Converting Java Programs to use Generic Libraries

Alan Donovan, Adam Kiežun
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;
    }
}

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();
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;
    }
}

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();
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;
    }
}

Cell<Number>x = new Cell<Number>();
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();
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 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 $\rightarrow$ 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) = { Cell1 }
x.set(new Float(1.0));  points-to(t1) = { Float }
x.set(new Integer(2));  points-to(t2) = { Integer }
Cell y = new Cell2();  points-to(y) = { Cell2 }
y.set(x);             points-to(t3) = { Cell1 }
```

$t_i$ 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 `{Cell1}`
- actual parameter `t1` points to `{Float}`
- formal parameter is of declared type `T`
- so `T_{Cell1} \geq Float`

For more complex types, structural recursion is required.

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

“unification generating subtype constraints”

Cell \( x = \text{new Cell}_1() \);
\( x.\text{set}(\text{new Float}(1.0)) \);
\( x.\text{set}(\text{new Integer}(2)) \);
Cell \( y = \text{new Cell}_2() \);
\( y.\text{set}(x) \);
\( \text{T}_\text{Cell}_1 \geq \text{Float} \)
\( \text{T}_\text{Cell}_1 \geq \text{Integer} \)
\( \text{T}_\text{Cell}_2 \geq \text{Cell}_1 \)
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:

- \( T_{Cell_1} \geq \{ \text{Float, Integer} \} \)
- \( T_{Cell_2} \geq \{ \text{Cell_1} \} \)

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
  - \( \text{e.g. } \{ \text{Float, Integer} \} \Rightarrow \text{Number} \)
Inferred allocation types:

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

\[ x := y \rightarrow [[y]] \leq [[x]] \quad [[x]] \text{ means “type of } x” \]

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<\tau_1> c; c.set("foo")`

`String \leq \tau_1` is conditional upon \( c \neq \text{raw} \)
Example type-constraint graph

Declarations are elaborated with unknowns $\tau_i$ standing for type arguments

```java
Cell<$\tau_1$> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<$\tau_2$> y = new Cell<Cell<Cell<Number>>>();
y.set(x);
```

Labelled edges denote conditional constraints
Example type-constraint graph

Declarations are elaborated with unknowns $\tau_i$ standing for type arguments

```
Cell<$\tau_1$> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<$\tau_2$> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints
Example type-constraint graph

Declarations are elaborated with unknowns $\tau_i$ standing for type arguments

```java
Cell<$\tau_1$> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<$\tau_2$> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints
Example type-constraint graph

Declarations are elaborated with unknowns $\tau_i$ standing for type arguments

```java
Cell<$\tau_1$> x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<$\tau_2$> y = new Cell<Cell<Number>>();
y.set(x);
```

Labelled edges denote conditional constraints
Example type-constraint graph

Declarations are elaborated with unknowns $\tau_i$ standing for type arguments

```java
Cell<$\tau_1>$ x = new Cell<Number>();
x.set(new Float(1.0));
x.set(new Integer(2));
Cell<$\tau_2>$ 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 $\tau$, try to \textit{reify} it

$\uparrow$ i.e. include $\tau$'s conditional edges and choose a type for $\tau$
(chosen type is lub of types that reach it)
$\uparrow$ then try to reify the remaining unknowns
$\uparrow$ if this leads to a contradiction, backtrack and discard $\tau$
(declaration in which $\tau$ appears becomes raw)

Result:

$\tau_1 = \text{Number}$ and $\tau_2 = \text{Cell<Number>}$

so: $[[x]] = \text{Cell<Number>}, [[y]] = \text{Cell<Cell<Number>>>$
Contradictions cause backtracking

Consider: \( \text{Cell}<\tau_3> \ r = \text{expr} ? \ p : \ q; \)

When we try to reify \( \tau_3 \), we get a contradiction, so \( \tau_3 \) 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

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

**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:

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

- Tool could be extended to handle these cases $\Rightarrow \sim 100$

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
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