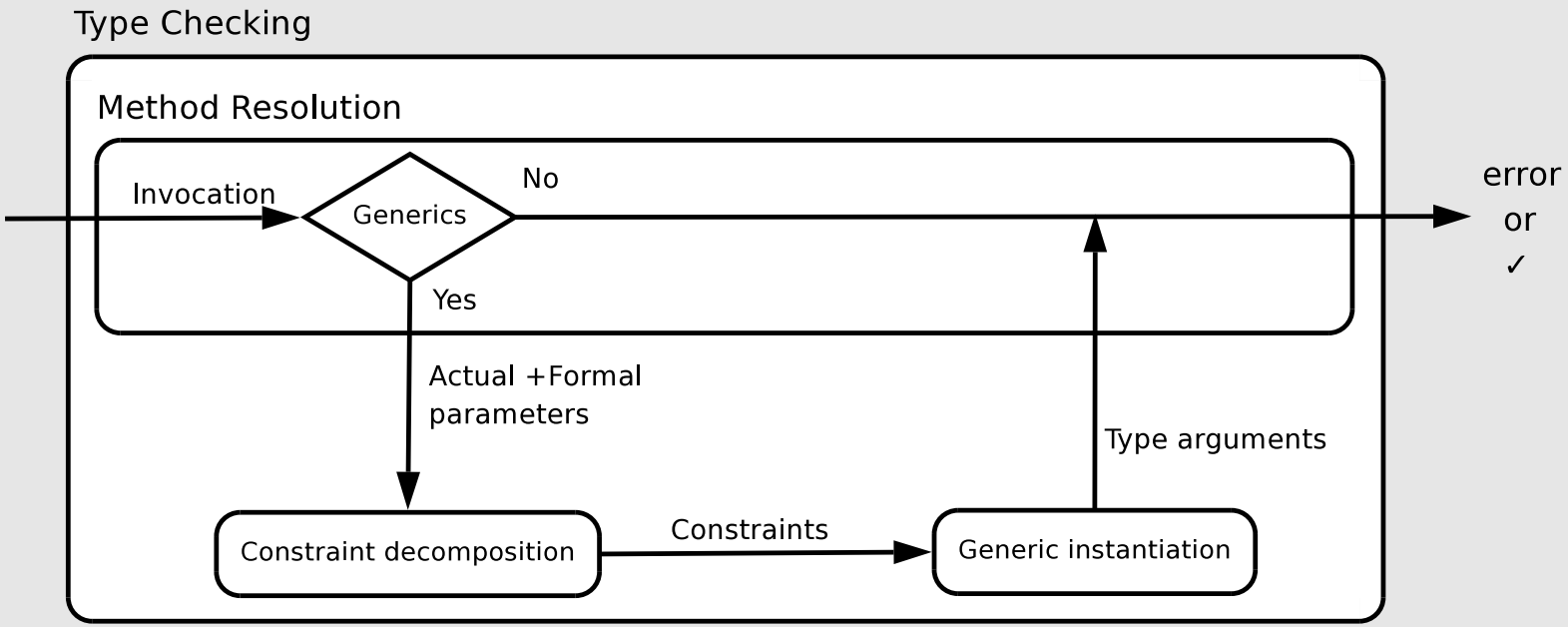
The Java type checking process



Decompose constraints: we get

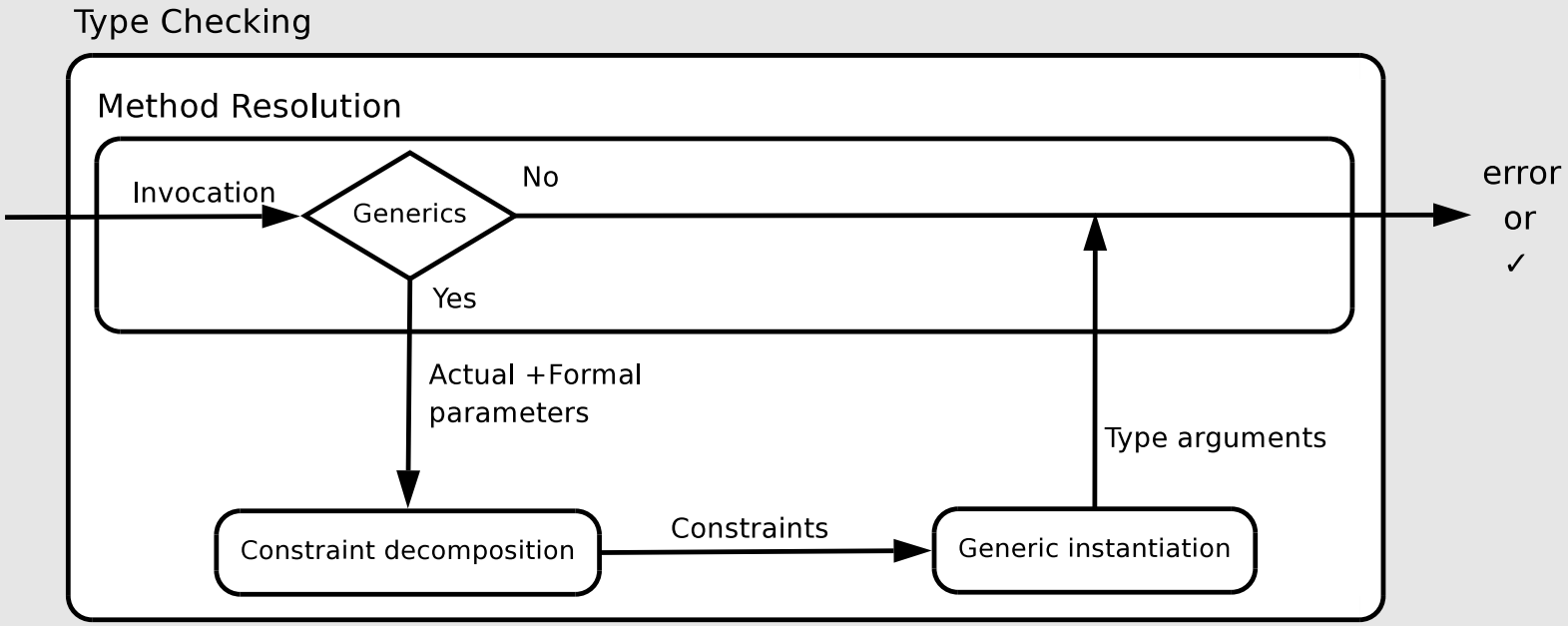
$$\{\mathtt{T} = \mathtt{Number}, \mathtt{S} = \mathtt{String}, \mathtt{S} <: \mathtt{T}, \mathtt{Integer} <: \mathtt{S}\}$$

The Java type checking process



Type inference/generic instantiation:
find "suitable" types for `T` and `S`, `T` = `Number` and `S` = `String`.

The Java type checking process



Type inference/generic instantiation:
find "suitable" types for `T` and `S`, `T` = `Number` and `S` = `String`.
Type checking:
subtype constraint `S` <: `T` and `Integer` <: `S` both fail.

Adaptations to the Java type checking process §2

- Principle 1: relax
  - Allow more than one candidate method.
  - Avoid taking decisions influenced by what is likely to be wrong. Erase generics!
- Principle 2: don't change, add
  - Error messages are constructed after type checking has failed.
- Principle 3: hold on (to constraints)
  - Message is based on the original constraints.
- Again, see (Boustani et al, PEPM '09)

Conflict of interest?

- deciding type correctness is easier with simplified constraints, but
- the original constraints are closer to the program.
  - easier to make informed decisions
  - error messages will make more sense
  - makes it possible to provide hints

Conflict of interest?

- deciding type correctness is easier with simplified constraints, but
- the original constraints are closer to the program.
  - easier to make informed decisions
  - error messages will make more sense
  - makes it possible to provide hints
- In practice, compiler builders choose one of the two approaches
- But why not do both?

How does it work, a stepwise description

- After a type error is discovered, we consider the original constraints anew.
- The constraints cannot be satisfied simultaneously.
- But which constraint(s) are most likely to be responsible?
- Based on that information we can
  - provide a suitable error message
  - try to come up with a hint that solves the problem.
- Various heuristics consider the set of constraints (independently) and try to find a suitable hint.
- An error manager decides which of these is best
  - Heuristics are currently ranked in a fixed order.
- and attaches it to the error message.

3. Heuristics

Heuristics

We add heuristics to the process to:

- Bias the error messages in the presence of enough evidence.
  - Ten say `T` should be `Number`, one says its `String`.
- Encapsulate expert knowledge about the kind of mistakes people make.
  - Not every programmer understands that or why collector classes are invariant, so if that knowledge solves the problem we can explicitly mention this.
- Heuristics decide which constraint is erroneous, but also suggest how to avoid generating that constraint in the next compile.
  - Suggest to replace `String` with `Number`.

In more detail: context type invariance and super Object

Example: exploit context type invariance (again) §3

```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

Works under the following circumstances

- method has a generic/parameterized return type
- that contains type variables
- method invocation is in an assignment context
- the assignment should give rise to an equality constraint
  - Here between `T` and `Integer`

Example heuristic: context type invariance §3

```
<T> List<T> foo(Map<T, ? super T> a){}
...
Map<Number, Integer> m = null;
List<Integer> ret = foo(m);
```

- The assignment gives concrete information about `T`.
- that we hope is correct.
- Verify that information is consistent with constraints derived from the method signature.
- What should change in the invocation to accommodate: `Map<Number, Integer>` $\Longrightarrow$ `Map<Integer, Integer>`
- The hint suggests this change: `However, replacing Number on 5:13 with Integer may solve the type conflict.`

Example: context type invariance (advanced) §3

```
<T, R, S extends Map<R, T>>
    Map<T, R> baz(T a, R b, S c) {}
Number t = ...;
Number r = ...;
Map<Double, Integer> ms = ...;
Map<Integer, Double> ret = baz(t, r, ms);
```

```
<omitted for reasons of space> because:
  [*] The type Map<Double, Integer> on 14:9 is
  not a subtype of S's upper bound Map<Number,
  Number> in 'S extends Map<R, T>'. However,
  replacing the types:
  - Number on 12:9 with Integer
  - Number on 13:9 with Double
  may solve the type conflict.
```

Example heuristic: super Object

- What can you do with a variable of the following type?

  `List<? super Object>`

- Any value can be added to the list: every type is a subtype of `Object`, so also of any supertype of `Object`.
- You can only safely read values of type `Object`.
- Therefore, the list behaves exactly like `List<Object>`.
- The use of `?` may make a program type incorrect.
- The heuristic
  - detects constraints that mention `? super Object`
  - replaces these parts with `Object`,
  - verifies that the constraint set becomes satisfiable.
- Similarly, `? extends X` where `X` is final can be dealt with.

Example: super Object


```
<T> void foo(Map<T, T> a) {}
...
Map<? super Object, ? super Object> m = ...;
foo(m);
```

```
Test1.java:6
Method <T>foo(Map<T, T>) of type Test1 is not
applicable to the argument of type
(Map<? super Object, ? super Object>), because:
  [*] '? super Object' may not always equal
  '? super Object' and invariance demands this.
  However, replacing
  - '? super Object' on 5:13
  - '? super Object' on 5:28
  with Object may solve the type conflict.
```

4. Wrapping up

Summary

- I have
  - sketched the type inference process,
  - discussed the use of heuristics,
  - illustrated by one or two example heuristics,
  - and gave examples of their effects.
- I omitted
  - majority heuristic for equality constraints (5.1)
  - confusing `super` and `extends` (5.3).
  - mismatches between `super` and `extends` (5.3.2)
  - majority heuristic for subtype conflicts (5.4)
  - warnings (5.5)
  - illustrated by many more examples
  - related work

Future work

- Become a (Generic) Java expert
- Thus far only (generic) invocations.
  - What about other constructs, like inner classes?
- More heuristics, also for the non-generic parts
- Expert and experimental validation.
- Collect programs: weird and normal.
  - Have any? Send them to jur@cs.uu.nl.
- Other combinations of subtyping and generics.
  - Johan Nordlander's Timber