Universiteit Utrecht

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

Improving Error Messages for Generic Java

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

March 31, 2009

The topic of our work

- How to modify the type checking process of Java,
- so that implementations give more informative error messages
- for generic method invocations

The topic of our work

- How to modify the type checking process of Java,
- so that implementations give more informative error messages
- for generic method invocations
- Complicated by complexity and size of Java Language Specification (JLS)
- Implementation as part of Jastad EJC (Hedin et al.)
- But first some motivational examples

Overview

Motivating examples

The type checking process

The new type checking process

Wrapping up

1. Motivating examples

Example: no explanation why


```
<T> void foo(Map<T,T> a){
  Map<Number, Integer> m1 = ...;
  foo(m1);
```

ejc:

```
1. ERROR in Listing1.java (at line 6)
    foo(m1);
The method foo(Map<T,T>) ... is not applicable
for the arguments (Map<Number,Integer>)
```

Example: no explanation why


```
<T> void foo(Map<T,T> a){
  Map<Number, Integer> m1 = ...;
  foo(m1);
```

ours:

```
Listing1.java:6
Method <T>foo(Map<T, T>) of type Listing1 is
  not applicable for the argument of type
  (Map<Number, Integer>), because:
  [*] The type variable T is invariant, but
  - Integer in Map<Number, Integer> on 5:9(5:21)
  - Number in Map<Number, Integer> on 5:9(5:13)
  are not the same type.
```

Example: exposes type checking artifacts §1

```
<T> void bar(Map<T, T> a) {
  Map<? extends Number, ? extends Number> m = null;
  bar(m);
}
```

javac:

```
Test1.java:20: cannot find symbol
symbol : method bar(Map<capture#954 of ? extends
 Number, capture#0 of ? extends Number>)
location: class Test1
  foo(m);
```

Example: exposes type checking artifacts §1

```
<T> void bar(Map<T, T> a) {
  Map<? extends Number, ? extends Number> m = null;
  bar(m);
}
```

ours:

```
Listing4.java:6
Method <T>bar(Map<T, T>) of type Test1 is not
  applicable for the argument of type
  Map<? extends Number, ? extends Number>, because:
  [*] The type variable T is invariant,
      but the type '? extends Number' is not.
```

Example: erroneous behaviour

Sun's JAVAC accepts the following, and similar programs:

```
<T extends Number> void foo(List<? super T> a)
...
List<String> x = ...
foo(x);
```

- Why is it wrong to accept this?
- `foo` should only work for lists of types that lie between `Number` and `Object`. Not for `String`.

Example: erroneous behaviour

Sun's JAVAC accepts the following, and similar programs:

```
<T extends Number> void foo(List<? super T> a)
...
List<String> x = ...
foo(x);
```

- Why is it wrong to accept this?
- `foo` should only work for lists of types that lie between `Number` and `Object`. Not for `String`.
- Why does it go wrong?
- Condition `T extends Number` is ignored.

Example: strange behaviour


```
<T extends Number>
  void foo(Map<? super T, ? super T> a)
...
Map<String, Number> m = ...;
foo(m);
```

ejc:

```
1. ERROR in Listing5.java (at line 10)
  foo(m);
Bound mismatch: The generic method foo(
 Map<? super T, ? super T>) of type Listing5 is not
 applicable for the arguments (Map<String,Number>).
 The inferred type String is not a valid substitute
 for the bounded parameter <T extends Number>
```

Example: strange behaviour


```
<T extends Number>
  void foo(Map<? super T, ? super T> a)
...
Map<Number, String> m = ...;
foo(m);
```

ejc:

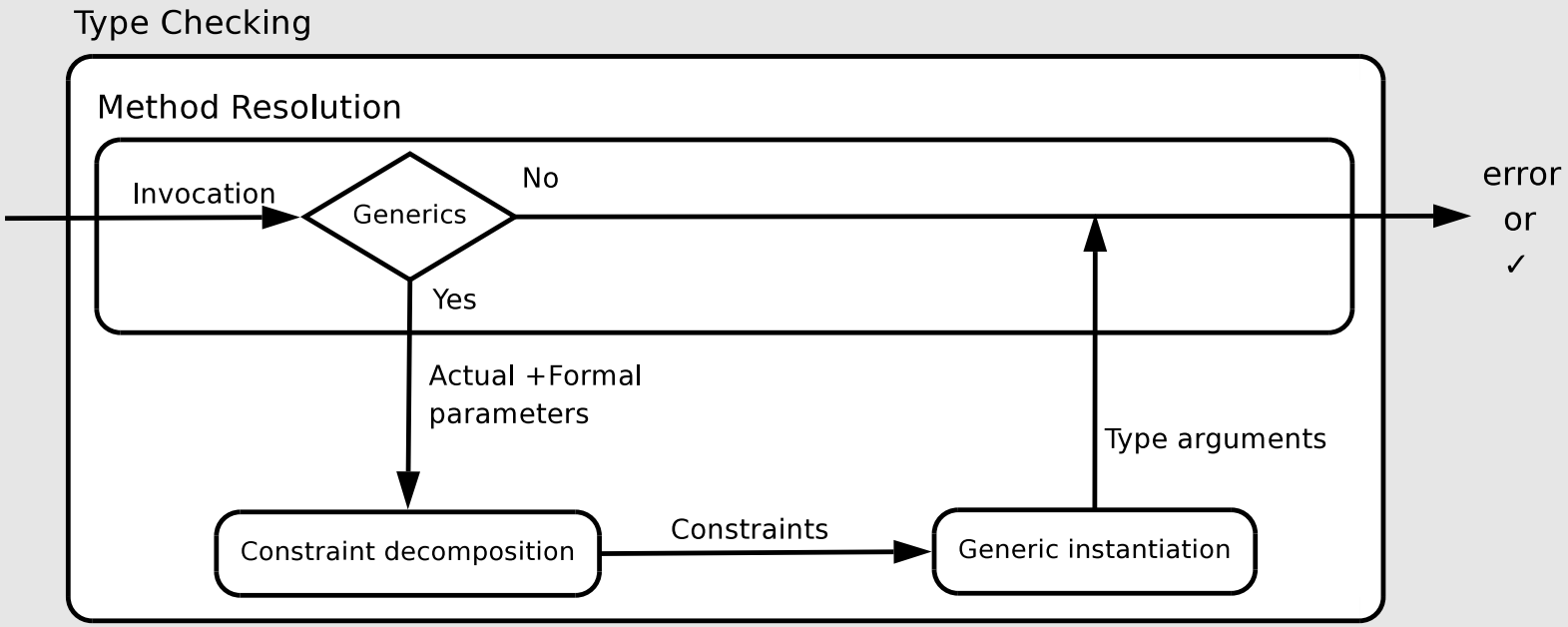
```
1. ERROR in Listing5.java (at line 10)
  foo(m);
The method foo(Map<? super T,? super T>) in the
 type Listing5 is not applicable for the arguments
 (Map<Number,String>)
```

A summary of complications

- The (generics part of the) JLS is large and complicated
- The JLS is more operational than declarative.
  - Hard on programmers, harder on compiler builders.
- Type error messages are not very informative
- Types not part of the program are constructed and appear in error messages
- And compilers follow the JLS slovenly
  - Similar problems, but dissimilar messages
  - Deviation sometimes gives programs that should not compile

2. The type checking process

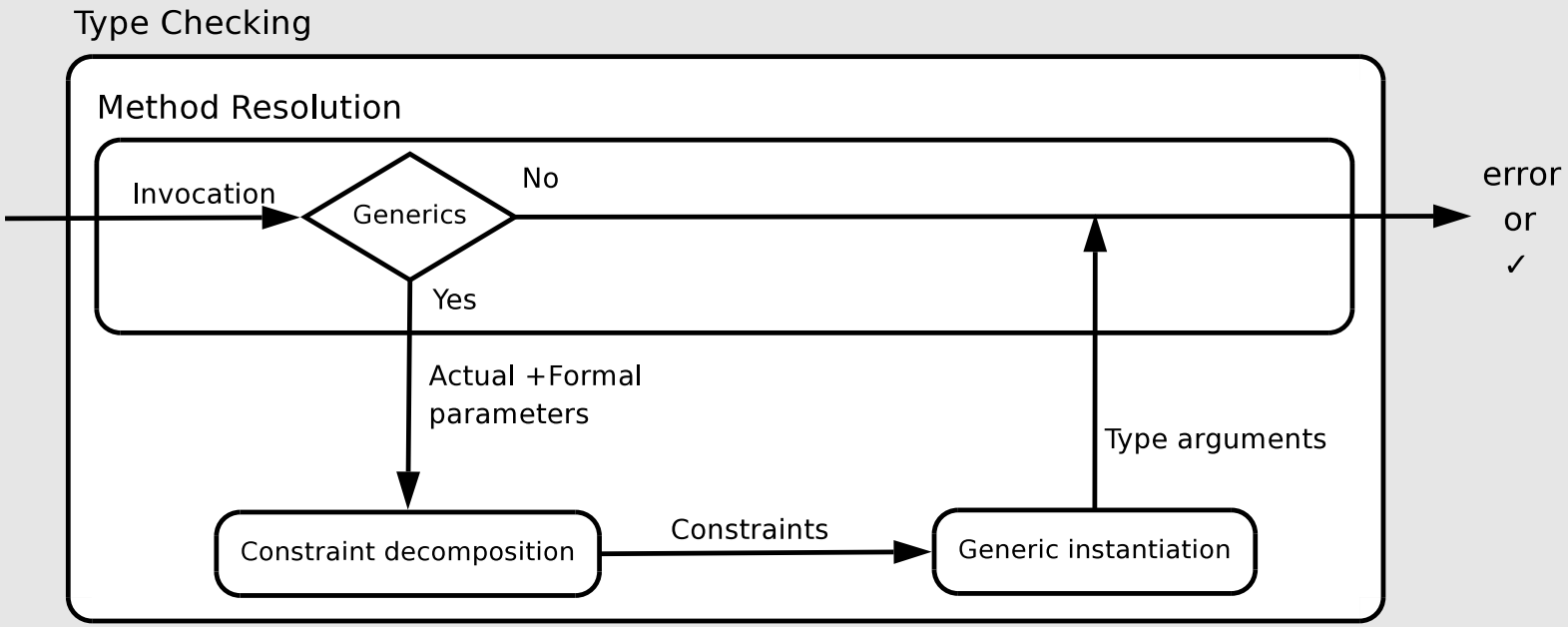
The Java type checking process



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))`

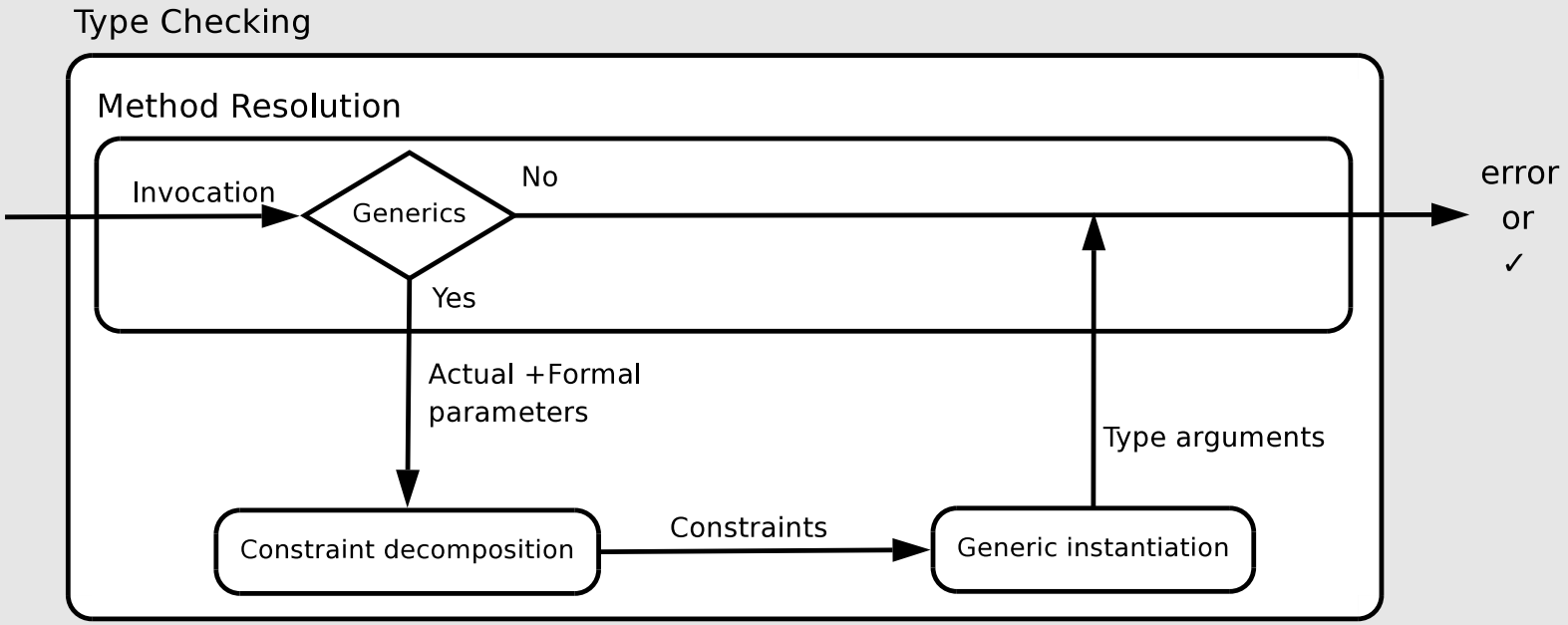
to call `void foo(HashMap<T,S extends T> map, S)`

gives

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

`Integer` $<:$ `S`

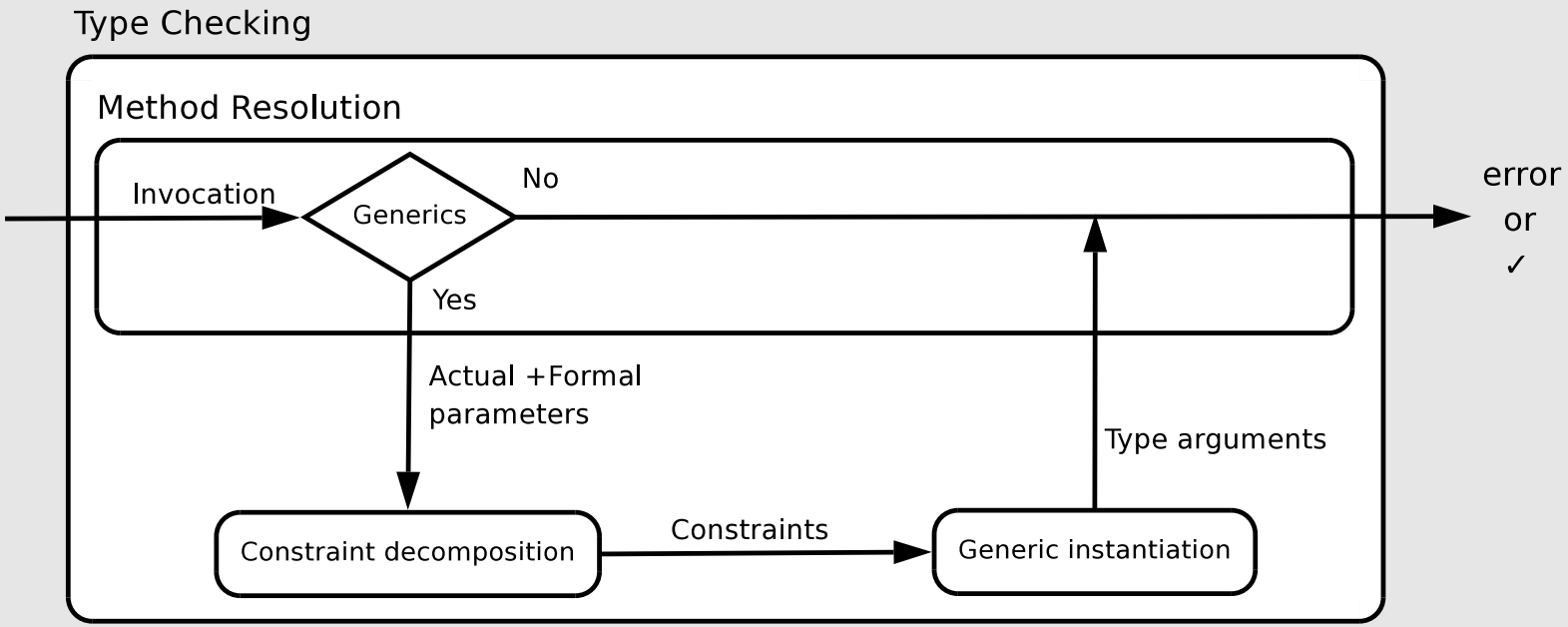
The Java type checking process



Decompose constraints: we get

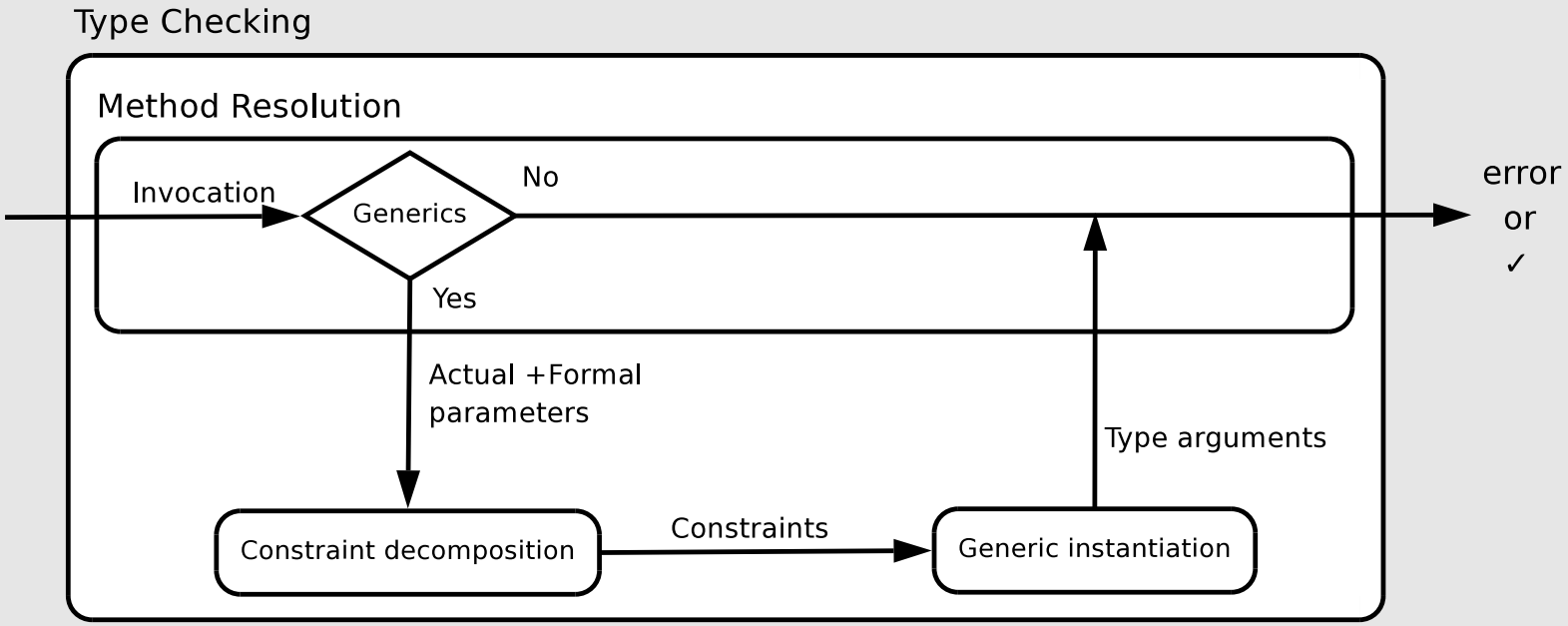
$$\{\mathtt{T} = \mathtt{Number}, \mathtt{S} = \mathtt{String}, \mathtt{S} <: \mathtt{T}, \mathtt{Integer} <: \mathtt{S}\}$$

The Java type checking process



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

The Java type checking process



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

3. The new type checking process

Principle 1: don't change, just add

- Why? It's too fragile.
- So, type checking process is performed as usual.
- If it fails, then we "redo" the process,
  - allow more candidates from resolution, and
  - don't simply decompose,
  - but also keep the original constraints around.
- Why?
  - failing at method resolution gives uninformative error message: constraints are not yet in the picture.
  - the original constraints more easily tie back to the source code.

A utility class


```
class UtilLib {
    <T> void foo(HashMap<T, ? extends T> a,
                 List<? super T> b){}
    <T> void foo(Map<T, ? extends T> a,
                 LinkedList<? super T> b){}
...
    foo(new HashMap<Integer, Number>(),
        new LinkedList<Number>());
}
```

javac:

```
UtilLib.java:11: cannot find symbol
symbol  : method foo(HashMap<Integer,Number>,
                     LinkedList<Number>)
location: class UtilLib
      foo(new HashMap<Integer, Number>(),
```

What is our messages here?

ours:

```
Method
 <T>foo(Map<T, ? extends T>, LinkedList<? super T>)
of type UtilLib is not applicable for the arguments
of type
 (HashMap<Integer, Number>, LinkedList<Number>),
because:
[*] The type Number from the expression
    ‘new HashMap<Integer, Number>()’
    on 11:11 is not a subtype of the inferred type
    for T: Integer.
```

Principle 2: relax

- Weakened Method Resolution instead of Method Resolution.
- Erase generic parts to obtain raw types.
  - `HashMap<T, ? extends T>` a becomes `HashMap a`
- JAVAC performs type inference and bounds checking to decide which `foo` is a likely suspect.
- For neither case, an explanation of the mistake is given.
- Why?
- Some checks are made to decide method resolution, but they are not used to generate the error message.

Why is method resolution like that?

Maybe because it is already quite complicated to begin with:

- Modifiers: private, static, ...
- Overloading: potentially many candidates
  - Need to weed out superfluous ones
- Autoboxing
- Methods with variable number of arguments
- Again, a very operational specification

Principle 3: hold on

- Type inference in JLS:
  - Instantiation based on decomposed constraints
  - During later checks original constraints are gone
  - Original constraints, however, link back to the source code
    - So keep them around!
  - Moreover, heuristics that weigh evidence won't work

Example

- Consider the set of constraints:
  $\{\texttt{String} <: \texttt{T}, \quad \texttt{Integer} <: \texttt{T}, \quad \texttt{T} <: \texttt{Number}\}$
- Original type inference sets `T` to the lub of `String` and `Integer`, which is `Object`
- The type checker later sees $\texttt{T} <: \texttt{Number}$, but it does not know how `T` got to be `Object`
- Indeed, no supertype of `String` can satisfy the third constraint
- So maybe the third constraint is wrong?
- Ignore it, and the lub is completely different
  - And more constraints are satisfied

Order of instantiation


```
<T, S extends T> void foo(Map<S, S> a, T a){
...
Map<Integer, String> m = ...;
foo(m, 1);
```

- Original algorithm instantiates randomly and independently
- What we do: type variables on which others depend are done first
- The more variables depend on it, the sooner we consider it
- Example: S depends on T.
- After T is set to `Integer`, we can see that the best instantiation for S is `Integer`, not `String`

4. Wrapping up

What I did not discuss,

- but is "part" of the paper:
  - Implementation in the Jastad EJC (Hedin et al.)
  - Many examples
  - Actual descriptions how to modify the various parts of the type checking process
  - We omit many of them from the paper too :-(
  - They are in Nabil's Master Thesis
- is not in the paper, but we did do:
  - heuristics for suggesting fixes to type errors
  - These are only discussed in Nabil's Master Thesis

Future work

- Become a (Generic) Java nerd
- Thus far only (generic) invocations.
  - What about other constructs, like inner classes?
- More heuristics, by capturing "expert" knowledge on error diagnosis
  - Also for the non-generic parts
- Collect programs: weird and normal
  - Have any? Send them to jur@cs.uu.nl.

Things to take home

- Implementing a type checking process is one thing.
- Having it explain why type checking fails is quite another.

Things to take home

- Implementing a type checking process is one thing.
- Having it explain why type checking fails is quite another.
- Ideally, type system designers would also consider the usability of a type system.
- Besides the usual soundness, completeness, ...

Things to take home

- Implementing a type checking process is one thing.
- Having it explain why type checking fails is quite another.
- Ideally, type system designers would also consider the usability of a type system.
- Besides the usual soundness, completeness, ...
- Three guiding principles:
  - Decouple type check from diagnostics.
    - Keep original process in tact.
  - Relax, ignore at first what is likely to be wrong.
    - E.g., the generic parts of types.
  - Keep more information, and longer.
    - Do not simply decompose constraints: structure is lost.

Just in case

- Type variables: `<T> void blah(Map<T,T> hm)`
- Wildcards: `LinkedList<?> st = ...`, but not `? x =`
- Capture conversion: local propagation of unknown types, but not for `Set<Set<?>>`
- Bounds: `T extends Number`, `? extends String`
- All in the presence of subtyping, interfaces, ...