Converting Java Programs to use Generic Libraries

Alan Donovan, Adam Kieżun
Matthew Tschantz, Michael Ernst
MIT Computer Science & AI Lab

OOPSLA 2004, Vancouver

Overview

Introduction: generic types in Java 1.5

The problem: inferring type arguments

Our approach

- Allocation type inference
- Declaration type inference

Results, Status & Related Work

```
class Cell {
   Object t;
   void set(Object t) { this.t = t; }
   Object get() { return t; }
   void replace(Cell that) {
      this.t = that.t;
   }
}
```

Library code

Client code

```
   Cell x = new Cell();
   x.set(new Float(1.0));
   x.set(new Integer(2));
   Number s = (Number) x.get();

   Cell rawCell = new Cell();
   rawCell.set(Boolean.TRUE);
   Boolean b = (Boolean) rawCell.get();
```

Generic class

Type variable

Bound

Generic method

```
class Cell<T extends Object> {
    T t;
    void set(T t) { this.t = t; }
    T get() { return t; }
    <E extends T> void replace(Cell<E> that) {
        this.t = that.t;
    }
}
```

Library code

Client code

```
Cell x = new Cell();
x.set(new Float(1.0));
x.set(new Integer(2));
Number s = (Number) x.get();

Cell rawCell = new Cell();
rawCell.set(Boolean.TRUE);
Boolean b = (Boolean) rawCell.get();
```

```java
class Cell<T extends Object> {
    T t;
    void set(T t) { this.t = t; }
    T get() { return t; }
    <E extends T> void replace(Cell<E> that) {
        this.t = that.t;
    }
}
```

Parameterized type

Library code

Client code

```java
    Cell<Number> x = new Cell<Number>();
    x.set(new Float(1.0));
    x.set(new Integer(2));
    Number s = (Number) x.get();
```

Type argument

Cast eliminated

Raw type

```java
    Cell rawCell = new Cell();
    rawCell.set(Boolean.TRUE);
    Boolean b = (Boolean) rawCell.get();
```

Cast still required

Invariant subtyping & raw types

Java 1.5 generics use **invariant subtyping:**

```
List<Float>   lf = ...;
List<Integer> li = ...;
List<Number>  ln = e ? lf : li; // wrong!
List          l  = e ? lf : li; // ok
```

Without **raw types**, `lf`, `li`, `lo` must be typed `List<Number>`

Therefore an analysis should address raw types

- but: they have subtle type-checking rules
- they complicate an approach based on type constraints
- raw `List` is not `List<T>` for any `T`

Overview

Introduction: generic types in Java 1.5

The problem: inferring type arguments

Our approach

- Allocation type inference
- Declaration type inference

Results, Status & Related Work

The Problem: Inferring Type Arguments

Generics bring many benefits to Java

- e.g. earlier detection of errors; better documentation

Can we automatically produce "generified" Java code?

There are two parts to the problem:

- **parameterisation:** adding type parameters
  - `class Set → class Set<T extends Object>`
- **instantiation:** determining type arguments at use-sites
  - `Set x; → Set<String> x;`

vonDincklage & Diwan address both problems together

We focus only on the instantiation problem. Why?

The instantiation problem

The instantiation problem is **more important**

- there are few generic libraries, but they are widely used

  e.g. collections in `java.util` are fundamental
- many applications have little generic code

Instantiation is **harder** than parameterisation

- parameterisation typically requires local changes

  (javac, htmlparser, antlr: 8-20 min each, by hand)
- instantiation requires more widespread analysis

Goals of the translation

A translation algorithm for generic Java should be:

- **sound**: it must not change program behaviour
- **general**: it does not treat specially any particular libraries
- **practical**: it must handle all features of Java, and scale to realistic programs

Many solutions are possible

- Solutions that eliminate more casts are preferred

Example: before

```
class Cell<T> {
    void set(T t) { ... }
    ...
}

Cell x = new Cell();
x.set(new Float(1.0));
x.set(new Integer(2));

Cell y = new Cell();
y.set(x);
```

Example: after

```
class Cell<T> {
    void set(T t) { ... }
    ...
}

Cell<Number> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));

Cell<Cell<Number>> y = new Cell<Cell<Number>>();
y.set(x);
```

Overview

Introduction: generic types in Java 1.5

The problem: inferring type arguments

Our approach

- Allocation type inference
- Declaration type inference

Results, Status & Related Work

Our approach

Allocation type inference

- At each generic allocation site, *"what's in the container?"*
- For soundness, must analyze all uses of the object
- `new Cell() → new Cell<Number>()`

Declaration type inference

- Propagates allocation site types throughout all declarations in the program to achieve a consistent typing
- Analyzes client code only; libraries remain unchanged
- Eliminates redundant casts
- `Cell x; → Cell<Number> x;`

Allocation Type Inference

Three parts:

1) Pointer analysis

what does each expression point to?

2) S-unification

points-to sets + declared types =

lower bounds on type arguments at allocations

3) Resolution

lower bounds → Java 1.5 types

Step 1: Pointer analysis

Approximates every expression by the set of allocation sites it points to ("points-to set")

`Cell x = new Cell₁();`	points-to(x) = { `Cell`$_1$ }
`x.set(new Float(1.0));`	points-to(t_1) = { `Float` }
`x.set(new Integer(2));`	points-to(t_2) = { `Integer` }
`Cell y = new Cell₂();`	points-to(y) = { `Cell`$_2$ }
`y.set(x);`	points-to(t_3) = { `Cell`$_1$ }

t_i are the actual parameters to each call to `set()`

`Cell`$_1$, `Cell`$_2$, `Integer` and `Float` are special types denoting the type of each allocation site

Pointer analysis details

Flow-insensitive, context-sensitive algorithm

- based on Agesen's Cartesian Product Algorithm (CPA)
- context-sensitive (for generic methods)
- fine-grained object naming (for generic classes)
- field-sensitive (for fields of generic classes)

Examines bytecodes for libraries if source unavailable (sound)

Step 2: S-unification

To determine constraints on type arguments, combine results of pointer analysis with declared types of methods/fields

Example: in call `x.set(new Float(1.0))`:

- `x` points to { $\underline{\texttt{Cell}_1}$ }
- actual parameter t_1 points to { $\underline{\texttt{Float}}$ }
- formal parameter is of declared type `T`
- so $\texttt{T}_{\texttt{Cell}_1} \geq \texttt{Float}$

For more complex types, structural recursion is required

e.g. in a call to `replace(Cell<E> v)`

S-unification example

"**unification** generating **s**ubtype constraints"

```
Cell x = new Cell1();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell y = new Cell2();
y.set(x);
```

$T_{\texttt{Cell}_1} \geq \texttt{Float}$

$T_{\texttt{Cell}_1} \geq \texttt{Integer}$

$T_{\texttt{Cell}_2} \geq \texttt{Cell}_1$

Step 3: Resolution

We must convert our richer type system to that of Java 1.5

For each type argument, s-unification discovers a set of lower bound types:

- $T_{\underline{\texttt{Cell}_1}} \geq \{ \underline{\texttt{Float}}, \underline{\texttt{Integer}} \}$
- $T_{\underline{\texttt{Cell}_2}} \geq \{ \underline{\texttt{Cell}_1} \}$

Resolution determines the most specific Java 1.5 type that can be given to each type argument

- process dependencies in topological order
  - cycles broken by introducing raw types (very rare)
- union types replaced by least-upper-bound
  - e.g. { <u>`Float`</u>, <u>`Integer`</u> } ➜ `Number`

Inferred allocation types:

```
Cell x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell y = new Cell<Cell<Number>>();
y.set(x);
```

Now we have a parameterised type for every allocation site

Next: determine a consistent Java 1.5 typing of the whole program...

Overview

Introduction: generic types in Java 1.5

The problem: inferring type arguments

Our approach

- Allocation type inference
- **Declaration type inference**

Results, Status & Related Work

Declaration Type Inference

Goal: propagate parameterized types of allocation-sites to obtain a consistent Java 1.5 program

- Input: types for each allocation site in the program
- Output: consistent new types for:
  - declarations: fields, locals, params
  - operators: casts, instanceof

Approach: find a solution to the system of type constraints arising from statements of the program

- Type constraints embody the type rules of the language
- Any solution yields a valid program; we want the most specific solution (least types)

Generation of type constraints

General form of type constraints:

- `x := y` → [[`y`]] ≤ [[`x`]] [[`x`]] means "type of x"

There are three sources of type constraints:

- **Flow of values**: assignments, method call and return, etc
- **Semantics preservation**: preserve method overriding relations, etc
- **Boundary constraints**: preserve types for library code

Conditional constraints handle raw types:

given: `Cell<`τ_1`> c; c.set("foo")`

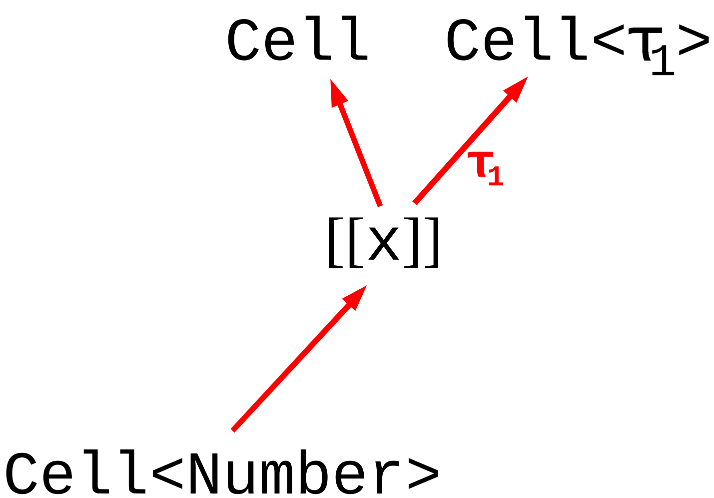
`String` ≤ τ_1 is conditional upon `c` ≠ raw

Example type-constraint graph

Declarations are elaborated with unknowns τ_i standing for type arguments

```
Cell<τ1> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<τ2> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints

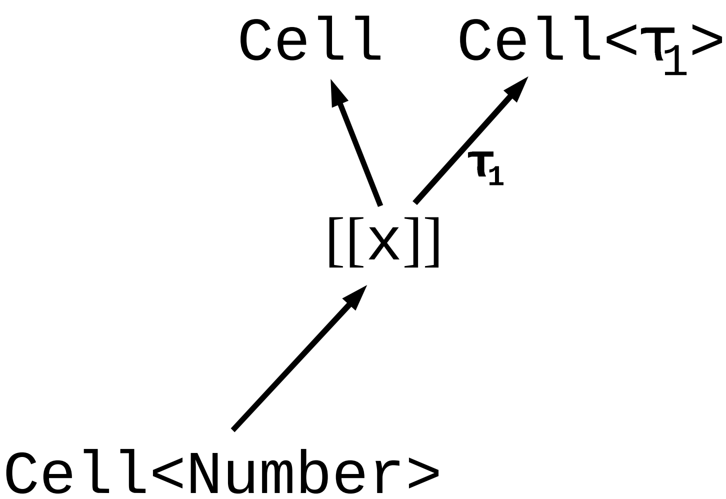


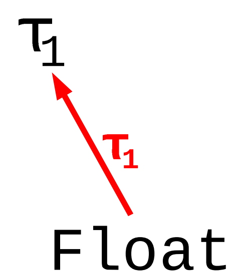
Example type-constraint graph

Declarations are elaborated with unknowns τ_i standing for type arguments

```
Cell<τ1> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<τ2> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints



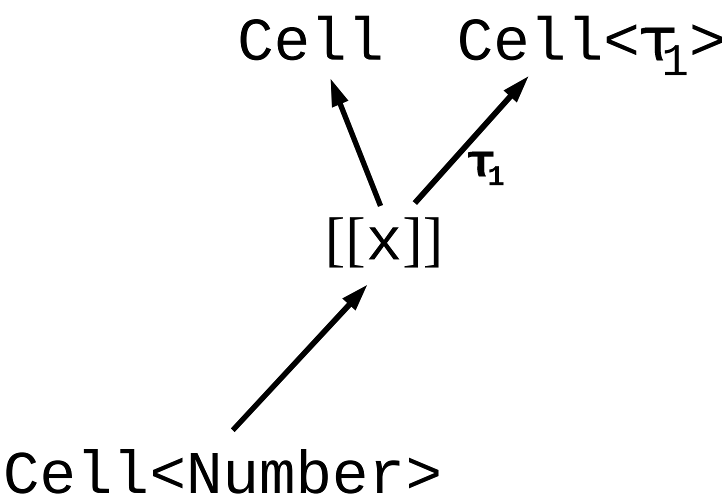


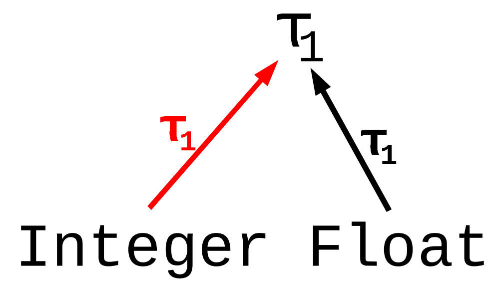
Example type-constraint graph

Declarations are elaborated with unknowns τ_i standing for type arguments

```
Cell<τ1> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<τ2> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints



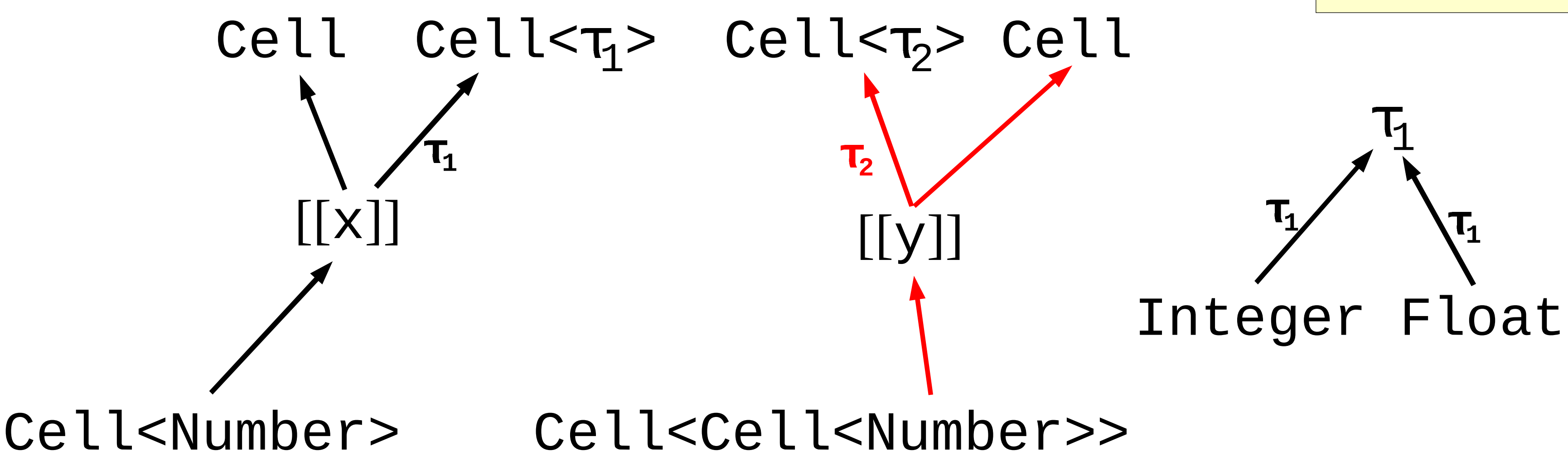


Example type-constraint graph

Declarations are elaborated with unknowns τ_i standing for type arguments

```
Cell<τ1> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<τ2> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints

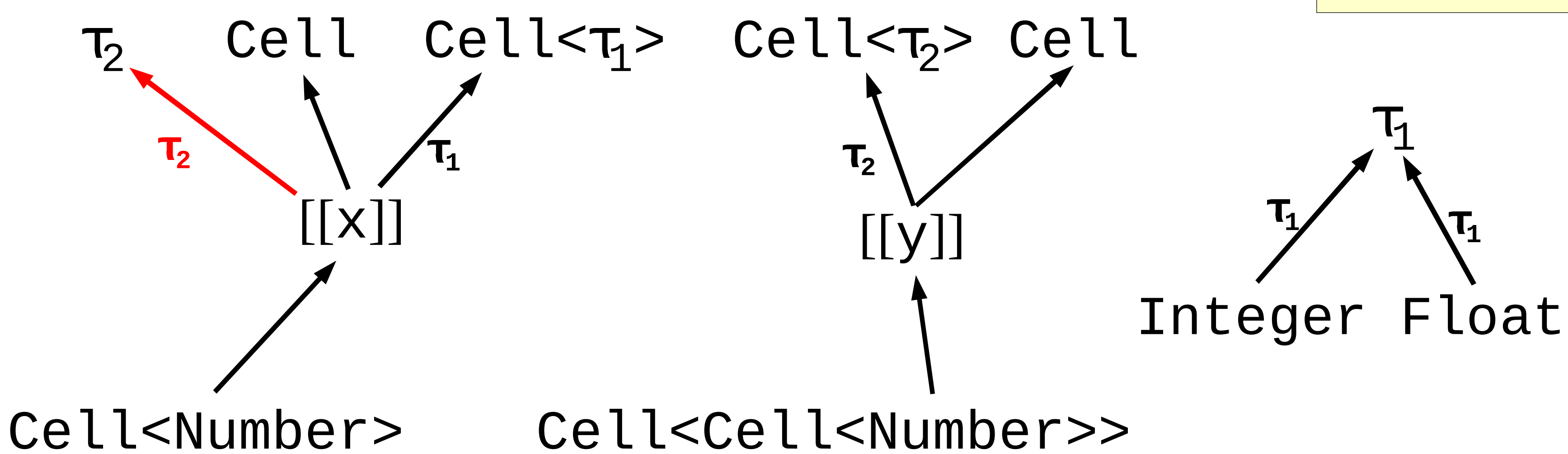


Example type-constraint graph

Declarations are elaborated with unknowns τ_i standing for type arguments

```
Cell<τ1> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<τ2> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints



Solving the type constraints

Initially, conditional edges are excluded

For each unknown τ, try to *reify* it

- i.e. include τ's conditional edges and choose a type for τ

  (chosen type is lub of types that reach it)
- then try to reify the remaining unknowns
- if this leads to a contradiction, backtrack and discard τ

  (declaration in which τ appears becomes raw)

Result:

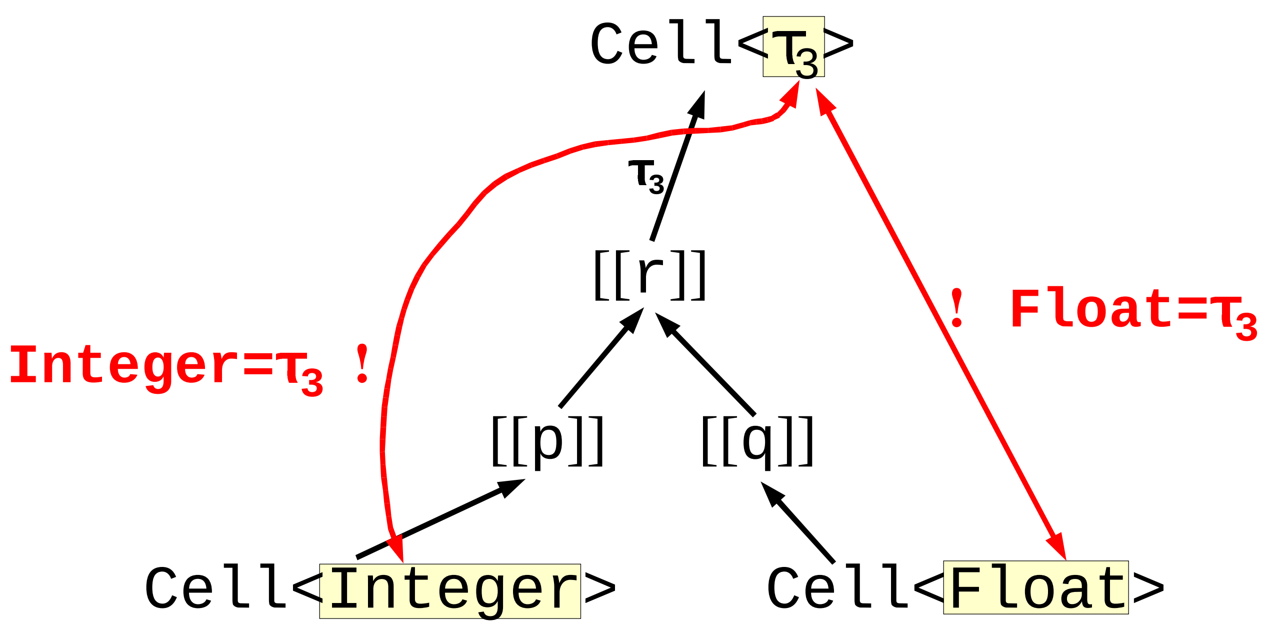
τ_1 = `Number` and τ_2 = `Cell<Number>`

so: [[x]] = `Cell<Number>`, [[y]] = `Cell<Cell<Number>>`

Contradictions cause backtracking

Consider: `Cell<τ3> r = expr ? p : q;`

When we try to reify τ_3, we get a contradiction, so τ_3 is killed and [[r]] becomes raw Cell.



Overview

Introduction: generic types in Java 1.5

The problem: inferring type arguments

Our approach

- Allocation type inference
- Declaration type inference

Results, Status & Related Work

Implementation

The analyses are implemented as a practical tool, *Jiggetai*

- it performs type analysis followed by source translation
- it addresses all features of the Java language
  (but: only limited support for class-loading, reflection)

Our tool operates in “batch” mode

- Future: could be used as an interactive application

Experimental results

Program	Lines	Casts	G.Casts	Elim	%
antlr	26349	161	50	49	98
htmlparser	13062	488	33	26	78
javacup	4433	595	472	466	99
jlex	4737	71	57	56	98
junit	5727	54	26	16	62
telnetd	3976	46	38	37	97
vpoker	4703	40	31	24	77

Lines = number of non-comment, non-blank lines of code

G.Casts = number of *generic* casts in original program

Elim = number of casts eliminated by the tool

All benchmarks ran within 8 mins/200MB on a 800Mhz PIII

Qualitative results

Four causes were responsible for most missed casts

- e.g. the “filter” idiom:

```
List strings = new ArrayList(); // <Object>!
void filterStrings(Object o) {
    if (o instanceof String)
        strings.add(o);
}
```

- Tool could be extended to handle these cases ➔ ~100%

Mostly, usage-patterns of generics are very simple

- infrequent “nesting” (e.g. `Set<List<String>>`)
- programmers avoid complex constructs if they are unaided by the type-checker

Related Work

Duggan [OOPSLA 1999]

- a small Java-like language
- simultaneous parameterisation & instantiation

von Dincklage & Diwan [OOPSLA 2004]

- Java 1.5 (without raw types)
- no guarantee of soundness
- simultaneous parameterisation & instantiation

Tip, Fuhrer, Dolby & Kieżun [IBM TR/23238, 2004]

- Java 1.5 (without raw types)
- specialised for JDK classes, but can be extended
- instantiation; parameterisation only of methods

Demo: 3.30pm
Courtyard
Demo Rm 1

Conclusion

Automatic inference of type arguments to generic classes is both feasible and practical

Our approach...

- ensures soundness in the presence of raw types
- is applicable to any libraries, not just the JDK
- readily scales to medium-size inputs (26 KLoC NCNB)
- gives good results on real-world programs

But: Java 1.5 type system is complex!

- raw types and unchecked operations make analysis hard
- solved lots of corner cases to build a practical tool