# Inference of generic types in Java

Alan Donovan  Michael D. Ernst

Technical Report MIT/LCS/TR-889
22 March 2003

MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139 USA
{adonovan,mernst}@lcs.mit.edu

## ABSTRACT

Future versions of Java will include support for *parametric polymorphism*, or *generic* classes. This will bring many benefits to Java programmers, not least because current Java practise makes heavy use of *pseudo*-generic classes. Such classes (for example, those in package `java.util`) have logically generic specifications and documentation, but the type system cannot prove their patterns of use to be safe.

This work aims to solve the problem of automatic translation of Java source code into Generic Java (GJ) source code. We present two algorithms that together can be used to translate automatically a Java source program into a semantically-equivalent GJ program with generic types.

The first algorithm infers a candidate generalisation for any class, based on the methods of that class in isolation. The second algorithm analyses the whole program; it determines a precise parametric type for every value in the program. Optionally, it also refines the generalisations produced by the first analysis as required by the patterns of use of those classes in client code.

## 1. INTRODUCTION

The next release of the Java programming language [12] is anticipated to include support for generic types. Generic types (or *parametric polymorphism* [6]), which make it possible to write a class or function abstracted over the types of its arguments, are one of the most wished-for programming language features in the Java community — in fact, their inclusion has been the #1 request-for-enhancement (RFE) for many years [13].

In the absence of generic types, Java programmers have been writing and using pseudo-generic classes, such as those in package `java.util`, which are expressed in terms of `Object`. Clients of such classes widen all the actual parameters to methods, and must down-cast all the return values to the type at which the pseudo-generic class is 'instantiated' in a fragment of client code. This leads to three problems:

1. **The Possibility of Error:** Java programmers often think in terms of generic types when using pseudo-generic classes. However, the Java type system is unable to prove that such objects are consistently used. This disparity allows the programmer to write, inadvertently, type-correct Java source code that manipulates objects of pseudo-generic classes in a manner inconsistent with the desired truly-parametric type. A programmer's first indication of such an error is typically a run time exception due to a failing cast; compile time checking is preferable.

2. **An Incomplete Specification:** The types in a Java program serve as a rather weak specification of the the behaviour of the program and the intention of the programmer. Generic types provide better documentation, and the type checker guarantees their accuracy.

3. **Lexical Complexity:** The user must explicitly downcast the objects retrieved from pseudo-generic classes, leading to syntactic clutter. (The non-generic type declarations are short, however.)

Non-generic solutions to the problems (e.g., defining wrapper classes such as `StringVector`) are inconvenient and error prone.

Currently, programmers who wish to take advantage of the benefits of Generic Java must translate their source code by hand; this process is time-consuming, tedious, and error-prone. We propose to automate the translation of existing Java source files into GJ, and of Java class-files to GJ class-files[1]. There are two parts to this task: adding type parameters to class definitions, and modifying uses of the classes to supply the type parameters.

There are multiple solutions to this problem. Two trivial solutions are as follows. (1) Do not use generic types at all. Since GJ is a superset of Java, a valid Java program is a valid GJ program in which each type is a GJ "raw" type, which is not parameterised. (2) Use some set of generic types, but always instantiate them at their upper bounds, and insert casts exactly where they appear in the Java program. For example, create one type parameter for each instance of a type name in the class, and instantiate the class using those actual types. Each of these two trivial solutions behaves exactly like the original program, but reaps none of the benefits of parametric polymorphism.

Our goal is to produce a set of polymorphic class abstractions that capture exactly those aspects of each class that are actually used generically, and the set of the most specific valid instantiations of those types for each use in the given client code. This is the ideal generalisation that experienced Java programmers would agree is the preferred GJ type for a particular Java class in the context of an application.

This paper presents two algorithms that together translate the source code of a Java program into source code for a semantically equivalent Generic Java (GJ) [4, 5] program. The first, *parameterisation* algorithm is an implementation-side analysis that infers both

[1]GJ class-files are class-files containing `Signature` attributes for each generic or parametric type.

which classes are inherently polymorphic and also a candidate set of type variables (and their bounds) over which each polymorphic class should be abstracted. The second, *instantiation* algorithm is a whole-program analysis that infers at what type clients instantiate the polymorphic classes. The instantiation analysis also refines the candidate type parameters of each class, based on client use of the code and on constraints inexpressible in GJ. Because Java and GJ treat primitive types identically and generic classes can only be abstracted over reference types, primitive types are largely irrelevant to this paper, so the algorithms ignore, or provide obvious default interpretations for, values of primitive types.

We constrain ourselves to the confines of the GJ language rather than selecting or inventing a new language that permits easier inference or makes different tradeoffs. (For example, some other designs are arguably more powerful or expressive, but lack GJ's integration with existing Java programs and virtual machines.) This decision sheds light on the GJ language design and makes our work of direct practical interest to Java programmers who wish to upgrade to the next version of the language. This paper uses the term *GJ* to refer to two closely related versions of Java with generic types [4, 14]. The differences between these languages are not significant to this paper.

We describe our analyses at the representation level of JVM bytecodes. This simplifies the treatment of a number of source language features, such as class-nesting, anonymous classes, generated methods, special operators (e.g., `+` for `String`), re-use of local variables, etc. Additionally, it permits the analysis to be run on classes for which source is not available (GJ allows one to *retrofit* a generic type onto a pre-existing class file). Section 4 discusses how to map the results into the GJ source domain.

Our algorithms are not guaranteed to produce a perfect result (which may depend on the intended use of the class in any event). However, the automated translation is guaranteed to be self-consistent and semantically equivalent and in typical cases (based on our hand simulation of dozens of potentially problematic classes) matches or is close to the desired goal. Programmers could interact with a tool to refine results (see Section 2.6) or make adjustments directly to the resulting code.

The remainder of this paper is organised as follows. Section 2 presents the analysis for determining how many type parameters a class definition should have, and section 2.6 explains how these results can be refined. Sections 3 explains how to instantiate generic types at uses such as declarations; the approach is to generate (Section 3.4) and resolve (Section 3.6) instantiation constraints. Section 4 shows how the results enable a translation of the Java program into GJ. Section 5 reviews related work, and Section 6 concludes.

# 2. PARAMETERISATION ANALYSIS

This section describes the algorithm that obtains intrinsically (via implementation-side analysis of a single class in isolation) a candidate set of type parameters for each class in the program.

The algorithm generates the most general possible type parameter set: it introduces as many distinct type variables as possible such that the program still type-checks. Section 2.6 discusses ways to improve the results of this analysis, both by additional analysis and by programmer intervention.

The algorithm is a dataflow analysis, and its aim is to determine a set of constraints between the type variables (and types) that will be used at each declaration (in our terminology, *origin*) in the translated GJ program. The algorithm works by computing which origins flow to each type variable.

As a simple example, consider the class class `Box` (sometimes known as `Cell`):

```
class Box {
    public void set(Object v) { this.v = v; }
    public Object get() { return v; }
    private Object v;
}
```

The analysis identifies three type variables and their bounds:

```
class Box<A extends B, B extends C, C extends Object> {
    public void set(A v);
    public C get();
    private B v;
}
```

This result is a valid GJ program, but it fails to capture the simple generic type (with a single type parameter) intended by the programmer. Our algorithms obtain the simpler type by examining uses of the class: either uses within the class itself (discovered by the parameterisation analysis, see Section 3.6.1) or external uses (discovered by the instantiation analysis, see Section 3.

We illustrate this process with a running example, a very simple `Stack` class shown in Figure 1.

```
 1:  class Stack
 2:  {
 3:      private Object[] data = new Object[10];
 4:      private int size = 0;
 5:      Object top() {
 6:          return data[size-1];
 7:      }
 8:      Object pop() {
 9:          return data[--size];
10:      }
11:      void push(Object o) {
12:          data[size++] = o;
13:      }
14:      void exchange() {
15:          Object o1 = pop(), o2 = pop();
16:          push(o1);
17:          push(o2);
18:      }
19:  }
```

Figure 1: A simple polymorphic stack implementation in Java.

## 2.1 Origins

The parameterisation algorithm begins by identifying *origins*. The set of origins is a superset of the set of type parameters. The parameterisation algorithm determines subtype and equality constraints among origins.

To a first approximation, the set of origins is the set of places in a class's signature-body where a type-variable may legally appear in GJ: an origin is a declarator of a parameter-type or return-type in the signature of a non-static method, or a declarator of a non-static field type. For the purposes of this analysis, each class should be thought of as the result of 'flattening' everything inherited from its superclasses and interfaces into a single record containing all its fields, non-overridden methods, and methods accessible via `super`.

There are additional origins for local variable declarations, array types, and array creation sites. This implies that the set of origins depends on the implementation as well as the signature of a class; see Section 2.3.3. For each origin $A$ of array type, there is origin $A'$ corresponding to the elements of origin $A$. (If $A'$ is itself of array type, it gives rise to $A''$, and so forth.) There is also an origin for each occurrence of the `new[]` array-creation operator in the methods of the class.

| Number | Name | Declared type |
|---|---|---|
| O1 | `Stack.data` | `Object[]` |
| O2 | `Stack.data'` | `Object` |
| O3 | `Stack.anewarray1` | `Object[]` |
| O4 | `Stack.anewarray1'` | `Object` |
| O5 | `Stack.top::return` | `Object` |
| O6 | `Stack.pop::return` | `Object` |
| O7 | `Stack.push::o` | `Object` |
| O8 | `Stack.exchange::o1` | `Object` |
| O9 | `Stack.exchange::o2` | `Object` |

Figure 2: Origins for the Stack example of Figure 1. In origin names, the `::` operator denotes 'in the scope of'. Origins O1 and O2 represent the expression `new Object[10]` that appears in the body of the (implicit) Stack constructor. Origin number have no semantic meaning, but are only used for presentation in this paper.

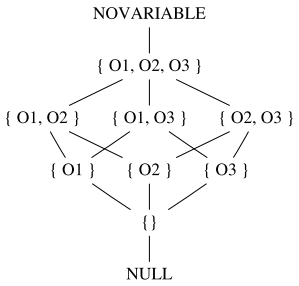


Figure 3: Lattice for the parameterisation analysis. Between NOVARIABLE and NULL is $\mathcal{P}(\{O1 \ldots On\})$, the power-set of the $n$ origins, ordered by subset inclusion $\subseteq$. The value UNKNOWN in the dataflow rules maps to NULL in the lattice.

Figure 2 shows the origins for class `Stack`.
The following properties are defined for each origin:

- javadecl(O) is the Java type associated with origin $O$; this is the *Declared type* column of Figure 2.
- element(O) is the origin associated with the element type of $O$; it is defined iff javadecl(O) is an array type. In the `Stack` example, element(O1) = O2.

The analysis makes use of the helper function origin$(name)$, which returns an origin given its name, which may be specified either by identifier (e.g., `C.f::o`) or by abstract syntax (e.g., `C.f::return`, `C.f::formal`$_1$).

## 2.2 Abstract values

### 2.2.1 Abstract value lattice

Each abstract value represents the types of values (more precisely, sets of origins that declare values) that can flow to a given Java (stack or local) variable or to an origin. One could call this the set of *reaching origins*, by analogy with reaching definitions.

The lattice $L = \langle \mathcal{P}(\mathbb{N}) \cup \{\text{NOVARIABLE}, \text{NULL}\}, \subseteq \rangle$ in Figure 3 is the domain of abstract values in the analysis.

NULL is the bottom ($\bot$) of the lattice since `null` values can flow into variables of any reference type. The UNKNOWN value indicating no information about a value (e.g., because they are reached via pointers other than `this`) is also mapped to the NULL ($\bot$) lattice value; thus, values for which we have no information do not affect the results. We distinguish UNKNOWN from NULL in the dataflow rules, even though they are the same in this lattice, because this permits a single set of dataflow rules to be used for both this analysis and an alternative one (not discussed in this paper) that eagerly fuses type variables.

NOVARIABLE is the top ($\top$) of the lattice since it represents values that cannot flow into variables whose type is given by a type-variable. In other words, such values stand for non-parameterised types.

### 2.2.2 Abstract state

Section 2.3 presents the transfer functions of the dataflow analysis as an alternative operational semantics for JVM bytecodes [16]. The abstract state of the JVM at each program point is represented as the triple *State*, defined as follows:

$$State = \langle Stack \times Locals \times Origins \rangle$$

- $Stack = Value^\star$ is a stack of abstract values, with the top-most element to the right; the invariant part is shown as '...', and the operands pushed and popped by that transfer function are named.
- $Locals = Value^{\mathsf{num\_locals}}$ is a fixed-size array of local abstract variables.
- $Origins = Origin^{\mathsf{num\_origins}}$ is a fixed-size tuple of origins, one for each origin in the current class. We use the functional notation $O[x := y]$ to represent the Origins tuple $O$ with the slot indexed by $x$ updated to contain $y$. (No join is necessary; the dataflow join operator takes care of that detail.) Analysis of each method generates an $Origins$ tuple; the result of analysing the entire class is the join of all of these tuples.

The *State* triple induces a cross-product lattice whose partial order relation is the pointwise application of the partial order relation of its three elements. In turn, the ordering relation for each of these three elements is the pointwise application of the partial order relation specified by the value lattice (see Figure 3) to the elements of each of these sets.

The well-formedness of the JVM program ensures that joins are well-defined; for example, all pairs of stacks compared are of the same height.

## 2.3 Dataflow rules

This section gives the dataflow rules — one per bytecode instruction — for the parameterisation analysis.

The dataflow analysis is applied to all instance methods of the class, including methods inherited from super-classes. Static methods are omitted since they are outside the scope of a class's type variables. Native and abstract methods have no bodies, so the analysis can do nothing with them; however, Section 2.5.1 describes a technique whereby constraints on the origins of such methods may be inferred.

In each rule, $M$ refers to the current method, and $C$ to the current class. `this` is a synonym for `M::formal`$_0$.

Figure 4 gives the results of the dataflow analysis for the `Stack` example. For brevity, the details of the abstract execution are not shown, but the annotations on the code show, for each source/sink origin pair connected by the dataflow solution, which line of source code generated it.

- ENTRY: pseudo-rule for procedure-entry block
  $\square \Longrightarrow \langle [], [\mathsf{arg}_0, \ldots, \mathsf{arg}_n], \langle \bot, \ldots, \bot \rangle \rangle$

```
class Stack
{
    private Object[] data = new Object[10]; // O3 -> O1
    private int size = 0;
    Object top() {
        return data[size-1]; // O2 -> O5
    }
    Object pop() {
        return data[--size]; // O2 -> O6
    }
    void push(Object o) {
        data[size++] = o; // O7 -> O2
    }
    void exchange() {
        Object o1 = pop(), // O6 -> O9
               o2 = pop(); // O6 -> O8
        push(o1); // O8 -> O7
        push(o2); // O9 -> O7
    }
}
```

Figure 4: Dataflow results for the `Stack` example of Figure 1.

$\forall i \in [0..n] : \mathsf{arg}_i = \{\mathsf{origin}(\texttt{M::formal}_i)\}$

Recall that `this` is $\texttt{M::formal}_0$.

- RETURN: procedure return

  $\langle[\ldots\ \mathsf{retval}], \mathsf{locals}, \mathsf{O}\rangle \overset{\mathsf{return}}{\Longrightarrow} \square$

  Retain $O[\mathsf{origin}(\texttt{M::return}) := \mathsf{retval}]$ as the result of analysing this method; the $Origins$ objects for each method are joined together to produce the result of the class analysis.

- INVOKE: method/constructor call. Calls to static methods are not parameterised; calls with a receiver of `this` have parameters identical to those of $C$; and nothing is known about calls with any other receiver.

  $\langle[\ldots\ \mathsf{receiver?}\ \mathsf{arg}_1\ ..\ \mathsf{arg}_n], \mathsf{locals}, \mathsf{O}\rangle$

  $\overset{\mathsf{invoke\ M'}}{\Longrightarrow} \langle[\ldots\ \mathsf{retval}], \mathsf{locals}, \mathsf{O'}\rangle$

  if M' is static (no receiver), $\mathsf{retval} = \mathsf{NoVariable}$ and $O' = O$.

  if receiver $\neq \{\mathsf{origin}(\texttt{this})\}$, $\mathsf{retval} = \mathsf{Unknown}$ and $O' = O$.

  if receiver = $\{\mathsf{origin}(\texttt{this})\}$: {

  $\mathsf{retval} = \{\mathsf{origin}(\texttt{M'::return})\}$

  $O' = O[\forall i \in [1..n] : origin(\texttt{M'::formal}_i) := arg_i]$

  }

- NEW: object creation expressions

  $\langle[\ldots], \mathsf{locals}, O\rangle \overset{\mathsf{new\ C'}}{\Longrightarrow} \langle[\ldots \mathsf{NoVariable}], \mathsf{locals}, O\rangle$

- NEWARRAY: new arrays of primitive type

  $\langle[\ldots\ \mathsf{count}], \mathsf{locals}, O\rangle \overset{\mathsf{newarray}}{\Longrightarrow} \langle[\ldots \mathsf{NoVariable}], \mathsf{locals}, O\rangle$

- STRING: string literals

  $\langle[\ldots], \mathsf{locals}, O\rangle \overset{\mathsf{ldc\ "foo"}}{\Longrightarrow} \langle[\ldots \mathsf{NoVariable}], \mathsf{locals}, O\rangle$

- CHECKCAST

  $\langle[\ldots\ \mathsf{expr}], \mathsf{locals}, O\rangle \overset{\mathsf{checkcast\ C'}}{\Longrightarrow} \langle[\ldots \mathsf{Unknown}], \mathsf{locals}, O\rangle$

- NULL: the `null` literal

  $\langle[\ldots], \mathsf{locals}, O\rangle \overset{\mathsf{aconst\_null}}{\Longrightarrow} \langle[\ldots, \mathsf{Null}], \mathsf{locals}, O\rangle$

- PUTFIELD: writes to instance fields

  $\langle[\ldots\ \mathsf{receiver\ value}], \mathsf{locals}, O\rangle \overset{\mathsf{putfield\ F}}{\Longrightarrow} \langle[\ldots], \mathsf{locals}, O'\rangle$

  if $\mathsf{receiver} = \{\mathsf{origin}(\texttt{this})\}$, $\mathsf{O'} = \mathsf{O}[\{\mathsf{origin}((\mathsf{F}))\} := \mathsf{value}]$

  otherwise, $O' = O$.

- GETFIELD: reads from instance fields

  $\langle[\ldots\ \mathsf{receiver}], \mathsf{locals}, O\rangle \overset{\mathsf{getfield\ F}}{\Longrightarrow} \langle[\ldots\ \mathsf{value}], \mathsf{locals}, O\rangle$

  if $\mathsf{receiver} = \{\mathsf{origin}(\texttt{this})\}$, $\mathsf{value} = \{\mathsf{origin}(\mathsf{F})\}$

  if $\mathsf{receiver} \neq \{\mathsf{origin}(\texttt{this})\}$, $\mathsf{value} = \mathsf{Unknown}$

- PUTSTATIC: writes to static fields

  $\langle[\ldots\ \mathsf{value}], \mathsf{locals}, O\rangle \overset{\mathsf{putstatic\ S}}{\Longrightarrow} \langle[\ldots], \mathsf{locals}, O\rangle$

- GETSTATIC: reads from static fields

  $\langle[\ldots], \mathsf{locals}, O\rangle \overset{\mathsf{getstatic\ S}}{\Longrightarrow} \langle[\ldots \mathsf{NoVariable}], \mathsf{locals}, O\rangle$

- ANEWARRAY: new array of references

  $\langle[\ldots\ \mathsf{count}], \mathsf{locals}, O\rangle \overset{\mathsf{anewarray_n\ C}}{\Longrightarrow} \langle[\ldots\ \mathsf{n}], \mathsf{locals}, O\rangle$

  where *n* is the origin number of the `anewarray` operator.

- AALOAD: load from array

  $\langle[\ldots\ \mathsf{arrayref\ index}], \mathsf{locals}, O\rangle \overset{\mathsf{aaload}}{\Longrightarrow} \langle[\ldots\ \mathsf{value}], \mathsf{locals}, O\rangle$

  if $\mathsf{arrayref} \notin \{\mathsf{NoVariable}, \mathsf{Unknown}\}$,
  $\mathsf{value} = \{\mathsf{element}(a) |\ a \in \mathsf{arrayref}\}$

  otherwise $\mathsf{value} = \mathsf{Unknown}$

- AASTORE: store into array

  $\langle[\ldots\ \mathsf{arrayref\ index\ value}], \mathsf{locals}, O\rangle \overset{\mathsf{aastore}}{\Longrightarrow} \langle[\ldots], \mathsf{locals}, O'\rangle$

  if $\mathsf{arrayref} \notin \{\mathsf{NoVariable}, \mathsf{Unknown}\}$,
  $O' = O\ [\forall a \in \mathsf{arrayref},\ \mathsf{element}(a) := \mathsf{value}]$

  otherwise, $O' = O$

- STACKMANIPULATION: all stack and local-variable manipulation operations (e.g., `dup`, `swap`, `push`, `pop`, `load`, `store`) are defined as in the standard semantics.

- ARITHMETIC:: all arithmetic operations simply pop and push the stack as required. All values pushed are primitive values, which are irrelevant to this analysis, so the $\perp$ lattice-value is used.

- CONTROLFLOW: all control flow operators simply pop the stack as required. Of course, they also define the control-flow graph as used by the dataflow infrastructure.

Note the symmetry of the rules for procedure entry/return and method invocation (ENTRY/RETURN and INVOKE), for reading and writing to fields (PUTFIELD and GETFIELD), and for indexing and storing to arrays (AALOAD and AASTORE).

### 2.3.1 *Following pointers*

The dataflow rules distinguish between the case where the pointer is `this`, and all other cases. Values obtained from non-`this` pointers (even those of the same class as `this`) are UNKNOWN, because the intra-class dataflow analysis cannot determine the proper type parameter instantiations for their type variables.

As an example, consider the following code:

```
1: class C {
2:   Set s;                          // O1
3:   Set foo1(C c) {                 // O2, O3
4:     return this.s;
5:   }
6:   Set foo2(C c) {                 // O4, O5
7:     return c.s;
8:   }
9: }
```

We have given the full GJ type, but our analysis is provided with unparameterised Java code and aims to determine the type parameters.

The `return this.s` statement on line 4 induces a widening from origin O1 (the declared type of `s`) to origin O2 (the return type of `foo1`): in other words, from `Set` to `Set`. Because of the use of the `this` pointer, we know that the two Sets have identical type parameter instantiations (though we do not yet know what that might be). The `return c.s` statement on line 7 induces a widening from origin O1 (the declared type of `s`) to origin O4 (the return type of `foo2`). Since parameter `c` might be instantiated with different type parameters than those (to be) declared on line 1, this widening only makes sense if a substitution is performed. for instance, suppose that the final GJ code is

```
1: class C<T> {
2:   Set<T> s;                          // O1
3:   Set<T> foo1(C<T> c) {              // O2, O3
4:     return this.s;
5:   }
6:   Set<Number> foo2(C<Number> c) { // O4, O5
7:     return c.s;
8:   }
9: }
```

The widening $O1 \leftarrow O4$ is sensible only under the substitution of `Number` for `T` in $O1$ (`Set<T>`). However, this substitution is not knowable by the parameterisation dataflow analysis: it knows neither the set of type-variables over which `C` and `Set` are parameterisation, nor what type-expressions $P_i$ are used to instantiate any given pointer. By contrast, the widening $O1 \leftarrow O2$ is sensible under the identity transformation; this transformation is known to be valid even though the instantiation of origins $O1$ and $O2$ are not yet known.

To simplify the parameterisation dataflow analysis (and to keep it intraclass), and because the information is easily obtained by the instantiation analysis, the parameterisation analysis ignores uses of pointers other than `this`.

### 2.3.2 Fixed-class expressions

Expressions that return a fixed class, e.g., `new C()`, `new P[n]` (where $P$ is primitive), and `"foo"`, cannot be assigned to a variable declared with a type-variable. GJ only permits upper-bounds to be specified for type-variables, yet assignments from these expressions all induce lower bound constraints.

Therefore the transfer function for the operations NEW and STRING return the lattice $\top$ value, NOVARIABLE. Any origin into which such expressions flow will not be declared with a type-variable. A similar GJ restriction applies to values obtained from static fields or data.

### 2.3.3 Array creation

The treatment of `anewarray` differs from that of `new C()`, which cannot be assigned to variables declared with a type-variable.

Consider the `Stack` example (Figure 1). If the types of values that flow into the elements of array `data` have an upper-bound of $T$ (where $T$ is a type-variable), then we would like to declare the field as `T[] data`. But then the assignment `T[] data = new Object[10]` would not be valid, since it does not represent a widening.

In GJ, if $T$ is a type-variable, we cannot create a new class instance with `new T()`. However, we can create a new array instance with `new T[10]` that allows reading and writing of elements of class `T` — even though the created object is in fact of class `Object[]`. So, by giving origins to `new[]` nodes, we allow them to be used in assignments such as that to `data`.

## 2.4 The candidate parameterisation

The solution to the dataflow problem is obtained in the usual way: forward-flow iteration to least-fixed point, using a single-entry, single-exit flowgraph. For each method, the dataflow equations give an $Origins$ component that associates each origin in the class with a lattice value. The value indicates, for each origin, what set of source origins may possibly flow into it by a series of assignments. The dataflow solution for the entire class is the elementwise join of the solutions for each method.

Given this dataflow solution, our aim is to select a set of type parameters and their (upper) bounds. The set of type parameters is certainly no more than the number of origins. This section shows how to select a subset of the origins, how to select bounds, and which origins to relate to each selected one.

If $Origins[O]$ is NOVARIABLE, then origin $O$ cannot be declared with a type-variable; the origin is unchanged in the GJ translation.

If $Origins[O]$ is NULL, then no values from origins in the current class flow to $O$, so we have no constraints on $O$. In the GJ translation, $O$ is replaced by a new type-variable bounded by `Object`.

The remainder of this section describes how type variables and their bounds are selected for origins into which other origins flow. The analysis consists of four steps. First, a graph of type constraints is created from the source code plus the dataflow solution. Second, the graph is augmented so that array types and element types are treated consistently. Third, the graph is simplified. Fourth, type variables that would represent array types are removed (GJ forbids parameterising over them). Finally, the set of candidate type variables and their bounds can be read directly form the graph: each node is a type variable (or a Java class) and each edge from a type variable is an upper bound on that variable.

### 2.4.1 The graph of type constraints

The analysis operates over a graph $G$ of type constraints. The nodes of the graph are all the classes in the system, plus the origins of the current class. The edges represent type constraints: they are the Java *extends* and *implements* relations, plus additional constraints due to dataflow (assignments) and bounds.

$$
\begin{aligned}
G &= \langle Classes \cup Origins, E \rangle \\
E &= \mathsf{extends} \cup \mathsf{implements} \cup \mathsf{flows} \cup \mathsf{bounds} \\
\mathsf{flows} &= \{(o, Origins[o]) \mid o \in Origin'\} \\
\mathsf{bounds} &= \{(o, \mathsf{javadecl}(\mathsf{o})) \mid o \in Origin'\} \\
Origin' &= \{o \mid Origins[o] \notin \{\text{NOVARIABLE}, \text{NULL}\}\}
\end{aligned}
$$

$Origin'$ contains the origins with lattice values that are sets; origins with lattice values of NOVARIABLE or NULL were dealt with above.

### 2.4.2 Consistent treatment of arrays

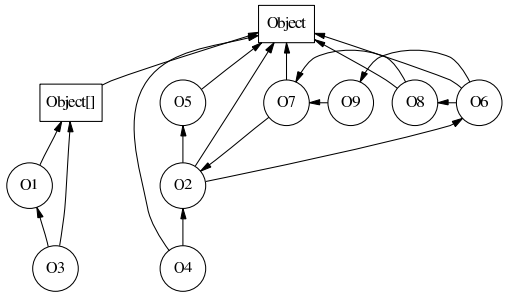


Figure 5: Constraint graph generated by analysis of the `Stack` class of Figure 1. Circles denote *origin nodes*; boxes are *class nodes* representing the classes of the program. The edges are a combination of those arising from the flow analysis and those from the Java inheritance graph.

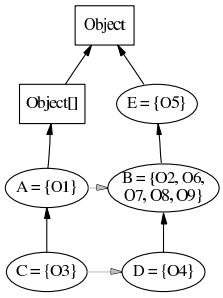


Figure 6: Constraint graph for `Stack` example after reduction step. Boxes are *class nodes* indicating whose SCC contains at least one Java class; elliptical nodes are *type-variable nodes* whose SCCs contain only origins. The grey arrows represent the element relation.

Java arrays have covariant subtyping. Therefore, for every directed edge between two array origins in the constraint graph, a corresponding edge must exist between their respective element origins (and so on, in the case of multidimensional arrays); similarly, each edge induces induces an edge between elements representing arrays of the types connected by the edge. In the `Stack` example, this process adds edge $O4 \rightarrow O2$ due to existing edge $O3 \rightarrow O1$. Figure 5 shows the constraint graph for the `Stack` example.

### 2.4.3 Graph simplification

The next step is SCC-merging, local variable elimination, and transitive reduction. This step fuses all the nodes in each strongly connected component and fuses each node containing only local variable origins with its least restrictive bound (lub). Finally, it removes the maximum number of edges possible while maintaining the partial-order relation. Figure 6 shows the reduced graph for the `Stack` example; no local variable elimination was necessary for this example.

Each SCC contains at most one Java class node. In the GJ translation of the input program, any origins that share a SCC with a Java class are cannot be represented by a type-variable, but are translated to the Java class.

### 2.4.4 Eliminate variables bounded by a `final` class

In GJ as in Java, one cannot extend a class declared `final`, so if any type-variable has such a class as one of its upper-bounds, then we eliminate that variable by fusing it with the bound class.

In practise, programmers often forget to annotate classes as `final`, so we find the principal benefit of this comes from eliminating variables $V \leq \mathsf{String}\langle\rangle$.

### 2.4.5 Eliminate `Object[]` bounded variables

GJ does not permit the bound of a type-variable to be a subclass of `Object[]` — since there would be no way to refer to the element type of such a type-variable. Therefore we eliminate each such variable as follows:

1. Colour grey all nodes labeled with a Java class derived from `Object[]`; leave all other nodes white.
2. Select any white node $N$ from which there is an edge to a grey node. If there are none, stop.
3. Let $O$ be the set of origins in the SCC associated with node $N$. Define $E$ to be the node representing the element type of node $N$:
$$E = \bigsqcup_{o \in O} \mathsf{element}(\mathsf{o})$$
4. Rename node $N$ to $E$`[]`. Color it grey. Go to step 2.

Figure 7 illustrates this process for the `Stack` example. First node $A$ is renamed `B[]`, then $C$ is renamed `D[]`.

### 2.4.6 Final solution

Now we can read the solution from the graph. The solution consists of seven type-expressions for the origin declarators and three type-variable bounds:

| Number | Type expression |
|---|---|
| O1 | `B[]` |
| O2 | `B` |
| O3 | `D[]` |
| O4 | `D` |
| O5 | `E` |
| O6 | `B` |
| O7 | `B` |
| O8 | `B` |
| O9 | `B` |

$$E \leq \texttt{Object}$$
$$B \leq E$$
$$D \leq B$$

This is all the information we would need to emit the parameterised class signature for `Stack`:

```
class Stack<E, B extends E, D extends B>
{
    private B[] data;
    private int size;
    E top();
    B pop();
    void push(B o);
    void exchange();
}
```

However, the output of this step is the set of type expressions, possibly containing bounded type-variables, for all origins in the table above, because the instantiation analysis of Section 3 requires information about local and array origins, not just about the class's signature.

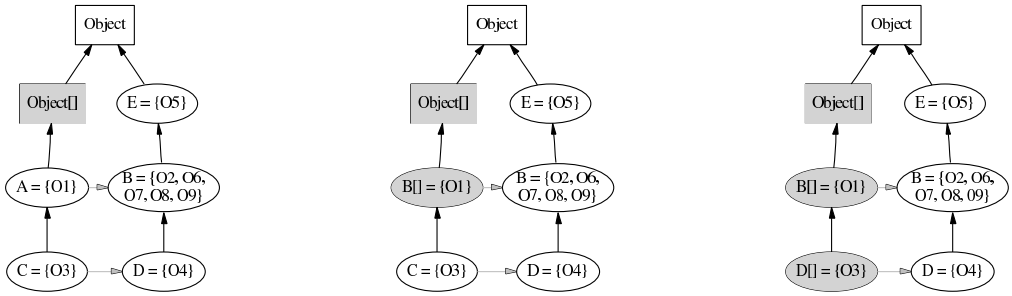


Figure 7: Elimination of array-bounded type variables in the `Stack` example. GJ does not permit the bounds of a type-variable to be an array-type. Such variables are replaced by `E[]`, where `E` is the least-upper-bound of their elements. (a) `A` is to replaced by `B[]`. (b) `C` is to be replaced by `D[]`. (c) All array-bounded variables have been eliminated.

## 2.5 Inheritance

The parameterisation analysis as described so far is applied to each class in the program in isolation. This section describes two techniques for combining per-class results to produce more precise results. The first ensures that inherited and overridden members have consistent types, and the second determines type instantiations for `extends` clauses.

### *2.5.1 Compatible inherited and overridden types*

In Java and in GJ, an overriding method must have identical formal parameter types as the overridden method. We describe a post-processing procedure to enforce this property. (An efficient implementation can combine this procedure, and also the 'flattening' pre-processing step of Section 2.1, with the main analysis, by analysing classes in depth-first pre-order, caching a stack of results obtained for superclasses.)

In this description (as in the rest of this paper), we do not distinguish between classes, abstract classes, and interfaces.

For each class $C$ in the system (in topological order), we examine in turn each class $D$ that inherits from it, either directly or transitively, and we examine the set of all origins in $D$ appearing in the signatures of methods present in both classes (i.e., inherited/overridden methods), and origins for fields of $C$ (which are inherited by $D$).

Let $E_{C,D}$ be the set of unordered pairs of origins of $C$ whose corresponding origins in $D$ belong to the same type variable as each other. (With $E$ we thus revisit the equivalence relation among origins that gave rise to the type variables.) Let $E_C = \bigcap_{D \leq C} E_{C,D}$, i.e., $E_C$ is the set of origin-pairs of $C$ that always belong to the same type variable in all subclasses of $C$.

Then, we ensure that the origins $O_a$, $O_b$ for each $(O_a, O_b) \in E_C$ belong to the same variable, fusing variables where necessary.

We will demonstrate this with an example:

```
class Abstract<A,B,C,D,E>
{
    A f(B x);
    C g(D x);
    E h;
}
class ConcreteOne<F,G> extends Abstract
{
    F f(F x) { ... }
    G g(G x) { ... }
    G h; // (inherited)
}
class ConcreteTwo extends Abstract
{
    String f(String b) { ... }
    String g(String d) { ... }
    String h; // (inherited)
}
```

Class `Abstract` has no method bodies, so no constraints are generated and each of the five origins has a different type variable. There are two concrete subclasses, each with different generalisations of the two inherited methods and the inherited field.

Numbering the origins 1–5 in order, and abbreviating the class names, we get:

$$E_{A,C1} = Pairs(\{1,2\}) \cup Pairs(\{3,4,5\})$$
$$E_{A,C2} = Pairs(\{1,\ldots,5\})$$
$$\text{and so :}$$
$$E_A = Pairs(\{1,2\}) \cup Pairs(\{3,4,5\})$$

$$\text{where } Pairs(S) = \{\langle a,b\rangle | a,b \in S, a < b\}$$

We conclude that, in all subclasses of `Abstract`, the variables $A$ and $B$ are instantiated at the same type, as are the variables $C$, $D$ and $E$. Therefore, we fuse the variables in each set, giving the following type for `Abstract`:

```
class Abstract<A,C>
{
    A f(A x);
    C g(C x);
    C h;
}
```

### *2.5.2 Superclass instantiation*

After variable fusion, we can deduce the `extends` relation for both subclasses; that is, the type parameters to the superclass. Define $T_i$ as the type expression with which the $i$th type-variable of class $C$ is instantiated by subclass $D$ in its `extends`-clause. For any origin in $C$ declared with variable $V$, let $T_i$ be the type-expression of a corresponding origin in class $D$.

Continuing our example, we obtain the following `extends`-clauses for the concrete classes:

```
class ConcreteOne<F,G>
    extends Abstract<F, G> { .. }
class ConcreteTwo
    extends Abstract<String, String> { .. }
```

This demonstrates how we can exploit patterns of use common to all subclasses present in the application to (1) infer precise parameterisations for abstract methods; (2) reduce unnecessary generality for all classes; and (3) deduce the `extends` relation which is required for the next analysis.

Section 3.7 takes a similar approach to eliminate unnecessary generality based on patterns of use common to all clients.

### *2.5.3* `Map.Entry` *example*

Here is an example from the `java.util` package. This technique generates the ideal result for interface `java.util.Map.Entry`, based on two of its subclasses, `TreeMap.Entry` and `HashMap.Entry`. In the absence of the subclasses, four distinct type-variables would have been produced for `Map.Entry`. (Results are truncated for brevity.)

```
interface java.util.Map.Entry<A, B>
{
    A getKey();
    B getValue();
    B setValue(B);
}
class java.util.TreeMap.Entry<C, D, E, F, G>
    implements java.util.Map.Entry<C, D>
{
    void Entry(C, D, G);
    C getKey();
    D getValue();
    D setValue(D);

    C key; D value; E left; F right; G parent;
}
class java.util.HashMap.Entry<H, I, J>
    implements java.util.Map.Entry<H, I>
{
    void Entry(.., H, I, J);
    H getKey();
    I getValue();
    I setValue(I);

    H key; I value; J next;
}
```

## 2.6 Refining the Parameterisation

The parameterisation analysis computes a candidate set of type variables and bounds for each class, but often the class is over-generalised. The instantiation analysis of section 3 can eliminate variables, but the results depend on exactly what constraints are be generated from the class's methods and the available client code.

We believe that the quality of the results could be further improved improved with some advice from the user. The advice would take the form of directions as to which type variables are irrelevant or unnecessary to the design of the class, and should consequently be eliminated.

We have begun implementation of a graphical tool that allows users to browse the class declarations, transformed to reflect the candidate parameterisations.

The tool displays each class signature, highlighting the uses of different variables in distinct colours. It allows the user to fuse a pair of variables together, or to eliminate a variable (replacing each occurrence of by its upper-bound) by pointing and clicking. Each edit causes the tool to update the display to reflect the new parameterisation.

The tool manages the type constraint graph, as described in section 2.4, iteratively adding constraints and re-solving in response to each user action. When the user is satisfied with the results, the parameterisation analysis is complete.

# 3. INSTANTIATION ANALYSIS

The parameterisation analysis of Section 2 determines a generic type for each class in isolation. This section presents the *instantiation* analysis that uses the results of the parameterisation analysis to deduce a complete, parametric type for every value in the program. This information can then be used to direct a source-to-source translation, as described in Section 4.

## 3.1 Overview

The instantiation analysis determines a parametric type for every type expression that makes reference to a parametric class. This includes those appearing in field and method declarations, bounds on type variables, `extends` clauses appearing in type-variable bounds, declarators of local variables, `new` expressions, and casts.

The instantiation analysis consists of five steps. First, it adds unknowns to instantiation sites (Section 3.3). Second, it generates a type constraint from each generalised assignment in the program (Section 3.4) via a one-pass whole-program static analysis. Third, it transforms some of the constraints to a more tractable form. Fourth, it solves the constraint resolution problem (Section 3.6), possibly performing more transformations as appropriate. Fifth, it optionally simplifies the results of the parameterisation analysis by eliminating unnecessary type parameters (Section 3.7).

The solution to the set of constraints over the unknowns gives concrete values (type expressions) to each of the unknowns. As noted in Section 1, this constraint system is guaranteed to have a solution. Our goal is to select the *most specific* possible instantiation type for each unknown parameter.

## 3.2 Example 1: Stack

Before presenting the five parts of the analysis in turn, we illustrate and motivate it via our running example of a `Stack` class, augmented by some client code (Figure 8). The instantiation analysis applies to the whole program at once; however, the only other classes in this program are `String` and `Object`, which take no parameters, hence we omit them. We indicate a parameterless class by `String<>` to distinguish it from the GJ raw type `String`.

First, the analysis annotates each class to reflect the result obtained from the parameterisation analysis of that class (see Section 2) and annotates all references to any class with a set of fresh unknowns, one for each variable on that class. Figure 8 shows the annotated `Stack` code.

Second, generalised assignments, declarations, and casts of the program induce type constraints.

Third, the type constraints are transformed and simplified. The constraints of the `Stack` program (annotated with their originating line numbers, and with trivial constraints omitted) are:

$$
\begin{array}{ll}
[\texttt{L21}] & \mathsf{Stack}\langle \#1, \#2, \#3\rangle \xleftarrow{=} \mathsf{Stack}\langle \#4, \#5, \#6\rangle \\
[\texttt{L21,1}] & \mathsf{Object}\langle\rangle \xleftarrow{=} [\#1, \#2, \#3/E, B, D]E \\
[\texttt{L21,1}] & [\#1, \#2, \#3/E, B, D]E \xleftarrow{=} [\#1, \#2, \#3/E, B, D]B \\
[\texttt{L21,1}] & [\#1, \#2, \#3/E, B, D]B \xleftarrow{=} [\#1, \#2, \#3/E, B, D]D \\
[\texttt{L22}] & [\#1, \#2, \#3/E, B, D]B \xleftarrow{=} \mathsf{String}\langle\rangle
\end{array}
$$

where $\xleftarrow{=}$ is a widening type constraint. Note that $[\#1, \#2, \#3/E, B, D]$ represents the substitution caused by following the `stk` pointer.

Fourth, these constraints simplify to:

$$
\begin{array}{c}
\#1 \equiv \#4 \\
\#3 \equiv \#6 \\
\mathsf{Object}\langle\rangle \xleftarrow{=} \#1 \xleftarrow{=} \#2 \xleftarrow{=} \#3 \xleftarrow{=} \mathsf{String}\langle\rangle
\end{array}
$$

For each unknown with a lower bound and only the trivial $\mathsf{Object}\langle\rangle$ upper bound, we instantiate the unknown to its lower bound, repeating the process until no further progress is made. This instantiates $\#3$, $\#2$, and $\#1$ (in that order) to $\mathsf{String}\langle\rangle$, giving us the result:

```
20:  String<> test(String<> str) {
21:      Stack<String<>, String<>, String<>> stk =
```

```
 1: class Stack<E extends Object<>, B extends E, D extends B>
 2: {
 3:     private B[] data = new D[10];
 4:     private int size = 0;
 5:     E top() {
 6:         return data[size-1];
 7:     }
 8:     B pop() {
 9:         return data[--size];
10:     }
11:     void push(B o) {
12:         data[size++] = o;
13:     }
14:     void exchange() {
15:         B o1 = pop(), o2 = pop();
16:         push(o1);
17:         push(o2);
18:     }
19: }
    ...
20: String<> test(String<> str) {
21:     Stack<#1, #2, #3> stk = new Stack<#4, #5, #6>();
22:     stk.push(str);
23:     return (String<>)stk.top();
24: }
```

Figure 8: `Stack` example, annotated to reflect the parameterisation analysis of Section 2, plus calling code, annotated with fresh unknowns for each type instantiation.

```
            new Stack<String<>, String<>, String<>>();
22:      stk.push(str);
23:      return (String<>)stk.top();
24:  }
```

Fifth, we observe that in this program, for any instantiation of class `Stack` in the whole program, the type-expressions for each parameter position are equal. Therefore, the three variables can be fused into one. Figure 9 shows the final result.

The following sections explain each of the above steps in more detail.

## 3.3 Insertion of type parameters

The first step is to annotate the program using the results of the parameterisation analysis. For each class, the result includes a set of upper-bounded type variables $\langle V_1 \leq B_1, \ldots, V_n \leq B_n \rangle$, and a mapping from origins to declaration type-expressions, possibly containing variables.

The annotation first adds type variables to class declarations and replaces origins by the type variables. Then, it annotates client code.

Every reference to a class identifier $C$ (whether in a declaration, `new`-expression, cast, etc.) is augmented by a fresh set of unknowns, one per type variable on class $C$.

Types appearing in `extends` and `implements` clauses are treated in a similar way. The parameterisation analysis is often able to determine some of the type-expressions appearing as parameters (see Section 2.5), because they are constants (such as `String`) or are a function of the variables exported from the extending class.

## 3.4 Constraint generation

We define *generalised assignment* as the propagation of (unmodified) reference values from one variable or expression to another location — anywhere that a reference expression is *assignment converted* or *method-invocation converted* according to the rules of the Java language [12]. Examples include ordinary assignment, parameter passing, returning a value from a method, assigning or reading a field or array element, etc.

```
 1:  class Stack<E extends Object<>>
 2:  {
 3:      private E[] data = new E[10];
 4:      private int size = 0;
 5:      E top() {
 6:          return data[size-1];
 7:      }
 8:      E pop() {
 9:          return data[--size];
10:      }
11:      void push(E o) {
12:          data[size++] = o;
13:      }
14:      void exchange() {
15:          E o1 = pop(), o2 = pop();
16:          push(o1);
17:          push(o2);
18:      }
19:  }
     ...
20:  String<> test(String<> str) {
21:      Stack<String> stk = new Stack<String>();
22:      stk.push(str);
23:      return stk.top();
24:  }
```

Figure 9: Final GJ code for the `Stack` example and calling code.

At each generalised assignment, there is the potential for a widening reference conversion. Therefore, in a GJ program, a generalised assignment indicates a subtype constraint between the types of its source expression and destination location.

The constraint generation step generates constraints from three sources. Each instance of a generalised assignment anywhere in the whole program generates a constraint of the form

$$\mathsf{typeof}(locn) \xleftarrow{=} \mathsf{typeof}(expr)$$

where $\mathsf{typeof}(x)$ is the type of the expression or location $x$ (figure 10). The constraint is read as "*expr* can be assigned to *locn*". Each type-variable bound also generates a generalised-assignment constraint $\xleftarrow{=}$, since an expression whose type is given by a type-variable can be assigned to a variable declared by the bound type-expression. Finally, each cast operator `(T)e` generates a cast constraint:

$$T \xleftarrow{\leq} \mathsf{typeof}(e)$$

Both constraint relations, $\xleftarrow{=}$ and $\xleftarrow{\leq}$, define partial orders over parametric types (they are reflexive and transitive). The grammar of reference types ($T$) and of constraints ($K$) is:

$$\begin{array}{rll} T ::= & \#n & (\text{unknowns}) \\ | & C\langle T_1, \ldots, T_n \rangle & (\text{parametric classes}) \\ | & V & (\text{type variables}) \\ | & T[\,] & (\text{arrays}) \\ | & P[\,] & (\text{primitive arrays}) \end{array}$$

$$\begin{array}{rll} K ::= & T_1 \xleftarrow{=} T_2 & (\text{assignment}) \\ | & T_1 \xleftarrow{\leq} T_2 & (\text{cast}) \\ | & T_1 \equiv T_2 & (\text{equality}) \end{array}$$

For example, `p.m(p.f, k)` creates two constraints, one from

| $x$ | typeof($x$) |
|---|---|
| `null` | Null |
| `"string literal"` | String$\langle\rangle$ |
| `new C<T1..Tn>()` | $C\langle T_1, \ldots, T_n\rangle$ |
| `new T[]` | T[] |
| `new P[]` | P[] |
| ArrayElement `a[x]` | element(typeof(a)) |
| Local `l` | decltype(l) |
| Field `p.f` | $[T_1, \ldots, T_n/V_1, \ldots, V_n]$ F |
| Method `p.m(...)` | $[T_1, \ldots, T_n/V_1, \ldots, V_n]$ R |

Figure 10: Informal definition of typeof$(x)$. The definitions for `p.f` and `p.m(...)` assume `p : C<T1..Tn>` and `class C<V1..Vn> { ... F f; R M(...); ... }`.

field `p.f` to `m`'s first parameter, one from local `k` to the second parameter.

Aside from constraints generated by generalised assignments and casts, we need to include bounds-constraints on unknowns (for every reference), and bound- constraints on type variables (for every class). The structures of these two kinds of constraint are parallel: one represents the bounds-constraints *inside* the class body, where type-variables are in scope, and the other represents the same constraints but *externally*, and thus we must apply the appropriate pointer substitution (see section 3.4.1) to the same constraint. Both kinds are subtype constraints, so we use the $\overset{=}{\leftarrow}$ relation. For class $C$, and reference $p$:

```
class C<B1 extends B2,
    B2 extends Number<>> {..}
...
C<#1,#2> p = somefunc();
```

we obtain these bounds-constraints on the variables:

$$\mathsf{Number}\langle\rangle \overset{=}{\leftarrow} B2 \overset{=}{\leftarrow} B1$$

and these on the unknowns:

$$[\#1, \#2/B1, B2] \ (\mathsf{Number}\langle\rangle \overset{=}{\leftarrow} B2 \overset{=}{\leftarrow} B1)$$
$$\text{i.e.} \quad \mathsf{Number}\langle\rangle \overset{=}{\leftarrow} \#2 \overset{=}{\leftarrow} \#1$$

### *3.4.1 Substitution*

In contrast to the parameterisation analysis, constraint generation follows pointers $p$, applying a transformation to the declared type of the entity referred to via the pointer

The following of a pointer with a parametric type establishes a different type environment for the body of the class referred to by that pointer. Just as $\beta$-reduction (function application) in the $\lambda$-calculus causes bound variables to be replaced by operand expressions throughout the $\lambda$-body, so in GJ do parametric instantiations cause type-variables to be replaced by type-parameter expressions throughout the class body.

Figure 10 abbreviates the following field rule and a similar one for methods:

$$\frac{\Gamma \vdash \mathtt{p} : C\langle T_1, \ldots, T_n\rangle \quad \Gamma \vdash \mathtt{C} : \mathtt{class}\ C\langle V_1, \ldots, V_n\rangle\{\ldots F\, f; \ldots\}}{\Gamma \vdash \mathtt{p.f} : [V_1/T_1, \ldots, V_n/T_n]F}$$

For example, in the following fragment of GJ code, the type of expression `p.v` is given by $[\mathsf{String}\langle\rangle, \mathsf{Integer}\langle\rangle/K, V]\ \mathsf{Vector}\langle V\rangle$. This type-expression can be simplified to $\mathsf{Vector}\langle\mathsf{Integer}\langle\rangle\rangle$, hence the assignment to `x` is valid.

```
class MyMap<K,V> {
    Vector<V> v;
}
void f(MyMap<String<>, Integer<>> p) {
    Vector<Integer<>> x = p.v;
    // p.v : [String<>, Integer<>/K,V]Vector<V>
    //     : Vector<Integer<>>
}
```

## 3.5 Constraint-set augmentation

The set of constraints generated in Section 3.4 is augmented to ease their solution. Some cast constraints give rise to generalised assignment constraints, and some generalised assignment constraints give rise to equality constraints.

### *3.5.1 Casts*

Not all casts should influence the final type parameterisation and instantiation. While some casts are guaranteed to succeed regardless of calling context, others depend on application invariants beyond the scope of this (or any) analysis.

If we cannot deduce statically that a cast is redundant, then we cannot assume that the parametric type of the cast is a subclass of the cast operand type — it could be an unrelated type. For example, the following Java program:

```
1:  class D extends C { ... }
2:
3:  void f(C c1) {
4:      Vector v = new Vector();
5:      v.set(0, new D());
6:      C c2 = (C)v.get(0);  // guaranteed to succeed
7:      D d = (D)c1;         // depends on application
8:  }
```

has this intermediate-GJ representation during the instantiation analysis:

```
1:  class D<V> extends C<String> { ... }
2:
3:  void f(C<#1> c1) {
4:      Vector<#2> v = new Vector<#3>();
5:      v.set(0, new D<#4>());
6:      C<#5> c2 = (C<#6>)v.get(0);
7:      D<#7> d = (D<#8>)c1;
8:  }
```

Let's assume that during constraint resolution, it becomes clear that the type of what is returned from `get` is the same as what is passed to `set`, i.e., typeof(v.get(0)) $= \#2 \equiv \#3 \equiv D\langle\#4\rangle$. Then the cast on line 6 (which generates the cast constraint $C\langle\#6\rangle \overset{\leq}{\leftarrow} D\langle\#4\rangle$) is a widening that need not appear in the translated GJ program. Thus, we can convert the cast constraint into the assignment constraint $C\langle\#6\rangle \overset{=}{\leftarrow} D\langle\#4\rangle$, which gives us $\#6 \equiv \mathsf{String}\langle\rangle$ (see Section 3.5.2).

The cast on line 7, however, from $C\langle\#1\rangle$ to $D\langle\#8\rangle$ cannot be eliminated by GJ, and thus we cannot conclude anything about the value of $\#8$ from the cast.

Therefore we can only draw the following limited conclusion from a cast constraint. A constraint $C\langle T_1, \ldots, T_n\rangle \overset{\leq}{\leftarrow} T_e$ may be converted to the constraint $C\langle T_1, \ldots, T_n\rangle \overset{=}{\leftarrow} T_e$ if and only if constraint resolution (Section 3.6) has identified (fused) $T_e$ with some parametric type $D\langle U_1, \ldots, U_m\rangle$ where $D \leq C$.

The reasoning behind this is as follows. If the cast operand type $D$ is a subtype of the cast type $C$, then the cast is trivial, made redundant by the type-system of GJ. This happens with casts inserted by the programmer using pseudo-generic Java classes. A trivial cast is, in effect, an assignment conversion — just like any other assignment.

### 3.5.2 Congruence

Some generalised assignment constraints give rise to equality constraints via a mechanism we call parameter *congruence*. A constraint of the form:

$$A\langle T_1, \ldots, T_n\rangle \xleftarrow{=} B\langle U_1, \ldots, U_m\rangle$$

implies (i) that $B \leq A$ in the Java inheritance graph, and (ii) that $B\langle U_1, \ldots, U_m\rangle \leq A\langle T_1, \ldots, T_n\rangle$ in GJ parametric type system.

Let us first consider the (common) case in which $B = A$, in which case $m = n$. We generate the equivalence constraints $T_1 \equiv U_1$, ..., $T_n \equiv U_n$, because since GJ does not admit parameter-variance of parametric types. i.e.:

$$C\langle T_1, \ldots, T_n\rangle \leq C\langle U_1, \ldots, U_n\rangle \iff \forall_{i=1..n} T_i = U_i$$

Note that since each $T_i \equiv U_i$ is a constraint over type expressions, we unify them, possibly giving rise to additional constraints. For example, $\mathsf{Pair}\langle \#3, \mathsf{F}\langle \#4\rangle\rangle \xleftarrow{=} \mathsf{Pair}\langle \#5, \#6\rangle$ gives us $\#3 \equiv \#5$ and $\mathsf{F}\langle \#4\rangle \equiv \#6$.

In the case where $A \neq B$, we must consider the effect of a widening from $B$ to $A$ on the parametric type. In GJ, when a class extends a generic class, it may generalise or specialise — or both — the super class. Therefore some of the type-variables of the super class are also variables of the subclass while others are instantiated to a type-expression by the subclass.

We define the function widen as follows:

$$\frac{\mathsf{class}\ B\langle V_1, \ldots, V_n\rangle\ \mathsf{extends}\ A\langle U_1, \ldots, U_m\rangle \quad S_i = [T_1, \ldots, T_n/V_1, \ldots, V_n]U_i}{\mathsf{widen}(B\langle T_1, \ldots, T_n\rangle, A) = \langle S_1, \ldots, S_m\rangle}$$

This function returns the parameter tuple $\langle S_1, \ldots, S_m\rangle$ with which $B\langle T_1, \ldots, T_n\rangle$ instantiates $A$, which the algorithm uses for generating a set of pointwise equivalence constraints.

So, for example, if $\mathsf{class}\ \mathsf{K}\langle P, Q, R\rangle\ \mathsf{extends}\ \mathsf{J}\langle R, \mathsf{String}\langle\rangle\rangle$, then $\mathsf{widen}(J\langle T_1, T_2, T_3\rangle, J) = \langle T_3, \mathsf{String}\langle\rangle\rangle$.

## 3.6 Constraint resolution

Constraint resolution can be viewed as an iterative graph-reduction process. Each type expression in each of the constraints represents a node in the graph. Each $\xleftarrow{=}$ constraint is a directed edge; $\xleftarrow{\leq}$ constraints do not appear in the graph.

The goal of constraint resolution is to find a set of assignments to the unknowns such that all the constraints are satisfied. There is at least one (trivial) solution, but in practise there are many solutions.

It is convenient to consider the set of equivalence-classes of the unknowns. Initially, each unknown is in its own equivalence class, but as resolution proceeds, these classes are fused. After each constraint resolution step, the constraint augmentation of Section 3.5 is run; this can be done incrementally.

### 3.6.1 SCC-merging

The primary form of resolution, as in the parameterisation analysis, is SCC-merging. Wherever a cycle exists in the directed graph, all nodes lying on that cycle must be equivalent, so their equivalence classes are fused.

### 3.6.2 Lower bounds

SCC-merging (and constraint augmentation) may not put every unknown in an equivalence class with a concrete type (i.e. a type-expression containing no unknowns).

So, when resolution can make no further progress, for each unknown $\#U$ whose equivalence class contains only other unknowns, and for which there is a defined greatest lower bound $B$, we push $\#U$ to that bound $B$. In other words, we use the *most specific* consistent set of instantiations. For example, if the only remaining constraint on $\#3$ was $\#3 \xleftarrow{=} C\langle \#2\rangle$, it would be replaced with $\#3 \equiv C\langle \#2\rangle$.

### 3.6.3 Upper bounds

After pushing to lower bounds, any remaining unknowns that have no equivalent concrete type expression are pushed to their least upper bound. All unknowns have the upper bound $\mathsf{Object}\langle\rangle$.

## 3.7 Instantiation patterns analysis

Section 2.5.2's technique for determining `extends`-clauses looked for simple patterns in the instantiation expressions among all subclasses of a class $C$. Now that we have computed the complete set of parametric references $C\langle T_1, \ldots, T_n\rangle$ — within $C$, all its subclasses, and its clients — we can look for more subtle patterns.

Consider a generic class $C\langle P_1, \ldots, P_n\rangle$. If there is some pair $\langle P_i, P_j\rangle$ of $C$'s parameters, such that in all subclasses of $C$, the $i$th and $j$th type expressions with which $C$ is instantiated are related by some function over types, then one of the variables may be eliminated, and replaced within class $C$ by the appropriate function of the other type variable.

For example, the following example shows class $C$ and the complete set of (two) parametric references to it. Note that $P$ and $R$ are always instantiated with identical type expressions:

```
class C<P, Q, R> { .. }
..
C<String, Vector<String>, String> myref1 = ...;
C<T, Vector<T>, T> myref2 = ...;
```

Therefore, they can be fused, giving rise to this simplified result:

```
class C<P, Q> { .. } // [Q/R] in class body
..
C<String, Vector<String>> myref1 = ...;
C<T, Vector<T>> myref2 = ...;
```

So far, this case is very similar to the method mentioned already in section 2.5.2; indeed, redundant variables such as $R$ would be found and eliminated by that analysis. But this technique can be generalised to handle cases where the pairs of parameter instantiation type-expressions are related by a relation other than equality. Continuing with our example, note that the instantiation expression for the second parameter is always a `Vector` of the first:

```
class C<P> { .. } // [Vector<P>/Q] in class body
..
C<String> myref1 = ...;
C<T> myref2 = ...;
```

Such patterns can be found by structural unification of corresponding elements in the parameter-tuples $T_1, \ldots, T_n$ of all references to a given class $C$.

We have found in working through many examples that a significant number of unwanted type variables are always instantiated predictably, either with a constant expression or with some type-expression of the other parameters, and so eliminating such cases would improve accuracy.

Applying such a refinement requires a new instantiation analysis-run to determine the new parameters for the smaller parameter sets.

## 3.8 Overconstrained variables

When a set of constraints places a lower bound on a type variable, we say the variable is *overconstrained*, and it must be eliminated.

For example, with the constraints $\mathsf{Graph}\langle\#1,\#2\rangle \xleftarrow{=} G \xleftarrow{=} \mathsf{Graph}\langle\#6,\#7\rangle$, where $G$ is a type variable, $G$ is overconstrained, because it has a lower bound of Graph.

In this case, it is evident that $G = \mathsf{Graph}\langle\#1 \equiv \#6, \#2 \equiv \#7\rangle$. In other cases, we may not be able to infer a precise type for $G$, so we simply use the lower-bound.

Firstly, we remove it from the class's variable list, replacing occurrences of that variable within the class body with the bound on that variable. Then we remove the unknown in the corresponding parameter position from every reference to this class.

## 3.9 Examples

We finish our discussion of the instantiation analysis with additional examples.

### *3.9.1 Example 2: Map*

Assume the following source program:

```
1: static Map test() {
2:     Map m = new HashMap();
3:     m.put("foo", new Integer(3));
4:     m.put("bar", new Float(3.0));
5:     return m;
6: }
```

and `HashMap<K,V> extends Map<K,V>` from the parameterisation analysis, we annotate the code as follows:

```
1: static Map<#1,#2> test() {
2:     Map<#3,#4> m = new HashMap<#5,#6>();
3:     m.put("foo", new Integer<>(3));
4:     m.put("bar", new Float<>(3.0));
5:     return m;
6: }
```

from which we generate the following constraints:

$$\mathsf{Map}\langle\#3,\#4\rangle \xleftarrow{=} \mathsf{HashMap}\langle\#5,\#6\rangle$$
$$\mathsf{Map}\langle\#1,\#2\rangle \xleftarrow{=} \mathsf{Map}\langle\#3,\#4\rangle$$
$$\#3 \xleftarrow{=} \mathsf{String}\langle\rangle$$
$$\#4 \xleftarrow{=} \mathsf{Integer}\langle\rangle$$
$$\#4 \xleftarrow{=} \mathsf{Float}\langle\rangle$$

From them, and the fact that method `Map.put` has type `put(K,V)`, we can conclude that $\#1 \equiv \#3 \equiv \#5$ and $\#2 \equiv \#4 \equiv \#6$, since the only assignment-compatible `Map` and `HashMap` types have the same parameter-sets (see Section 3.5.2) and that $\#4 \xleftarrow{=} \mathsf{Number}\langle\rangle$, since `Number` is the join of `Integer` and `Float`. By pushing all remaining unknowns to their lower bounds (if any), and any unknowns remaining after that to their upper bounds, we obtain the ideal type for this client code:

```
1: static Map<String<>,Number<>> test() {
2:     Map<String<>,Number<>> m =
           new HashMap<String<>,Number<>>();
3:     m.put("foo", new Integer<>(3));
4:     m.put("bar", new Float<>(3.0));
5:     return m;
6: }
```

### *3.9.2 Example 3: Graph*

The next example, `Graph`, is part of a directed graph class in our prototype Java-to-GJ translator. Its `scc()` method returns a `Graph` whose instantiation type is more complex than that of `this`. This example violates an assumption made by related work [9] that, within the methods of a class $C$, any reference of type $C$ must be instantiated with the same type-parameters as `this`.

(The idea behind `scc` is that it returns a `Graph` $G'$, each node of which represents an SCC of the original graph $G$ and is labeled with the `Set` of $G'$ nodes in that SCC. Thus if $G$ has type $\mathsf{Graph}\langle T\rangle$, $G'$ has type $\mathsf{Graph}\langle\mathsf{Set}\langle\mathsf{Node}\langle T\rangle\rangle\rangle$. For brevity, the `scc` method shown here does not actually compute the SCCs, but it merely has the same type as a method that would. Likewise, `Set` is simplified to contain a single object.)

```
class Node { Object label; }
class Set { Object value; } // Very small set!

class Graph
{
    Set nodes = new Set();

    void addNode(Object label) {
        Node n = new Node();
        n.label = label;
        nodes.value = n; // Add to 'set'
    }

    Graph scc() {
        Graph g = new Graph(); // new graph has one node
        g.addNode(nodes);      // labeled by set of old nodes
        return g;
    }
}
```

In this simple example, the parameterisation analysis gives each of the three classes one variable, bounded at $\mathsf{Object}\langle\rangle$.

```
 1  class Node<A extends Object<>> { A label; }
 2  class Set<B extends Object<>>  { B value; }
 3
 4  class Graph<C extends Object<>>
 5  {
 6      Set<#1> nodes = new Set<#2>();
 7
 8      void addNode(C label) {
 9          Node<#3> n = new Node<#4>();
10          n.label = label;
11          nodes.value = n;
12      }
13
14      Graph<#5> scc() {
15          Graph<#6> g = new Graph<#7>();
16          g.addNode(nodes);
17          return g;
18      }
19  }
```

The non-trivial generated constraints (with line numbers) are:

$$\mathsf{Set}\langle\#1\rangle \xleftarrow{=} \mathsf{Set}\langle\#2\rangle \quad [L6]$$
$$\mathsf{Node}\langle\#3\rangle \xleftarrow{=} \mathsf{Node}\langle\#4\rangle \quad [L9]$$
$$[\#3/A]A = \#3 \xleftarrow{=} C \quad [L10]$$
$$[\#1/B]B = \#1 \xleftarrow{=} \mathsf{Node}\langle\#3\rangle \quad [L11]$$
$$\mathsf{Graph}\langle\#6\rangle \xleftarrow{=} \mathsf{Graph}\langle\#7\rangle \quad [L15]$$
$$[\#6/C]C = \#6 \xleftarrow{=} \mathsf{Set}\langle\#1\rangle \quad [L16]$$
$$\mathsf{Graph}\langle\#5\rangle \xleftarrow{=} \mathsf{Graph}\langle\#6\rangle \quad [L17]$$

L6, L9, L15, and L17 induce equivalence constraints, and the remainder, after substitution, give a lower bound to each equivalence-class of unknowns:

$$\#1 \equiv \#2 \xleftarrow{=} \mathsf{Node}\langle\#3\rangle$$
$$\#3 \equiv \#4 \xleftarrow{=} C$$
$$\#5 \equiv \#6 \equiv \#7 \xleftarrow{=} \mathsf{Set}\langle\#1\rangle$$

Now we push all unknowns with lower bounds (all of them, in this case) to those bounds:

$$\#1 \equiv \#2 \equiv \mathsf{Node}\langle C\rangle$$
$$\#3 \equiv \#4 \equiv C$$
$$\#5 \equiv \#6 \equiv \#7 \equiv \mathsf{Set}\langle \#1\rangle$$

The code below shows the result. We should point out that these constraints were generated only by looking at the class `Graph` itself, and no external client code. The result is the ideal type. Many classes' own methods generate sufficient constraints to obtain an ideal (or close to ideal) result, even in the absence of client code.

```
class Node<A extends Object<>> { A label; }
class Set<B extends Object<>>  { B value; }

class Graph<C extends Object<>>
{
    Set<Node<C>> nodes = new Set<Node<C>>();

    void addNode(C label) {
        Node<C> n = new Node<C>();
        n.label = label;
        nodes.value = n;
    }

    Graph<Set<Node<C>>> scc() {
        Graph<Set<Node<C>>> g = new Graph<Set<Node<C>>>();
        g.addNode(nodes);
        return g;
    }
}
```

### *3.9.3 Example 4: Violated genericity*

This section presents another example in order to indicate how the algorithm handles faulty user code. In this example, a client of a potentially-generic class violates the intrinsic generic-type invariants of the class by direct assigning the wrong class of object into a public field:

```
class OpenBox {
    void set(Object v) { this.v = v; }
    Object v;    // not private!
    Object get() { return v; }
}
```

The parameterisation analysis produces:

```
class OpenBox<A extends B, B extends C, C extends Object<>> {
    void set(A v) { this.v = v; }
    B v;
    C get() { return v; }
}
```

Now let us add the 'rogue' client code, which violates the class's encapsulation and writes directly to `v`:

```
    OpenBox b = new OpenBox();
    b.set("foo");
    String s = (String)b.get();
    ..
    b.v = new Integer(3);    // wrong class!
```

The first three lines appear to be using `b` in a manner consistent with $\mathsf{OpenBox}\langle\mathsf{String}\langle\rangle\rangle$. However, the assignment on line 5 breaks this consistency.

When we run the instantiation analysis and simplify the constraints, we obtain:

```
OpenBox<#1,#2,#3> b = new OpenBox<#4,#5,#6>();
b.set("foo");
String<> s = (String<>)b.get();
..
b.v = new Integer<>(3);
```

$$\#1 \equiv \#4$$
$$\#2 \equiv \#5$$
$$\#3 \equiv \#6$$

$$\mathsf{Object}\langle\rangle \overset{=}{\leftarrow} \#1 \overset{=}{\leftarrow} \#2 \overset{=}{\leftarrow} \#3 \overset{=}{\leftarrow} \mathsf{String}\langle\rangle$$
$$\#2 \overset{=}{\leftarrow} \mathsf{Integer}\langle\rangle$$
$$\#3 \overset{\leq}{\leftarrow} \mathsf{String}\langle\rangle$$

The dual lower bounds on $\#2$ cause constraint resolution to yield the result $\#3 \equiv \mathsf{String}\langle\rangle \wedge \#2 \equiv \mathsf{Object}\langle\rangle \wedge \#1 \equiv \mathsf{Object}\langle\rangle$, giving the following translation for the line 1 of the client code:

```
OpenBox<Object,Object,String> b =
    new OpenBox<Object,Object,String>();
```

The result demonstrates that the algorithm does not rely on clients being well-behaved (with respect to encapsulation, etc) in order to give correct results. However, the cost of this 'rogue' use is that the class's inferred generic type is far from ideal.

# 4. TRANSLATION OF JAVA TO GJ

The instantiation analysis of Section 3 associates a parametric type with every declaration in the source program. The remaining step is to translate the program from Java to GJ.

This translation can be effected at source level, at the class-file level (when source code is not available, e.g., for third-party libraries), or at a user-specified combination of the two. We describe the two approaches below.

## 4.1 Generating GJ source

We described the analyses at the level of JVM bytecodes (and our prototype implementation operates at that level), but the analyses could also be performed on an AST (abstract syntax tree) instead. No matter how computed, the results can be applied to source-to-source translation so long as the class-files contain accurate Line Number and Local Variable tables. (The order of declarations distinguishes multiple declarations that appear on a single line.)

The source-to-source translating tool maintains whitespace and comments to the greatest extent possible. It replaces Java casts by GJ casts and omits redundant casts — those for which the target type of the cast is equal to, or is a superclass of, the inferred type of the expression.

This paper does not discuss the additional required for the correct handling of certain constructs: top-level assignments of static fields and instance fields (which are implicitly moved into `<clinit>` and `<init>` methods respectively), generated default constructors, hidden parameters (between outer and inner classes), the `assert` and `.class` constructs (which desugar to multiple basic blocks), etc. As one simple example, consider a declarator that introduces multiple variables, for example `Object o1 = f(), o2 = g();`. If the two variables are given different GJ types by the analysis, then the declarator `Object` must be replaced by two distinct ones.

### *4.1.1 Splitting local variables*

One possible optimisation is *local splitting*: static analysis of the flowgraph will show when a local variable is reused, i.e., it has two

or more disjoint live ranges. In such cases, we may want to *split* the variable into two, giving each live range a distinct name, and inferring the most precise type for each one [11]. Doing this before the flow-insensitive instantiation analysis can indicate when a Java programmer has reused a variable at a different type.

In this example, the local `Object o` has two live ranges, the first as `Integer`, the second as `String`:

```
Vector vi1 = new Vector(); // Vector<Integer>
vi1.add(new Integer(3));
Vector vs1 = new Vector(); // Vector<String>
vs1.add("foo");
...
Object o;
for (int i=0; i<vi1.size(); ++i)
{
    o = vi1.get(i);
    System.out.println(o);
}
for (int i=0; i<vs1.size(); ++i)
{
    o = vs1.get(i);
    System.out.println(o);
}
```

Without local-splitting, the type of `o` would remain unchanged at `Object`, and the two casts would still be required in the translated GJ code. However, local splitting permits replacing the declaration of `o` by two new declarations, one inside the body of each loop, `Integer o1` and `String o2`.

Local-splitting further complicates the treatment of declarators in the source-to-source translation, since it requires both the removal and addition of declarations, and potentially the renaming of references to variables.

### *4.1.2 Inner classes*

In GJ, inner classes (i.e., non-static nested classes), are within the scope of the type variables of their corresponding outer classes. Ideally, we do not wish to duplicate the implicit passing of type parameters, just as in regular Java we do not wish to duplicate the implicit passing of value parameters from outer instances to inner instances.

If we can observe that, for every allocation of the form `outer.new` $C\langle T_1, \ldots, T_n \rangle$ for a given $C$, there are some parameters in common between $T_i$ and the type of the expression `outer`, then those parameters can be eliminated (by fusion with the outer class parameter) and removed from the definition of, and every reference to, the inner class.

## 4.2 Retrofitting class-files

GJ is based upon *type erasure*; that is, the GJ compiler, after typechecking, discards all type variable annotations, inserts casts where required, and generates identical code to that of the equivalent Java program. Java classfiles have a dual role: they are the executable, and they provide signatures for separate compilation. GJ adds generic `Signature` attributes to the class-file to permit the GJ compiler to reconstruct the generic type of the class contained within it.

The GJ *retrofitter* superimposes a generic type on an existing Java class-file (containing a pseudo-generic class). Users of third-party Java libraries who wish to work in GJ need not translate the library (which they might have no source for) by hand into GJ. They need only specify the generic type of the library's interface, a much smaller problem that requires no information of the library beyond that provided in the interface documentation. The retrofitter can then add the generic type to the library.

Our work is complementary. Using our system, generic types can be inferred automatically for classes for which only class-files are available, and then retrofitted back into those class-files.

There are some additional subtleties, such as the potential lack of debugging tables (local variable tables in particular). However techniques exist for computing a conservative set of declarations for locals [11].

# 5. RELATED WORK

This paper presents a constraint-based generic type inference algorithm for the Java language. The most closely related work is other type inference algorithms that operate in the presence of polymorphism.

Milner [17] introduced the notion of polymorphic type inference, which is fundamental to the ML programming language. The original type checking algorithm did not address object-oriented programming languages with their type hierarchies and inheritance, but subsequent work [20, 22] extends Hindley-Milner typechecking to OO languages and to many other application domains. Polymorphic type inference for object-oriented languages can be roughly categorized according to the task that it supports (reverse engineering or optimization); the variety of polymorphism (data or parametric) supported; the analysis technique (constraints or dataflow); and the type system (dynamic or static; explicit or inferred). We discuss each of these dimensions in turn before discussing the most closely related papers in greater detail.

**Analysis.** Two basic approaches to type inference are constraint resolution [2, 9, 15, 26] and abstract interpretation [24]. Constraint resolution builds a set of constraints (such as equalities or inequalities) from the problem domain (the program text), then hands the constraints off to a resolution system that returns either a simplified set of constraints or a specific solution (if only one exists). Our work uses constraint resolution, but since there are many solutions to our constraints, we must take care to produce the best solution among the possible ones.

The alternative to constraint resolution is abstract interpretation [8, 24, 23], typically implemented via dataflow. An abstract value (for instance, a set of possible run-time types) is flowed around the program, and each program operation affects the abstract value in a well-defined manner.

**Task.** Polymorphic type inference aimed at reverse engineering [25, 9] aims to broaden the applicability of a pre-existing component in order to permit it to be used in more situations. Code transformations either enable the broader applicability or provide compile-time type correctness guarantees. Our work fits in this category. Type inference for ML-style languages also arguably has primarily a software engineering goal, since its key purpose is early detection of errors that would otherwise persist until run time, if they were ever noticed at all.

A more common type inference application is optimization. Three specific applications are statically discharging run-time casts [7, 26], eliminating virtual dispatch [3], unboxing, and alias analysis [19, 18]. A high-precision context-sensitive abstract interpretation [24] serves such an application well. The analysis determines a set of possible run-time types for each static operation in the program. If there is only one run-time type, then objects of only one class ever reach that program point, and any virtual method dispatch can be inlined or converted into a static procedure call of the appropriate overriding implementation of the method. Likewise, if all run-time types that reach a check satisfy the check, then the check can be removed.

**Polymorphism variety.** Polymorphism occurs in both parametric (functional) and data varieties. Parametric polymorphism [22,

7, 21, 1, 2, 15, 26] refers to the ability of procedures to operate on arguments of arbitrary types, without caring what the specific type is; for example, length : list $\alpha \rightarrow$ int). Data polymorphism [9, 23] is the ability to store objects of different types in a variable or field. It is enabled by object-oriented subtypes or by dynamic typing [7].

C++ classes and functions, which range over both primitive and class types, manifest parametric polymorphism, whereas inheritance enables data polymorphism. In Java, parametric polymorphism (e.g., `java.util.Vector`) is implemented in terms of the data polymorphism of the `Object` hierarchy. Even though the underlying JVM remains unchanged, GJ separates these functions conceptually by eliminating the need for explicit casts. Our work, which targets GJ, is most concerned with parametric polymorphism.

Constraint-based analyses, like ours, tend to be most appropriate for detecting parametric polymorphism. Abstract interpretation deals well with data polymorphism, since the goal is to determine what types may appear in a particular variable.

**Type system.** The language's type system affects the analysis that must be performed on it. In a dynamically or implicitly typed language [20, 22, 21, 15, 10, 7, 1, 2], data polymorphism is implicit and elimination of type checks is a major motivation. However, there is little room for standard type analysis.

Statically typed languages take advantage of compile-time type checking. Type inference or reconstruction [17, 20, 15, 25] must be used even for explicitly typed languages, if the source types do not capture the analysis information; that is the case for our analysis. The Hindley-Milner algorithm was originally proposed to operate over equality constraints [17]. More recent work that extends it to object-oriented languages use subtype constraints instead of equality constraints [10, 9].

Gagnon et al. [11] present a modular, constraint-based technique for inference of static types of local variables in Java bytecode; this analysis is typically unnecessary for bytecode generated from Java code, but is sometimes useful for bytecode generated from other sources. No polymorphic types are inferred, however.

## 5.1 Generalisation for re-use

There are two notable previous papers that use automated inference of polymorphism where the application is source-code generalisation for re-use.

Since the result is source code for human consumption, rather than deductions for later analysis or optimisation, one of the primary goals is restricting the degree of polymorphism so that the results do not overwhelm the user. Typically, programs contain much more 'latent' polymorphism than that actually exploited by the program.

Siff and Reps [25] aim to translate C to C++; they detect latent polymorphism in C functions designed for use with parameters of primitive type and generalise the functions into template functions to work over arbitrary types. Recall that in C++ one can define arithmetic operators for class types. Their algorithm determines — and documents — the set of constraints imposed by the generalised function on its argument. (They give as an example the $x^y$ function `pow()`, which is defined only for numbers but could be applied to any type for which multiplication is defined, such as `Matrix` or `Complex`.) Their work focuses exclusively on generic functions, not classes, and tries to detect latent reusability; in contrast, our work seeks to enforce stronger typing where reusability was intended by the programmer. Furthermore, C++ templates need not typecheck; they operate by simple textual substitution, and only the resulting code need typecheck. Therefore, the problem is quite different.

The most closely related research to ours is Duggan's constraint-based type analysis for inferring genericity in a Java-like language [9]. Duggan gives a modular (intra-class) constraint-based parameterisation analysis for a monomorphic OO kernel language called MiniJava; the target is a polymorphic variant, PolyJava, that permits abstracting classes over type parameters. The translation makes some casts provably redundant.

We extend Duggan's work in a number of ways. He does not address abstract classes or interfaces. He does no instantiation analysis, nor doe he use client information to reduce genericity, so his discovered generic types are unusably over-generic. He assumes that within class $C$, all references to instances of class $C$ have the same parameters as `this`. His type hierarchy is a forest of trees, each of which has exactly the same number of parameters on all classes within it. (Each tree inherits from Object$\langle\rangle$ via a special-case rule.) Subclasses may neither add nor remove type parameters, and the number of parameters inferred for a tree of classes is based only on the class at the root of that tree. In Java, the generic type is rarely manifest in the base: most generic classes have relatively abstract superclasses.

# 6. CONCLUSION

## 6.1 Status and future work

We are in the midst of implementing the algorithms described in this paper. Parts of the process are now automated, but for our experiments we performed other steps by hand. We have experimented with the algorithms on a suite of test classes that prove problematic for other approaches, and also on more realistic code, such as parts of the implementation itself. When the implementation is complete, we plan to analyze larger codebases quantitatively and also to gain experience with use of the tool through case studies. That will indicate, for example, whether there is need for additional techniques (either human-assisted or automatic) to refine the results of the parameterisation analysis. For example, is full unification of type parameters advantageous, or is the technique of Section 3.7 of little practical use?

Finally, we would like to experiment how constraints are added to the instantiation analysis model. Currently, each new client adds more constraints, removing (unused) aspects of the generalisation. Rather than removing spurious aspects from the maximum generalisation, it would be interesting to compute the maximum generalisation, but to include in the results only as much of it as is actually used. The distinction is similar to that separating optimistic and pessimistic analyses.

## 6.2 Contributions

We have presented a constraint-based whole-program reverse engineering algorithm for inferring generic types from Java programs. The algorithm operates in two steps: first, it determines a generic type for each class declaration, and then it determines the actual types at which each use of the class is instantiated. The two algorithms together enable translating Java programs into semantically equivalent GJ (Generic Java) programs with generic types. Preliminary investigation of the algorithms suggests that the result is usually ideal or close to ideal (that which an experienced Java programmer would have written). Automatically-inferred generic types for Java classes will permit programmers to enjoy the benefits of parametric polymorphism — such as machine-checkable documentation of programmer intent, compile-time checking for errors, and reduced code clutter — at much lower cost than converting legacy code by hand. This is an attractive proposition with the upcoming release of Java 1.5.

Our research improves previous work in several respects. First,

our research applies to real languages: it handles all Java language features, and it accounts for limitations and features of GJ. Examples include handling of arrays and of relations between subclasses and superclasses. A toy source or destination language for the translation would have simplified the algorithms, but would not have been as practical.

Second, it uses context information from clients and from subclasses to improve its results. For instance, client-side information can refine the parameterisation of classes by discovering patterns among all uses of a class in the particular application. Similarly, the algorithms obtain parameterisations for abstract classes by combining information the concrete subclasses. And they refine superclass information based on constraints in subclasses. The use of context information eliminates unwanted generality and, due in part to self-uses, can produce ideal results even for a small or nonexistent application.

Third, it handles many realistic special cases. Our algorithms accommodate the generic specialisation and extension that may accompany inheritance. They do not assume that client uses that lie within the class being analyzed must use the same type parameters. They have limited handling of complex and recursive bounds. The algorithms are robust even in face of realistic casting scenarios such as overwidening, application invariants, and client errors.

Fourth, it determines instantiation types for clients of parametrically polymorphic classes; the algorithms push genericity results back through the whole program.

Fifth, it enables translation of (client and implementation) source code to a language with genericity, rather than (for example) producing a result only for optimization or for examination by humans.

As a result of these improvements, our research produces cleaner abstractions than other approaches to the same or similar problems. We believe it to be a promising approach to migrating programmers towards parametric polymorphism.

## REFERENCES


[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In *Static Analysis Symposium*, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP*, pages 2–26, 1995.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 324–341. ACM Press, 1996.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ specification, 1998.

[5] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[7] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, 1977.

[9] D. Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 97–113. ACM Press, 1999.

[10] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 169–184. ACM Press, 1995.

[11] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[13] JavaSoft, Sun Microsystems. Top 25 requests for enhancement to Java, 2003. `http://developer.java.sun.com/developer/bugParade/top25rfes.html`.

[14] JavaSoft, Sun Microsystems. Prototype for JSR014: Adding generics to the Java programming language v. 1.3, May 7, 2001. `java.sun.com/aboutJava/communityprocess/review/jsr014/index.html`.

[15] T. Jim and J. Palsberg. Type inference in systems of recursive types with subtyping, 1999. `http://www.cs.purdue.edu/homes/palsberg/`.

[16] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[18] R. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 2001.

[19] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference, 1997.

[20] A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 445–456, New York, NY, 1989. ACM Press.

[21] N. Oxhoj, J. Palsberg, and M. I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.

[22] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, New York, NY, 1991. ACM Press.

[23] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 324–340. ACM Press, 1994.

[24] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-145.

[25] M. Siff and T. W. Reps. Program generalization for software reuse: From C to C++. In *Foundations of Software*

*Engineering*, pages 135–146, 1996.

[26] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. *Lecture Notes in Computer Science*, 2072:99–117, 2001.